EGG-SynC: Exact GPU-parallelized Grid-based Clustering by Synchronization

Jakob Rødsgaard Jørgensen
jakobrj@cs.au.dk
Department of Computer Science
Aarhus University
Aarhus, Denmark

Ira Assent
ira@cs.au.dk
Department of Computer Science
DIGIT Aarhus University Centre for Digitalisation, Big Data and Data Analytics
Aarhus University
Aarhus, Denmark

ABSTRACT

Clustering by synchronization (SynC) is a clustering method that is motivated by the natural phenomena of synchronization and is based on the Kuramoto model. The idea is to iteratively drag similar objects closer to each other until they have synchronized. SynC has been adapted to solve several well-known data mining tasks such as subspace clustering, hierarchical clustering, and streaming clustering. This shows that the SynC model is very versatile. Sadly, SynC has an $O(T \times n^2 \times d)$ complexity, which makes it impractical for larger datasets. E.g., Chen et al. [8] show runtimes of more than 10 hours for just $n = 70,000$ data points, but improve this to just above one hour by using R-Trees in their method FSynC. Both are still impractical in real-life scenarios. Furthermore, SynC uses a termination criterion that brings no guarantees that the points have synchronized but instead just stops when most points are close to synchronizing.

In this paper, our contributions are manifold. We propose a new termination criterion that guarantees that all points have synchronized. To achieve a much-needed reduction in runtime, we propose a strategy to summarize partitions of the data into a grid structure, a GPU-friendly grid structure to support this and neighborhood queries, and a GPU-parallelized algorithm for clustering by synchronization (EGG-SynC) that utilize these ideas. Furthermore, we provide an extensive evaluation against state-of-the-art showing 2 to 3 orders of magnitude speedup compared to SynC and FSynC.

1 INTRODUCTION

Clustering is the task of grouping similar objects to identify unknown structures in the data, e.g., customer groupings, and is one of the most common data mining tasks. Clustering by synchronization (SynC) [6] is a clustering definition that can capture arbitrarily shaped clusters, requiring only a neighborhood radius ε and a threshold for the termination criterion. SynC is based on the Kuramoto model from physics which captures the natural phenomena of synchronization. In their paper, they show that SynC can capture clusters that visually stand out as actual clusters, which other clustering methods like DBSCAN [10] or k-means [11] do not. Furthermore, SynC [6] also provides a method to test increasing sizes of ε and only returns the clustering with the best score. This effectively hides ε for the user but at a much higher runtime. The SynC algorithm has shown to be versatile and has been used to solve several related data mining tasks such as outlier detection [18], hierarchical clustering [19], subspace clustering [21], and clustering streaming data [20].

The concept of clustering by synchronization is powerful, as seen in several papers [18–21], and experiments [6, 7]. However, SynC is very slow to compute due to the $O(T \times n^2 \times d)$ time complexity, where T is the number of iterations, n is the number of points, and d is the dimensionality. This complexity arises since, for each iteration, SynC computes the update using the ε-neighborhood by going through all points. FSynC [8] tries to remedy this by using an R-Tree to speedup the neighborhood query and achieves one order of magnitude speedup. However, the paper still reports that it takes more than an hour to cluster just $70,000$ points. Furthermore, SynC uses a measure r_c for synchronization; when r_c reaches 1, all points have synchronized with their neighborhoods. However, since the update never actually moves the points to the neighborhood's mean, r_c does not necessarily reach 1. SynC instead terminates whenever $r_c \geq \lambda$ which implies that not necessarily all points have synchronized and that SynC is effectively computing an approximation, with no bounds, of the definition of clustering by synchronization.

To get the necessary speed and accuracy, we provide an exact and fast algorithm. FSynC has investigated the use of data structures to speed up the computation, and this provides up to around 10× speedup. To achieve further speedup, we see great potential in using modern hardware's high computational power, such as the graphic processing unit (GPU). However, to utilize the many cores of the GPU, algorithms and data structures must adhere to the computational model of the GPU, which is vastly different from the CPU model.

Our contributions. We propose:

- A new termination criterion for SynC that guarantees that the correct clustering is found,
- a strategy for partitioning the data into a grid of cells that can be summarized in a way that lets us compute the update of each point much faster,
- a GPU-friendly grid structure that supports this summarization and balances time and space efficiency,
- and an exact and fast GPU-parallelized algorithm for clustering by synchronization that utilizes these ideas.

All our contributions are manifested in a new exact and fast GPU-parallelized Grid-based algorithm for clustering by synchronization called EGG-SynC.

2 RELATED WORK

In the literature, various approaches for data clustering are studied. SynC [6] is a clustering method that captures clusters revealed by synchronization, makes no assumption about data distribution, requires no human interaction, and allows the detection of clusters of arbitrary shape, size, and density. Density-based approaches like DBSCAN [10], and DENCLUE [12] also detect clusters of arbitrary shape and size but require a threshold for the global density of a cluster and do not capture clusters of varying density. DPC [17] and OPTICS [2] capture clusters of varying density, but require user selection of density parameters.

Some work on GPU-parallelizing clustering algorithms is based on analysis of neighborhoods, like G-DBSCAN [1], GPU accelerated OPTICS [14], and GPU-INSCY [13]. They mainly achieve speedup by precomputing neighborhoods in parallel in three stages; computing the size of each neighborhood, performing an inclusive scan to identify where each neighborhood should start and end in memory, and populating the neighborhoods. G-DBSCAN computes the neighborhood and then utilizes a GPU-parallelized breadth-first search (BFS) to assign the clusters. GPU-accelerated-OPTICS only computes the neighborhood on the GPU and the rest on the CPU. GPU-INSCY uses the neighborhoods in a subspace to prune the neighborhoods in its superspaces. An adaptation of G-DBSCAN processes multiple subspaces concurrently and grows clusters simultaneously instead of running BFS for each cluster. Precomputing neighborhoods comes at the cost of high space usage, $O(n \times \mathbb{E}[|N_\varepsilon(p)|])$. For SynC this would quickly become prohibitively large. Data points move closer to each other, implying that the expected neighborhood size becomes $O(\mathbb{E}[|N_\varepsilon(p)|]) = O(n)$. Therefore the space usage and runtime would become $O(n^2)$. Instead, we propose a space- and runtime-efficient method that both prunes and summarizes data points for computing the clustering. Moreover, because SynC changes the location of points, existing pruning strategies that rely on information about previously computed neighborhoods do not apply to SynC. We, therefore, propose a new strategy for pruning in this work.

The concept of clustering by synchronization is also used for outlier detection [18], hierarchical clustering [19], subspace clustering [21], and stream clustering [20]. However, a drawback of SynC is the complexity of $O(T \times n^2 \times d)$, which has given rise to works on improving its runtime. FSynC [8] is a version of SynC that uses the indexing structure R-Tree to support an efficient finding of the neighborhoods. This reduces the time it takes to find the neighborhoods from $O(n \times d)$ to $O(\log(n) \times d + |N_\varepsilon(x)| \times d)$. However, since SynC synchronizes the points of each cluster at a common location, the neighborhoods quickly become the size of each cluster. Even the best case, where the points are distributed equally among the k clusters, implies a $O(n/k \times d)$ runtime for each iteration. In their experiments, they show one order of magnitude speedup. However, this still implies that it takes more than an hour to run FSynC on just 70,000 data points. As the neighborhoods become denser in the later iterations, FSynC becomes slower. We propose a strategy that leverages that the center of each neighborhood becomes denser to summarize regions fully within the neighborhood and avoid looking at the points in this region.

LSSPC [22] is a variation of SynC that can handle larger datasets. This is achieved by reducing the dataset using a method called CDC, running SynC on the reduced dataset, and assigning the remaining data points to clusters. CDC works by creating a minimum enclosing ball in an expanded feature space and, while there are still points outside the ball, expanding the ball to include the point farthest away from the center. The support vectors are returned as the reduced dataset when all points are covered. After running SynC, the remaining points are assigned to the cluster with the largest overlap with the neighborhoods. The remaining points are assigned to isolated clusters or outliers.

Algorithm 1 SynC(D, ε, λ)

```
1: $t = 0, r_c = 0$
2: while $r_c < \lambda$ do
3:     for $p \in D$ do
4:         for $i = 0, ..., d-1$ do
5:             $p_i^{t+1} = p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \times \sum_{q \in N_\varepsilon(p^t)} \sin(q_i^t - p_i^t)$
6:     $r_c = \frac{1}{|D|} \sum_{p \in D} \frac{1}{|N_\varepsilon(p^t)|} \sum_{q \in N_\varepsilon(p^t)} e^{-||q^t - p^t||}$
7:     $t = t + 1$
8: return synCluster($D^t$)
```

PSynC [7] is a CPU-parallelized SynC version that reduces the runtime by partitioning the dataset into areas with a roughly equal number of points, performing SynC on each partition in parallel, and merging the results to create the full clustering. Since the neighborhood of a point may span multiple partitions, PSynC computes the means of so-called K-neighborhood regions for each point as an approximation of their location. This approximation, unfortunately, comes without a guarantee as to the deviation from the correct result. PSynC also notes that the termination criterion is not exact and uses a criterion that considers the number of clusters but still does not provide an exact termination criterion. We propose the first exact SynC algorithm that provides efficient GPU-parallel computation and scalability to large datasets without the need for approximations.

Several clustering methods are based on k-nearest neighbors (kNN) [3, 16], which fixes the size of the neighborhood to k, instead of varying the number of points as in ε-neighborhoods. Some clustering definitions based on ε-neighborhood have been adapted to kNN-neighborhood instead. This provides results faster, but only approximately, e.g., the DPC approximation Fast-DPeak [9]. Another strategy is data summarization via some suitable set of statistics. BIRCH clustering [23] approximates sets of points as micro-cluster spheres, which are clustered to create the actual result. BIRCH assigns points to micro-clusters in their processing order, and the resulting approximation quality depends on this order. In this work, we focus on exact clustering according to the SynC cluster model without any loss in accuracy.

3 CLUSTERING BY SYNCHRONIZATION

Clustering by synchronization (SynC) [6] is inspired by the natural phenomena of synchronization, e.g., a group of people with similar traits often come together and form common opinions; as time evolves, they become more similar and reach a state of local synchronization. The Kuramoto model from physics captures this interaction pattern. The basic idea of SynC is to iteratively move points closer to the points in their ε-neighborhood $N_\varepsilon(p) := \{q \in D| \, |p - q| \le \varepsilon\}$, see Algorithm 1. Given a dataset $D \in \mathbb{R}^{n \times d}$, an ε radius, threshold λ, and a γ radius, the location of points $p \in D$ is iteratively updated using a function based on the Kuramoto model:

$$p_i^{t+1} = p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \times \sum_{q \in N_\varepsilon(p^t)} \sin(q_i^t - p_i^t), \qquad (1)$$

where i is the dimension and t is the current iteration. Since the sin function is used, distances must be within 0 and $\pi/2$ for points to approach each other. Böhm et al. [6], therefore, normalize the data between 0 and 1. Throughout this paper we use *drag* and *move* as synonyms for points being updated using Equation 1.

Instead of running the algorithm until the points have fully synchronized, Böhm et al. [6] compute what they call the Cluster Order Parameter:

$$r_c = \frac{1}{|D|} \sum_{p \in D} \frac{1}{|N_\varepsilon(p^t)|} \sum_{q \in N_\varepsilon(p^t)} e^{-||q^t - p^t||}. \quad (2)$$

Where r_c approach 1 when the points have synchronized. However, $r_c = 1$ is never reached, since we use sin to update the location, instead SynC terminates whenever $r_c \geq \lambda$. Böhm et al. [6] use $\lambda = 0.999$. When the points reach a state of local synchronization, the algorithm terminates, and the sets of points synchronizing together are returned as the clusters in the final clustering. SynC assigns the clusters by going through all not yet clustered points; all points within the γ-neighborhood constitute a cluster. However, the termination criterion does not guarantee that the points that synchronize are within a γ-neighborhood and, therefore, does not guarantee a correct result nor a bounded approximation quality.

SynC shows several desirable properties. It can capture arbitrarily shaped clusters with no assumption about data distribution, size, density, or the number of clusters. Furthermore, it naturally separates outliers from the cluster without specific measures, requiring no human interaction. As with all clustering methods, SynC also has its drawbacks. For SynC, we have identified the inaccuracy in termination and cluster gathering and the long runtime. We, therefore, strive to fix the inaccuracies and reduce the runtime.

FSynC [8] tries to reduce the runtime using R-Trees, an indexing structure that supports neighborhood queries, but their experiments show that it still takes more than an hour to cluster just 70,000 points. We propose using modern hardware such as the Graphics Processing Unit to achieve a faster runtime. We also provide a GPU-friendly grid structure that can be used to correctly terminate when the points have synchronized, gather the clusters, and perform neighborhood queries. Furthermore, we propose a strategy to summarize the grid cells to achieve even higher speedup.

4 EGG-SYNC

As mentioned in Section 1, SynC is a clustering concept with many advantages; however, SynC's biggest drawback is its slowness. As shown in Section 2, there have been multiple works on making SynC faster. PSynC gains up to 160× speedup by partitioning the data and running SynC on each partition of different CPU threads, but at the cost of a less accurate result. On the other hand, FSynC does not lose accuracy and gains around 10× speedup by using the R-Tree for indexing to speed up the neighborhood query used in the update function. However, FSynC still reports more than an hour of runtimes for just 70,000 data points. Even though FSynC computes the same result as SynC, the λ-termination of SynC and FSynC does not guarantee a correct result. We are neither content with an approximative solution nor a runtime of hours for a relatively small dataset. To ensure a correct result, we propose a new termination criterion, and to achieve further speedup, we propose to utilize modern hardware such as the GPU; however, this requires developing an algorithm for a vastly different computational model. We thus present a new exact GPU-parallel grid-based algorithm for clustering by synchronization (EGG-SynC). Our proposed algorithm includes a novel exact termination criterion and proof of correctness. We also devise a GPU-friendly grid-based data structure to support neighborhood queries efficiently. To further reduce the runtime of the costly update function, we propose a strategy to summarize the points in the grid cells and use the precomputed values to reduce the number of points the update function needs to go through. We show how to compute the new update function in parallel across points. Furthermore, we use the grid structure to check the synchronization criterion, Definition 4.2, and gather the final clustering when the synchronization criterion is satisfied. At last, we collect the individual parts and propose our algorithm EGG-SynC.

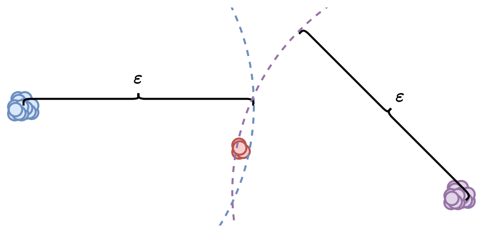


Figure 1: A cluster that should synchronize (be *dragged together* in later iterations), but where λ-termination incorrectly terminates with 3 separate clusters instead.

4.1 Exact termination criterion

SynC aims to assign points that synchronize at the same location to the same cluster. Böhm et al. [6] define points synchronizing as a cluster. SynC uses a cluster order parameter r_c, Equation 2, as a measure of local synchronization. When r_c reaches 1, all points have synchronized. However, SynC uses the sin of the distance between the points to drag them closer, Equation 1, and for $0 < x \leq 1$, $\sin(x) < x$ implying that points that are not at the same location will never reach the same location, and r_c never reach 1. Instead SynC terminate whenever r_c exceeds λ where they use a value of $\lambda = 0.999$. The λ threshold provides a termination criterion, but the approximation quality of the clustering result depends on the choice of λ. Unfortunately, no guarantees on the quality of the approximation are provided. λ-termination, as we will call it for simplicity, may indeed produce incorrect results: consider, e.g., a small cluster on the border of the ε radius of larger clusters which in later iterations can "drag" the clusters together, see Figure 1. As we can see in the figure, the issue is that the impact of the smaller cluster on the termination condition in this iteration is small, but later iterations will make these clusters synchronize into a single cluster nonetheless. Consider a dataset of 1,000,000 points, a $\lambda = 0.999$, and $\varepsilon = 0.025$, then there may be thousands of points in the small cluster, but the λ-termination criterion of r_c still reaches values above λ, even though the clusters should eventually be dragged into a single cluster. Furthermore, λ is an extra parameter that the user must set. The fact that λ-termination [6] does not indicate how close the points are to reaching their local synchronization point, is also noted by Chen et al. [7] who propose to add the number of clusters to the termination criterion. Still, their termination criterion suffers from the same fundamental problems. In this work, we propose a different approach to overcome these issues. Instead of defining an approximate measure of synchronization, we determine a state where the algorithm can safely terminate and gather points that eventually synchronize, thereby providing the first exact termination criterion. We begin with our formal definition of clustering by synchronization.

Definition 4.1 (Clustering by Synchronization). Given dataset D, parameter ε, and iterative updates using Equation 1. A non-empty $C \subseteq D$ is a cluster iff there exists an iteration t such that the following conditions are satisfied for all future iterations t':

(1) $\forall t' \geq t, \forall p, q \in D : p \in C, q^{t'} \in N_\varepsilon(p^{t'}) \Rightarrow q \in C$
(2) $\forall t' \geq t, \forall p, q \in C : q^{t'} \in N_\varepsilon(p^{t'})$

Given the Clustering by Synchronization, Definition 4.1, the state we want to capture in the synchronization criterion, Definition 4.2, is, therefore, when the neighborhoods do not change anymore, since this would imply that we know exactly which points will synchronize together when the iterations go towards infinity. To check this, we define two terms that should be satisfied for all points p. First term verifies that all points q within the ε-neighborhood of p are within the half radius $\varepsilon/2$ as well, implying that all neighborhoods either fully overlap $N_\varepsilon(q_1) \cap N_\varepsilon(q_2) = N_\varepsilon(q_1)$ or has no overlap at all $N_\varepsilon(q_1) \cap N_\varepsilon(q_2) = \emptyset$. The second term verifies that no points can be dragged into the ε-neighborhood of p using the update function. Since a point, q can only be dragged close to and not beyond all points within its neighborhood, the minimum bounding rectangle (MBR), the smallest axis-aligned hyper-cube that encloses a set of points, is a conservative approximation for where q can be moved. To conclude that q can not move into the neighborhood of p, therefore, it suffices to check that the $\varepsilon/2$-neighborhood's MBR of q does not intersect the ε radius of p.

Definition 4.2 (Synchronization Criterion). The criterion for termination is:

$$\begin{aligned}\forall p \in D : \nexists q :& (\varepsilon/2 < ||p - q|| \leq \varepsilon)\\ \wedge \nexists q :& (\varepsilon < ||p - q|| \leq \varepsilon + \delta\\ &\wedge (dist(MBR(N_{\varepsilon/2}(q)), p) \leq \varepsilon),\end{aligned}$$

where $dist(MBR, p) = \sqrt{\sum_i^d \min_{c \in MBR} |p_i - c_i|^2}$ is the smallest Euclidean distance from any corner of the MBR to points p and $\delta = \varepsilon - \varepsilon \times \sqrt{\frac{15}{16}} + \varepsilon/2 - \sin(\varepsilon/2)$ is the extra radius that must be checked, see Appendix D for proof.

We provide the following theorem and lemmas to prove that this criterion ensures that the final clustering is correctly determined. We want to prove that the first term implies that all points sharing a common neighborhood always move closer to each other. First, Lemma 4.3 proves that if the first term is satisfied for all points, then all intersecting neighborhoods are fully intersecting. Next, Lemma 4.4 proves that all points with a common neighborhood move closer to each other.

Lemma 4.3 (Identical set of neighbors.). *Given a point $p \in D$, if there does not exist a point $q \in D$ where $\varepsilon/2 \leq ||p - q|| \leq \varepsilon$, then all points $o \in N_\varepsilon(p)$ must have the same neighbors, i.e., $N_\varepsilon(p) = N_\varepsilon(o)$. See Appendix A for proof.*

Lemma 4.4 (Denser neighborhoods.). *Given points $p, q \in D$ at iteration t, if $N_\varepsilon(p) = N_\varepsilon(q)$ then $||p^{t+1} - q^{t+1}|| \leq ||p^t - q^t||$, i.e., the distance between points is smaller in subsequent iterations. See Appendix B for proof.*

We now have the foundation to prove that no points leave the neighborhood. Even though points are being updated closer to their neighborhood and, therefore, further from other points, there is still a small chance that points in a neighborhood could drag themself into another neighborhood and, by that, merge the two neighborhoods. To prove that if the second term is satisfied this can not happen we provide Lemma 4.6. Since the update Equation 1 can tilt slightly from a straight line, we need to prove that this becomes smaller in later iterations, to support this we provide Lemma 4.5.

Lemma 4.5. *Given $x, y \in (0, 1]$ and $y > x$, then:*

$$\frac{\sin(y - \sin(y))}{\sin(x - \sin(x))} > \frac{\sin(y)}{\sin(x)}. \tag{3}$$

See Appendix C for proof.

Lemma 4.6. *Given $p, q \in D$ at iteration t if the synchronization criterion, Definition 4.2, is met and $q^t \notin N_\varepsilon(p^t)$ then $\nexists t' > t$: $q^{t'} \in N_\varepsilon(p^{t'})$, i.e., no new point q can move into the neighborhood $N_\varepsilon(p)$ of any point p. See Appendix D for proof.*

We provide Theorem 4.7 to collect all the parts and conclude that when the synchronization criterion Definition 4.2 is met, we can correctly gather the final clustering.

Theorem 4.7 (Gathering Clusters). *Given $p \in D$ at iteration t, if the synchronization criterion, Definition 4.2, is met then $N_\varepsilon(p^t) = C | p \in C$, i.e., $N_\varepsilon(p)$ is the set of points that p synchronizes with and, therefore, the final cluster that p belongs to.*

Proof. By Lemma 4.3, since all points only have neighbors within the $\varepsilon/2$ neighborhood, all points must have the same points in their neighborhoods as their neighbors do. By Lemma 4.4, all points with identical neighborhoods move closer; thus, the neighborhoods never lose points. Lastly, by Lemma 4.6, no points move into the neighborhood when the synchronization criterion, Definition 4.2, is met. This implies that the neighborhoods do not change anymore. Therefore, when the synchronization criterion is met, each point's p neighborhood is the set of points that p synchronizes with and, therefore, the final cluster that the point p belongs to according to Definition 4.1. □

With this proof, we establish the first exact termination criterion for clustering by synchronization, determining a state where the algorithm can safely terminate and gather the final clusters.

4.2 GPU-friendly grid structure

Clustering algorithms have been primarily developed with the implicit assumption of a sequential single-threaded model of the CPU. Modern hardware architectures, however, employ different computational models requiring different algorithmic solutions. The modern CPU contains up to tens of cores where threads can execute individual instructions concurrently as SMT (Simultaneous Multi-Threading). It provides hardware units that can execute a single instruction on hundreds of data entries simultaneously as SIMD (Single Instruction, Multiple Data).

In this work, we propose to exploit the massive parallelism in modern GPUs (Graphics Processing Units) for efficient SynC clustering. GPUs consist of thousands of cores that provide high computational power at the cost of a more restricted computational model, where warps, groups of 32 threads, execute with a shared program counter. All threads in a warp execute the same operation as SIMT (Single Instruction, Multiple Threads). In the CUDA programming environment, threads are organized into blocks and further distributed among warps. The blocks are, furthermore, organized in a grid. Physically, cores on the GPU are grouped in SMs (Streaming Multiprocessors), which share fast access to L1-cache and can synchronize during execution. Thread in a block is executed within the same SM and therefore has the capabilities of the SM. Furthermore, all threads can access the slower main memory of the GPU, known as global memory. Due to threads accessing memory concurrently, several considerations must be taken into account. Threads that access the same memory address can lead to race conditions, and atomic operations can be used with care to avoid these; however, the atomic operation takes longer to perform. Global memory access by the same warp can be combined into one transfer if the memory accesses coalesce; this requires that the memory accesses are consecutive and aligned with global memory. In this paper, *parallel* is used to denote parallel execution on the GPU unless specified otherwise. *Computed in parallel* means distributing a for-loop among thread blocks as well as threads within each block.

The main and most time-consuming operation of Equation 1 is to compute the neighborhood of each point. Using just the dataset requires going through the entire dataset each time the neighborhood is computed. Therefore, an indexing structure is often used to speed up the neighborhood query. Tree structures, such as the R-Tree used in FSynC, support fast neighborhood queries on the CPU; however, they are not constructed with the GPU in mind. When constructing a tree, as nodes in the tree reach their maximal capacity, they are split in two, and the tree's structure is altered. If we try to construct an R-Tree in parallel, points may be inserted by some threads while the tree is being altered by others and therefore could end up in the wrong node. Furthermore, when performing a neighborhood query, each thread may need to go through multiple different parts leading to branch-divergence, which slows down performance substantially. Instead, we propose to use a grid structure to speed up the neighborhood query. This naturally reduces the runtime of the neighborhood queries, but in addition, we devise a strategy to summarize the grid cells such that we do not have to access all points during the update, following Equation 1. We also show how to check the synchronization criterion using the grid structure and gather the final clustering.

When designing the grid structure for the GPU, the main considerations are access-time, query-time, construction-time, space-usage, and how to construct and access it in parallel using warps. To make it easy to compute which grid cell a point is located within, we decide to use a grid structure with a fixed cell width c_w, implying that the ID of a cell containing a point p is

$$ID(p) = \sum_{i=0}^{d-1} \frac{p_i}{\lceil 1/c_w \rceil^i} \mod \lceil 1/c_w \rceil, \tag{4}$$

where IDs are enumerated as seen in Figure 2a. Similarly, this approach also makes it easy to determine the cells that possibly intersect the neighborhood radius ε.

In order to fully utilize the many GPU cores when constructing and accessing the grid structure, we must devise a GPU-friendly implementation. Recall that all threads within a warp must perform the same instruction at all times, or else branches diverge, slowing processing down. We must also ensure that a thread does not change part of the structure other concurrent treads are working on. Furthermore, we must also consider data access and workload distribution to achieve the highest performance. Likewise, dynamic memory allocation is relatively expensive to perform, and we aim to reduce this as much as possible.

4.2.1 List construction.

In order to handle the frequent creation of lists or sets of elements, we need to consider memory allocation. In the sequential model, elements can be added to lists on the fly and memory allocated for lists is expanded when needed. Since dynamic memory allocation is expensive on the GPU and threads in a warp wait for each other, allocating memory on the fly may quickly lead to a very high runtime. Instead, we adopt the strategy of maintaining several lists per single memory allocation. First, compute the size *sizes* of each list, then perform an inclusive scan of the sizes to find the end index *ends* of each list, allocate the total space for the elements in all lists *elements*, and at last populate each list. This requires only one memory allocation for all lists combined and makes the memory coalesced, both of which are important for efficiency. If the total size is fixed between iterations, the memory allocation can even be reused. The start index of list number i is:

$$\text{getStart}(ends, i) = \begin{cases} 0 & \text{if } i = 0 \\ ends[i-1] & \text{else} \end{cases},$$

and end index as $\text{getEnd}(ends, i) = ends[i]$. Here, we use standard C++ notation, where the end is the index after the last entry.

4.2.2 Random access.

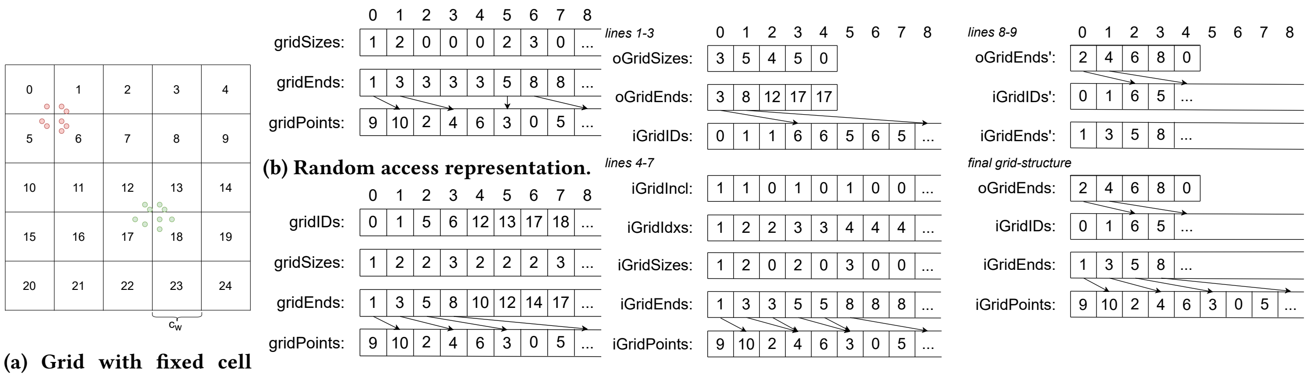
The first implementation we propose represents all possible cells in the grid structure enumerated as in Figure 2a, where the grid cell of a point is determined using Equation 4. To access the points in each cell, we compute lists of points as in Section 4.2.1, with the lists of points in *gridPoints*, their sizes in *gridSizes*, and the end index of each list in *gridEnds*.

Access. Since the grid cell width is fixed, the index of each cell can be computed in $O(d)$ time and the cell can be located by random access. The total look-up time is, therefore, $O(d)$ to access the grid cells and $O(d+|g|)$ to get all points in the grid cell. Since we do not know the non-empty cells in advance, we must look at all cells intersecting the neighborhood of a point. This makes the complexity of updating each point p $O(v^d + |N_\varepsilon(p)|)$, where $v = \lceil \varepsilon/c_w \rceil \times 2 + 1$ is the possible number of grid cells along each dimension that the ε radius can overlap with.

Construction. The random access grid structure is a set of lists of points constructed as described in Section 4.2.1. This grid structure is illustrated in Figure 2b. We discuss how to add the summarized statistics for all cells in Section 4.3.1. Since we have an index for all cells, also non-empty ones, the number of cells increases exponentially with d, i.e., $O(w^d)$ space usage, where w is the number of cells along each dimension. For lower-dimensional data, this representation is efficient because the time complexity depends on the number of dimensions. However, for higher-dimensional datasets, it leads to space issues; therefore, we propose an alternative structure for higher-dimensional data in the following.

4.2.3 Sequential access.

To reduce the space usage, we could represent the grid as a list of non-empty cells, this way we would never use more than $O(n \times d)$ space. Since there is no direct mapping between the array entries and the cells in this representation, we must keep track of which cell each entry represents. We, therefore, maintain an array *gridIDs* of the IDs of each non-empty cell, see also Figure 2c. In this paper, we view it as a single value for simplicity's sake, but in reality, these IDs can become quite large since the number of cells increases exponentially in the dimensionality. It is, therefore, represented using $O(d)$ integers in the implementation. Furthermore, we use two arrays for housekeeping to remove duplicates and tightly pack the non-empty cells. An array *gridIncl* to mark which cells should be included and an array *gridIdxs* containing the new index of the first occurrence of each non-empty cell.


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
c_w
gridSizes: 1 2 0 0 0 2 3 0 ...
gridEnds: 1 3 3 3 3 5 8 8 ...
gridPoints: 9 10 2 4 6 3 0 5 ...
gridIDs: 0 1 5 6 12 13 17 18 ...
gridSizes: 1 2 2 3 2 2 2 3 ...
gridEnds: 1 3 5 8 10 12 14 17 ...
gridPoints: 9 10 2 4 6 3 0 5 ...
lines 1-3
oGridSizes: 3 5 4 5 0
oGridEnds: 3 8 12 17 17
iGridIDs: 0 1 1 6 6 5 6 5 ...
lines 4-7
iGridIncl: 1 1 0 1 0 1 0 0 ...
iGridIdxs: 1 2 2 3 3 4 4 4 ...
iGridSizes: 1 2 0 2 0 3 0 0 ...
iGridEnds: 1 3 3 5 5 8 8 8 ...
iGridPoints: 9 10 2 4 6 3 0 5 ...
lines 8-9
oGridEnds': 2 4 6 8 0
iGridIDs': 0 1 6 5 ...
iGridEnds': 1 3 5 8 ...
final grid-structure
oGridEnds: 2 4 6 8 0
iGridIDs: 0 1 6 5 ...
iGridEnds: 1 3 5 8 ...
iGridPoints: 9 10 2 4 6 3 0 5 ...

(a) Grid with fixed cell size.

(b) Random access representation.

(c) Sequential access representation.

(d) Mixed access representation.

Figure 2: Grid representations

Access. To find a specific cell, we scan the array *gridIDs*, implying a $O(n \times d)$ access-time. However, to retrieve the neighborhood of a point p, we only need to traverse the list once, and the complexity is $O(n \times d)$ as well.

Construction. In the construction of this implementation, we additionally keep a list of non-empty grid cells but otherwise again separate it into multiple steps. We need to create all cells in parallel, but we do not know which cells are empty in advance, and we do not want to create duplicates. First, we go through all points in parallel and compute the cell ID *cID*, which is saved at the index corresponding to the point's ID $gridIDs[p] = cID$. The array *gridIDs* is now a list of all non-empty cells but possibly with duplicates. Next, to remove duplicates and compute the location of the points in each grid cell, we go through each point in parallel, compute the cell ID *cID* it belongs to and find the first entry *idx* in *gridIDs* matching *cID*. At the corresponding location *idx* in *gridSizes*, we increment the size and set *gridIncl* to TRUE to mark that this cell ID is the first of multiple duplicates and, therefore, is the one that should be included. All other duplicates can be ignored. Next, we perform an inclusive scan on *gridIncl* to find the location idx' for where each non-empty cell needs to be placed to be tightly packed and save the result in *gridIdxs*. Similarly, we perform an inclusive scan on *gridSizes* to find the end index of the cell's list of points and save it in *gridEnds*. We set *gridSizes* to zero and populate the cells' list of points as in Section 4.2.1. At last, we repack the grid structure such that all first occurrences are tightly packed at the beginning. This is done for each point in parallel. If the cell is marked as included $gridIncl[p]$, we move the end index, the size, and the cell ID to the new location $idx' = gridIdx[p] - 1$.

Theoretically, the sequential and random access grid structures have the same worst-case access time since all points could be within the neighborhood, making the query time $O(n \times d)$. However, for most datasets, we do not just have a single dense area at the neighborhood's size; therefore, the random access representation would be the fastest. On the other hand, the space complexity of the random access structure is exponential in d making it impractical for higher-dimensional datasets, where the sequential access structure uses $O(n \times d)$.

4.2.4 *Mixed access.*

Both the random access and the sequential access representation have their drawbacks. To get the best of both worlds, we propose a mix of these representations that balances the access time and the space usage. It is a heuristic to distribute the long list of grid cells into as many buckets with random access cells that we can maintain in $O(n \times d)$ space.

To get a compact representation of the grid structure, we create a random access grid structure for the first d'-dimensions only, where $w^{d'} \leq n \times d$. We refer to this partial structure as the outer-grid *oGrid*. Then for each cell in the outer-grid, we keep a list of all full-dimensional non-empty cells that fall within the outer-grid cell. We refer to these full-dimensional cells as the inner-grid *iGrid*, which is implemented as a sequential access grid structure. The outer-grid allows us to quickly locate a subset of cells in the inner-grid that potentially intersect the neighborhood. We then sequentially check each cell if it intersects the neighborhood.

Similar to the random access, the outer-grid structure consists of an array *oGridSizes* with the number of inner-grid cells, and an array *oGridEnds* with the end-locations, see Figure 2d. However, instead of indexing into the end of a list of points, it indexes into the inner-grid cells within each outer-grid cell, see Figure 2d. The inner-grid is exactly the same as the sequential access, but the inner-grid cells are grouped by the outer-grid cells.

The space use of the outer-grid is $O(n \times d)$ by choice of d' and the inner-grid can at most have n non-empty grid cells and, therefore, uses $O(n \times d)$ space. Therefore, the total space usage is $O(n \times d)$. For higher-dimensional datasets, this is much better than the $O(w^d)$ for the random access representation and as good as the sequential access strategy, but we can still access portions of the cells by random access.

Access. To retrieve the neighborhood, we identify each of the outer-grid cells intersecting the neighborhood of p this is $O(v^{d'}) = O(n \times d)$. Then we traverse the list of inner-grid cells in each outer-grid cell, we can again at most have n non-empty inner-grid cells and, therefore, this also takes $O(n \times d)$ time. In total, we use worst-case $O(n \times d)$ to query the neighborhood. However, in practical experiments, EGG-SynC performs much faster, see Section 5.

Construction. The mixed access grid structure is constructed by first building the random access grid structure for the first d' dimensions, and then for each cell, building a sequential access grid structure for the full dimensional space, as described in Algorithm 2. All arrays are allocated at the beginning of Algorithm 4 and reused in all iterations to avoid expensive memory allocations. Alongside the construction description, we provide an example of how the arrays change in Figure 2d (small captions with respective algorithm lines). To construct the outer grid, we aim to fill it with the non-empty inner grid cells. However, it is

Algorithm 2 constructGrid(D, ε)

```
1: ∀ points $p \in D$ in parallel: atomically increment the size of each outer grid cell $oGridSizes$
2: inclusive scan of $oGridSizes$ saved in $oGridEnds$
3: $\forall p \in D$ in parallel: atomically add the inner cell ID $iID$ containing $p$ to the list of inner grid cell in the outer grid cell containing $p$ with ID $oID$
4: $\forall p \in D$ in parallel: compute outer cell with ID $oID$ and inner cell with ID $iID$ containing $p$, find the first occurrence of $iID$ in the list of inner grid cell in $oID$, and mark it as included; atomically increment the size of each inner grid cell
5: inclusive scan of $iGridIncl$ saved in $iGridIdxs$
6: inclusive scan of $iGridSizes$ saved in $iGridEnds$
7: $\forall p \in D$ in parallel: atomically add $p$ to the inner grid cell
8: $\forall iIdx$ in parallel: relocate ends $iGridEnds[iIdx]$ and ids $iGridIds[iIdx]$ to new location $iIdx' = iGridIdxs[iIdx] - 1$
9: $\forall oID$ in parallel: compute new end of each outer grid cells list of inner grid cells $oGridEnds'[oID] = getStart(iGridIdxs, oGridEnds[oID])$
10: swap($iGridIDs$, $iGridIDs'$), swap($iGridEnds$, $iGridEnds'$), swap($oGridEnds$, $oGridEnds'$)
```

challenging to avoid duplicates in parallel processing. In Lines 1-3, we instead temporarily accept potential duplicates, for each point adding the ID of the inner grid cell the point is located in, then removing duplicates in Lines 4 where we mark the first occurrence of each inner cell ID to be included. To save computations and memory accesses, we count the number of points in each inner grid cell at the same time. At this stage, the first occurrence of non-empty inner grid cells is spread out sparsely in the outer grid. To compute a compact index *iGridIdxs*, Line 5 performs an inclusive scan on *iGridIncl*. In Lines 6,7, the inner grid cells are populated as in Sect. 4.2.1. Lines 8-10 repack the grid using the computed indices *iGridIdxs* into new arrays to avoid breaking the old structure while reading from it.

To conclude, we now have a data structure that uses $O(n \times d)$ space and where a neighborhood can be found in worst-case $O(n \times d)$ time, but likely much faster. This is the same as the R-Tree used in FSynC; however, our grid structure can be constructed and accessed by multiple GPU threads in parallel and supports the summarization that we discuss in Section 4.3.1.

4.2.5 Precomputing the surrounding cells. When computing the update to a point p, the thread handling this point must access the surrounding grid cells to see if they contain any points that should be included in the neighborhood. Many of these cells are empty, and for the random access strategy, this implies that some threads, in a warp, access empty cells while other threads access non-empty cells. In turn, this results in some threads waiting on other threads finishing treating the non-empty cells before continuing to the next cell. To reduce the number of idle threads, we precompute the non-empty surrounding cells of each cell in advance. The ids of the non-empty cells are saved in an array *preGridCells* similar to the points in our grid structure.

First, we compute all non-empty cells *preGridNonEmpty* in parallel by atomically incrementing the number of non-empty cells *noOfNonEmpty* and saving the outer-grid cell ID *oID* at that location. Second, in parallel for each non-empty cell *cID*, we go through the surrounding cells and count the non-empty cells $preGridSizes[cID]$. Third, to get the start and end indices *preGridEnds* of each list of surrounding outer-grid cells in *preGridCells*, we perform an inclusive scan on the counts of surrounding cells *preGridSizes* to get the end index of each list. At last, we set the counts *preGridSizes* to *zero* and, in parallel, for each non-empty cell *cID*, we go through the surrounding cells. If it is non-empty, we increment the count $preGridSizes[cID]$ to get the location $loc = atomicInc(preGridSizes[cID])$, compute the starting location $offset = getStart(preGridEnds, cID)$ in *preGridCells* and save the surrounding non-empty cells *oID* at that location *loc* plus the starting location *offset*. Having this precomputation implies that threads always work with non-empty cells; however, the number of points contained in each cell can still differ. We address this issue in the following.

4.2.6 Execution order. Precomputing the surrounding non-empty grid cells implies that no threads have a non-empty workload per cell we handle. However, we still have an unbalanced workload since some cells can contain a lot of points and others only a few. To make it more likely that threads in the same warp have a balanced workload, we aim to have warps handle points that are located close to each other. To achieve this, we leverage that the array of all points in the grid structure *iGridPoints* is sorted in order of the grid cells. Instead of updating each point in the order given in the data set, we access them in sorted order in the grid structure. This implies that it is much more likely that all threads in a warp access the same surrounding grid cells and, therefore, have a more balanced workload since the threads are likely to access the same points. However, it is still possible that threads in a warp handle points in different grid cells. It would be possible to make a warp only handle points within the same grid cells and just let the remaining threads do nothing, but this would lead to even lower utilization of the threads.

4.3 Efficient cluster algorithm on the grid

We now have the definitions needed for an exact clustering by synchronization and a GPU-friendly grid structure to support our summarization strategy. This section proposes our summarization strategy and an exact GPU-parallelized grid-based algorithm for clustering by synchronization (EGG-SynC).

4.3.1 Summarized grid cells. Supporting neighborhood queries provide speedup, but the worst-case complexity of SynC using any indexing structure is still quadratic in the number of data points. In each iteration, to update each point p requires all points in the neighborhood $N_\varepsilon(p)$, Equation 1. This is an expensive task since, as the points synchronize, the neighborhoods become larger and larger until they contain an entire cluster. If one cluster contains the majority of the points, updating each point in this cluster would take $O(n \times d)$. Even if the k clusters are of equal size, the update still takes $O(n/k \times d)$. This implies that the complexity of SynC is still $O(T \times n^2 \times d)$. The challenge is, therefore, the inherent problem of SynC, that the neighborhoods become extremely dense and that SynC has to look at all points when computing Equation 1. We propose an entirely new approach that avoids these costly computations in many cases. The core idea is to precompute summarized statistics that fulfill several requirements. Since grid cells can be fully included within multiple neighborhoods, the summarized statistics should be computed per grid cell and be reusable among points when computing the update using Equation 1. To ensure an exact result, the summarized statistics should not provide an approximation but sufficient information for correct updates. At last, it should be efficient to precompute the summarized statistics.

It is known that $\sin(y-x) = \sin(y)\cos(x) - \cos(y)\sin(x)$. We use this to rewrite the update of a point p:

$$\begin{aligned} p_i^{t+1} &= p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \sum_{q \in N_\varepsilon(p)} \sin\left(q_i^t - p_i^t\right) \\ &= p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \sum_{q \in N_\varepsilon(p)} \sin\left(q_i^t\right)\cos\left(p_i^t\right) \\ &\quad - \cos\left(q_i^t\right)\sin\left(p_i^t\right) \\ &= p_i^t + \frac{1}{|N_\varepsilon(p^t)|} \left(\cos\left(p_i^t\right) \left(\sum_{q \in N_\varepsilon(p)} \sin\left(q_i^t\right) \right) \right. \\ &\quad \left. - \sin\left(p_i^t\right) \left(\sum_{q \in N_\varepsilon(p)} \cos\left(q_i^t\right) \right) \right). \end{aligned} \tag{5}$$

The sums $\sum_{q \in N_\varepsilon(p)} \sin(q_i^t)$ and $\sum_{q \in N_\varepsilon(p)} \cos(q_i^t)$ can then be separated into two parts, one for the points in the grid cells GF fully within the neighborhood, and one for the points in grid cells GP partially within the neighborhood:

$$\sum_{q \in N_\varepsilon(p)} \sin(q_i^t) = \sum_{g \in GF} gridSin[g]_i + \sum_{g \in GP} \sum_{q \in g \cap N_\varepsilon(p)} \sin(q_i), \tag{6}$$

where $gridSin[g] = \sum_{p \in g} \sin(g)$ and analogically for cos, which are the terms that we can precompute and use in multiple updates. This implies that instead of going through all the points in the grid cell, we can look up the precomputed sums and use them, saving us a lot of time. This precomputation is computed in parallel across points, by computing the grid cell ID and adding the $\sin(p)$ and $\cos(p)$ atomically to $gridSin[g]$ and $gridCos[g]$. This makes it extremely fast in practice compared to the time it saves doing the update. The points are spread more or less equally across the neighborhood in the early iterations, but as the iterations progress, the points come closer and closer to the center. This implies that EGG-SynC becomes faster in the later iterations, as we also confirm empirically in the experiments.

4.3.2 Efficient EGG-update. Updating the location of each point p requires that we compute the ε-neighborhood of p and use it to compute the direction in which point p should move. As mentioned in related work Section 2, other GPU-parallelized algorithms precompute the neighborhood before use. However, as the neighborhoods of SynC increase in size for each iteration, the space usage becomes prohibitively expensive, and we must find alternative strategies. To balance the workload and reduce branch-divergence, we precompute the non-empty cells in Section 4.2.5, group the points by location in Section 4.2.6, and reduce point access using summarized statistics in Section 4.3.1. The update of each point, using these concepts, proceeds as in Algorithm 3. For each point p in parallel, we compute the center outer-grid cell and the center inner cell where p lies. For each surrounding outer cell, we go through each inner-grid cell. If the inner-grid cell is fully within the ε radius of p, then we can use the summarized statistics of the inner-grid cell. Else if the inner-grid cell only overlaps, we must go through all points in that inner-grid cell.

4.3.3 Termination using the grid structure. To efficiently check the synchronization criterion, Definition 4.2, we propose leveraging our grid structure. We split the criterion into two checks to reduce the amount of work that must be performed in each iteration. First, we check the first term of the criterion of whether neighborhoods either fully intersect or not at all. If the first term

Algorithm 3 EGG-update($D, \varepsilon, grid, preGrid$)

```
1:  for $p \in iGridPoints$ - in parallel do
2:      compute center outer cell ID $cOID$ of $p$
3:      compute center inner cell index $cIIdx$ of $p$
4:      for $\forall oID \in preGridCells[cOID]$ do
5:          for $\forall iIdx \in$ outer cell $oID$ do
6:              if inner cell at $iIdx$ fully within $\varepsilon$ radius of $p$ then
7:                  $sum_i = sum_i + \cos(p_i) \times iGridSin[iIdx]_i - \sin(p_i) \times$
                    $iGridCos[iIdx]_i \forall i$
8:                  $neighbors = neighbors + iGridSizes[iIdx]$
9:              else if inner cell at $iIdx$ intersect $\varepsilon$ radius of $p$ then
10:                 for $q \in$ inner cell at $iIdx$ do
11:                     $sum_i = sum_i + \sin(q_i^t - p_i^t)$
12:                     $neighbors = neighbors + 1$
13:     $p_i^{t+1} = p_i^t + \frac{1}{neighbors} \times sum_i \forall i$
14:     if $neighbors \neq iGridSizes[cIIdx]$ then
15:         $r_c = 0$
```

is satisfied, we also check the second term: no new points can be dragged into any neighborhood.

First term. We need to check if $N_{\varepsilon/2}(p) = N_\varepsilon(p)$, naively this could be done by going through all points $q \in N_\varepsilon(p)$ and checking if $\varepsilon/2 \geq |p - q|$ for any point p. However, looking at all points in the neighborhood is to be avoided, as discussed before. Instead, we aim to find a method that does not need to look at all points. The core idea is to find a lower-bound of the size of $N_{\varepsilon/2}(p)$ that can easily be computed. We propose to make the cell width $c_w \leq \sqrt{(\varepsilon/2)^2/d}$, such that the diagonal is less than $\varepsilon/2$. The grid cell g containing p is then fully within $N_{\varepsilon/2}(p)$ and we can use $|g|$ as a lower-bound of $|N_{\varepsilon/2}(p)|$ to avoid computing the exact value by terminating if $|g| = |N_\varepsilon(p)|$. This still determines the termination criterion fully correctly since this lets the algorithm run until all points within $N_{\varepsilon/2}(p)$ are also within g.

Second term. The second term is more expensive to check but only needs to be checked when the first term is true. For all points p we must go through all points in the surrounding grid cells to check if there exists any points q_1 within $\varepsilon < ||p - q_1|| < \varepsilon + \delta$. This is done in parallel across points p. Then for each pair p, q_1 in parallel, we go through all points q_2 in the surrounding grid cells to check if the minimum bounding rectangle containing q_1, q_2 intersects the ε neighborhood of p.

4.3.4 Cluster gathering. Since the λ-termination criterion does not guarantee that the points have synchronized, some points may be left out when gathering the clusters. We, therefore, propose a new method for gathering the clusters, `gatherCluster`, that guarantees that all points are assigned to the correct cluster. Recall that we only terminate when all neighbors are within the center grid cell of the neighborhood. Theorem 4.7 states that when we terminate, the neighborhoods contain all the points that synchronize together. Thus, the center grid cell of each neighborhood must contain the final cluster. Therefore, we can return all non-empty grid cells as the clustering.

4.3.5 The full algorithm. The full overview of EGG-SynC is described in Algorithm 4. While the points have not yet synchronized, we construct the grid, compute the summarized statistics, and update the points. We check if all points in the neighborhood are within the center cell, and if so, we check if the surrounding points can be dragged into the neighborhood. At last, when the points have synchronized, we gather the clusters.

Algorithm 4 EGG-SynC(D, ε)

```
1:  $t = 0, r_c = 1$
2:  while $r_c \neq 0$ do
3:      $r_c = 1$
4:      $grid$ = constructGrid($D^t, \varepsilon$)
5:      computeSinAndCosSums($grid, D^t, \varepsilon$)
6:      $preGrid$ = preComputeNonEmptyCells($D^t, \varepsilon, grid$)
7:      EGG-update($D, \varepsilon, grid, preGrid$)
8:      $t = t + 1$
9:      if $r_c = 1$ then
10:         Check second term of Def. 4.2, if not satisfied, set $r_c = 0$.
11: $grid$ = constructGrid($D^t, \varepsilon$)
12: return gatherCluster($D^t, \varepsilon, grid$)
```

5 EXPERIMENTS

We perform the experimental evaluation on a workstation with Intel Core i9 10940X 3.3GHz 14-Core, 258 GB RAM, and a GeForce RTX 3090 with 24 GB dedicated RAM. All algorithms have been implemented in C++ or CUDA, where all CUDA experiments are run with a block size of 128. For repeatability, the source code is provided at: https://au-dis.github.io/publications/EGG-SynC.

Methods. Our proposed algorithm is compared against the original algorithm for clustering by synchronization, SynC [6], and the more recent speedup FSynC [8]. For a fair comparison, we have implemented both in C++ as well. In initial experiments, our implementation of SynC provides approximately 4× speedup compared to the Java implementation provided by the authors. We have implemented straightforward SynC versions that are GPU-parallel (GPU-SynC) and CPU-parallel using multiprocessors (MP-SynC). Both parallelizations distribute updates of all points among threads. All runtime measurements for GPU algorithms also include data transfer time to GPU memory. Böhm et al. [6] employ a strategy for selecting the best ε, as Chen et al. [8], we do not include this in our experiments to make each runtime on different ε values transparent.

Hyperparameters. SynC takes parameters ε and λ; default values in all experiments are $\varepsilon = 0.05$, and $\lambda = 0.999$. EGG-SynC uses our new exact termination criterion and does therefore not need the λ parameter to terminate. FSynC, on the other hand, introduces an additional parameter, the maximum fanout B for the R-Tree; initial experiments suggest $B = 100$ performs best.

Synthetic data. We use the synthetic dataset generator provided by Beer et al. [4] to control data distribution and size, which produces Gaussian distributed clusters. The default parameters for the generated data are 100, 000 points with 2 dimensions, each dimension has values in the range −100 to 100. The points are distributed among 5 Gaussian distributed clusters existing in the full-dimensional space and with a standard deviation of 5.0.

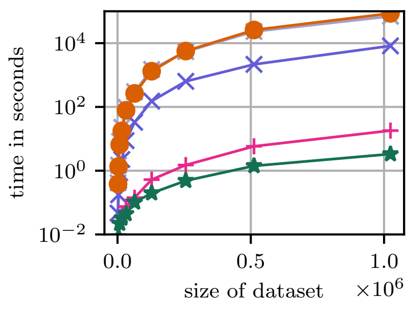
Real-world data. We study the same seven real-world datasets as Chen 2018 [8] from the UCI repository [15]; *data banknote authentication (Bank)* with 1, 372 points and 4 dimensions, *Yeast* with 1, 484 points and 8 dimensions, *Wilt* with 4, 838 points and 5 dimensions, *CCPP* with 9, 568 points and 5 dimensions, *Tamilnadu Electricity Board Hourly Readings (EB)* with 45, 781 points and 2 dimensions, *Skin_NonSkin (Skin)* with 245, 057 points and 3 dimensions, *3D_spatial_network (Roads)* with 434, 874 points and 3 dimensions. In addition, we study higher-dimensional datasets, namely, *Eye State (EEG)* with 10, 000 points and 14 dimensions, *Letter Recognition (Letter)* with 20, 000 points and 16 dimensions, both also from the UCI repository. All datasets are min/max-normalized between 0 and 1.

5.1 Performance comparison

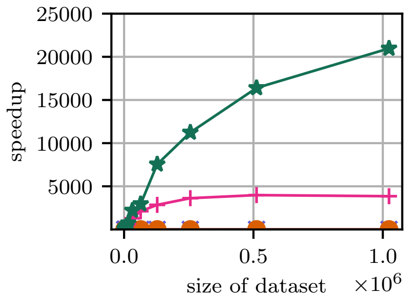
5.1.1 Scalability. We first investigate the runtime when scaling the input size in the number of points and dimensions. In Figure 3a, we see that EGG-SynC is about 2-3 orders of magnitude faster than SynC, MP-SynC, and FSynC and almost a magnitude faster than GPU-SynC. Moreover, the speedup provided by EGG-SynC over SynC and GPU-SynC, see Figure 3b, keeps increasing as the number of points increases. This can be attributed to our summarized statistics strategy since the more points we have, the higher the probability that points fall within the same cells and can be effectively summarized by our algorithm. In Figure 3c, we see that the runtime increases with the dimensionality for all algorithms and that EGG-SynC has the largest speedup for lower dimensions. We observe that all algorithms show a drop in runtime for higher dimensional datasets. As the dimensionality increases, points are more likely to be spread out, which in turn likely leads to an increased number of smaller clusters instead of a few large ones that require more synchronization, thus reducing the number of iterations required. This is in line with the effects of the curse of dimensionality [5]; when the number of dimensions increases, the points are further apart and cover more cells, meaning that speedup starts to converge at around 350× speedup. In all cases, EGG-SynC provides a substantial speedup, particularly for large datasets.

5.1.2 Distribution. Besides the size of the input data, data distribution may also affect the runtime of clustering algorithms. We, therefore, evaluate datasets with varying spread and number of clusters. In Figure 3d, we see that EGG-SynC maintains several orders of magnitude speedup compared to SynC and FSynC. Furthermore, as the number of clusters increases, all three algorithms become faster. This behavior is most apparent for FSynC and EGG-SynC and can be attributed to their use of an indexing structure for the neighborhood queries. When increasing the standard deviation of the generated clusters to study clusters with a larger spread in Figure 3e, we similarly see several orders of magnitude speedup for EGG-SynC compared to SynC and FSynC. Furthermore, the runtime is lowest for all three algorithms when the standard deviation is low. This makes sense since a smaller cluster would imply fewer iterations until the points reach the local synchronization.

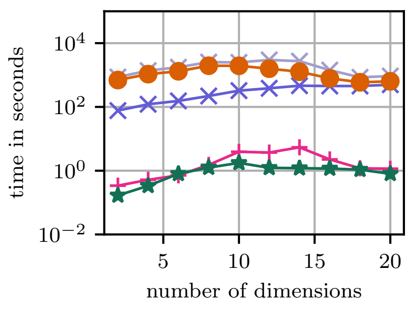
5.1.3 Real-world data. We also evaluate the performance on benchmark data from the UCI repository (Figure 4), where we again see large speedups for the GPU-parallelized versions of SynC. From the synthetic experiments, we expect EGG-SynC to be faster than GPU-SynC for the larger dataset. This is true for Roads but not for Skin. This can be explained by recalling the example in Figure 1, where a smaller part connects bigger parts of a cluster. Such a case would have a high cluster order parameter and make SynC, FSynC, and GPU-SynC stop too early, even though it could require many more iterations to cluster correctly. This is exactly what happens for the Skin dataset in this experiment (Figure 4): several clusters are approximated incorrectly but found correctly by our method, at the cost of less speedup. More concretely, GPU-SynC stops after 7 iterations, whereas EGG-SynC continues for the 343 iterations needed to find a correct clustering in this case.



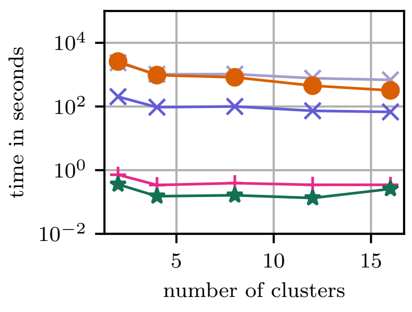
(a) Increasing size of dataset



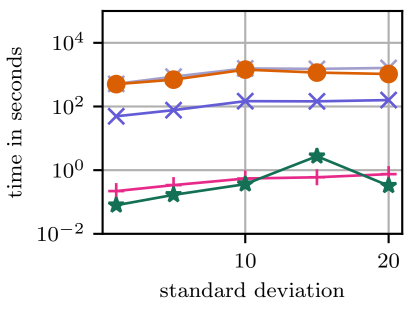
(b) Speedup as size increases



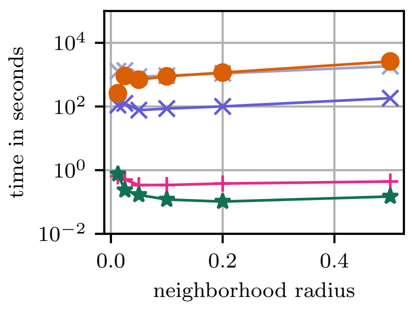
(c) Increasing dimensionality



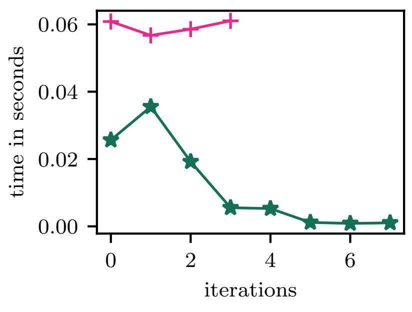
(d) Increasing number of clusters



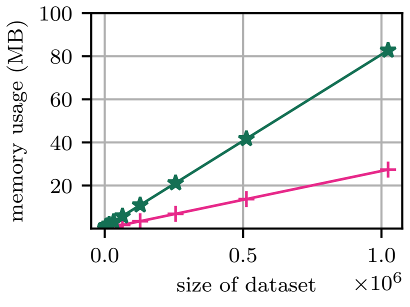
(e) Increasing standard deviation



(f) Increasing neighborhood radius



(g) Runtime per iteration.



(h) Space usage.

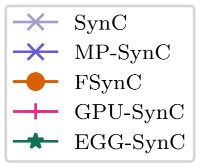


Figure 3: Synthetic experiments

We demonstrate the impact of such cluster approximations for Skin by varying neighborhood radius ε, resulting in different clustering results. As we can see in Figure 5, for other values of ε, EGG-SynC is substantially faster than GPU-SynC when there is no need to resolve slowly converging clusters.

The main take-away is thus that EGG-SynC is most often substantially faster than GPU-SynC and especially SynC. In cases where it is not, this is due to more iterations for a correct result. If speed-up should be preferred to accuracy, a termination threshold as in SynC or a maximum number of iterations could be used.

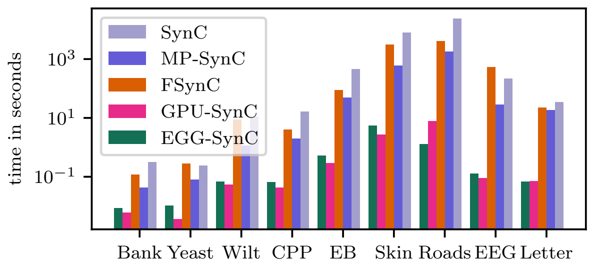


Figure 4: Real world datasets

5.2 Hyperparameters

We study the sensitivity of the runtime of the algorithms w.r.t different settings of hyperparameters. The only hyperparameter to set for the three algorithms SynC, FSynC, GPU-SynC, and EGG-SynC alike is the neighborhood radius ε. Intuitively, a lower ε implies fewer points in the neighborhood and, therefore, a lower runtime, especially for the algorithms that utilize a data structure to find the neighborhood without looking at all points. For EGG-SynC, a lower ε implies that it has to iterate over fewer points, but it also implies that each cell becomes smaller and, therefore, in the beginning, the data points are spread across more non-empty cells. In Figure 3f, EGG-SynC still provides substantial speed-up

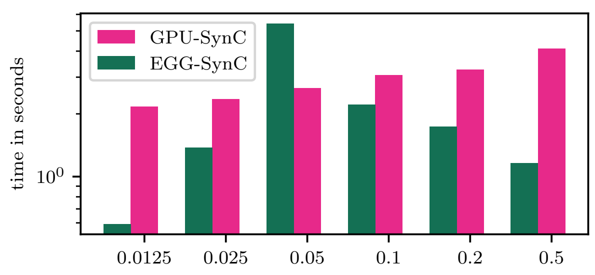


Figure 5: Changing ε for the Skin dataset

for all values of ε compared to SynC and FSynC. At very low values, the speedup for FSynC compared to SynC increases slightly and the speedup of EGG-SynC decreases slightly. However, for all other values, the speedup of EGG-SynC compared to SynC and FSynC remains several orders of magnitude.

5.3 Stage and iteration breakdown

To study how different stages of EGG-SynC contribute to the overall runtime, we provide a breakdown of the runtime of each stage of GPU-SynC and EGG-SynC, see Table 1. We see that as the data size increases, the construction time of the grid structure becomes minuscule, and the update of points strongly dominates the runtime. More importantly, we see that compared to GPU-SynC, both the update and the gathering of clusters are reduced dramatically as an effect of using the efficiently constructed grid structure. Thus, spending relatively little time on the construction of the grid structure to speed up the update of points and the gathering of clusters is clearly worth the effort.

In Figure 3g, we see that GPU-SynC's iteration becomes slightly more expensive as the iterations increase, and our EGG-SynC spends less time. This is because our summarized statistics provide more benefits for dense data. Furthermore, thanks to our effective statistics and data structure, our approach is much faster than GPU-SynC, even though it terminates later when all points have been correctly clustered according to clustering by synchronization, Definition 4.1.

5.4 Space usage

In Section 4.2.4, we state that the space usage is linear in the size of the dataset $O(n \times d)$. We validate this in Figure 3h, where we see that, indeed, the space increases linearly as the number of data points increases. As expected, EGG-SynC uses a constant factor of more space on the grid structure, which GPU-SynC does not use, but the memory consumption is reasonable and provides a clear runtime benefit.

6 CONCLUSION

In this paper, we propose a novel GPU-parallelized approach to clustering by synchronization, named EGG-SynC. EGG-SynC introduces the first exact termination criterion that guarantees that the synchronization process is finished when the algorithm terminates. EGG-SynC presents a strategy for summarizing the data in a GPU-friendly grid structure and proposes a highly efficient GPU-parallel algorithm for exact clustering by synchronization.

The experimental evaluation on synthetic and real-world data shows substantial, 2 to 3 orders of magnitude, speedup over existing algorithms for varying data and problem sizes as well as hyperparameter settings.

ACKNOWLEDGMENTS

This work is supported by Independent Research Fund Denmark.

A PROOF OF LEMMA 4.3

Proof. We prove by contradiction. Assuming that there exists a point $a \in N_\varepsilon(p)$ where $b \in N_\varepsilon(a)$ but $b \notin N_\varepsilon(p)$. First notice that $\forall p \in D, \nexists q \in D : \varepsilon/2 \leq ||p-q|| \leq \varepsilon$ implies that for all points p $N_\varepsilon(p) = N_{\varepsilon/2}(p)$. Since $a \in N_{\varepsilon/2}(p) \implies ||p-a|| \leq \varepsilon/2$ and $b \in N_{\varepsilon/2}(a) \implies ||a-b|| \leq \varepsilon/2$ and using the triangle inequality, we get $||p-b|| \leq ||p-a|| + ||a-b|| \leq \varepsilon$. However then it cannot be true that $b \notin N_\varepsilon(p)$ since it would imply that $||p-b|| > \varepsilon$. Therefore, all points in $N_\varepsilon(p)$ must have the same neighbors. □

B PROOF OF LEMMA 4.4

Proof. Given Equation 5, we have:

$$\begin{aligned}
|p_i^{t+1} - q_i^{t+1}| &= \Bigg| p_i^t + \frac{1}{|N_\varepsilon(p^t)|}\left(\cos(p_i^t)s(p,i) - \sin(p_i^t)c(p,i)\right) \\
&\quad -\left(q_i^t + \frac{1}{|N_\varepsilon(q^t)|}\left(\cos(q_i^t)s(p,i) - \sin(q_i^t)c(p,i)\right)\right)\Bigg| \\
&= \Bigg| p_i^t - q_i^t + \frac{1}{|N_\varepsilon(p^t)|} \times \left(\left(\cos(p_i^t) - \cos(q_i^t)\right)s(p,i)\right. \\
&\quad \left. - \left(\sin(p_i^t) - \sin(q_i^t)\right)c(p,i)\right)\Bigg|,
\end{aligned}$$

where $s(p,i) = \left(\sum_{y \in N_\varepsilon(p)} \sin(y_i^t)\right)$, $c(p,i) = \left(\sum_{y \in N_\varepsilon(p)} \cos(y_i^t)\right)$. For the SynC algorithm to work, Shao et al. [21] require that the data is normalized between $[0,1]$. This implies that $\sum_{y \in N_\varepsilon(p)} \sin(y_i^t)$ and $\sum_{y \in N_\varepsilon(p)} \cos(y_i^t)$ are both always positive. We split the proof into two cases, for $p_i^t - q_i^t \geq 0$ and $p_i^t - q_i^t < 0$. If $p_i^t - q_i^t \geq 0$ then $\cos(p_i^t) - \cos(q_i^t) \leq 0$ and $\sin(p_i^t) - \sin(q_i^t) \geq 0$ in the interval between $[0,1]$, implying that $|p_i^{t+1} - q_i^{t+1}| \leq |p_i^t - q_i^t|$. Similarly, if $p_i^t - q_i^t < 0$ then $\cos(p_i^t) - \cos(q_i^t) > 0$ and $\sin(p_i^t) - \sin(q_i^t) < 0$ in the interval between $[0,1]$, implying that $|p_i^{t+1} - q_i^{t+1}| < |p_i^t - q_i^t|$. We can therefore conclude that $|p_i^{t+1} - q_i^{t+1}| \leq |p_i^t - q_i^t|$. □

C PROOF OF LEMMA 4.5

Proof. Since $\frac{\sin(y-\sin(y))}{\sin(x-\sin(x))} > \frac{\sin(y)}{\sin(x)}$ implies $\frac{\sin(x)}{\sin(x-\sin(x))} > \frac{\sin(y)}{\sin(y-\sin(y))}$, and for $0 < x \leq 1$, $\frac{d}{dx}\left(\frac{\sin(x)}{\sin(x-\sin(x))}\right) = \csc(x - \sin(x))(\cos(x) + \sin(x)(\cos(x) - 1)\cot(x - \sin(x))) < 0$, which implies that $\frac{\sin(x)}{\sin(x-\sin(x))}$ is decreasing as x increases within the interval of concern, and, furthermore, that $\frac{\sin(y-\sin(y))}{\sin(x-\sin(x))} > \frac{\sin(y)}{\sin(x)}$. □

D PROOF OF LEMMA 4.6

Proof. By Lemma 4.3 all neighbors share a neighborhood when the synchronization criterion is met, and by Lemma 4.4 all points in a shared neighborhood move closer to each other, implying that they stay in the neighborhood. A neighborhood can then only expand, if points drag each other into another neighborhood using Equation 1. When all points are within $\varepsilon/2$ distance of their neighbors, this can only happen when two or more points $q_1, q_2, \ldots$ are outside ε radius of a point p, but within each other $\varepsilon/2$ neighborhoods, as illustrated in Figure 6. This implies that there is a distance from the neighborhood where there can exist points $q_1, q_2, \ldots$ that can potentially be dragged

size of dataset	Method	Allocating	Build structure	Update	Extra check	Clustering	Free Memory
256000	GPU-SynC	0.003808	0.000000	1.123219	0.000000	0.228805	0.000000
	EGG-SynC	0.000977	0.006878	0.316819	0.007083	0.000461	0.000000
512000	GPU-SynC	0.002676	0.000000	4.663134	0.000000	0.869475	0.000000
	EGG-SynC	0.001469	0.022316	1.088505	0.000403	0.001016	0.000000
1024000	GPU-SynC	0.004343	0.000000	14.145755	0.000000	3.361058	0.000000
	EGG-SynC	0.002172	0.026141	2.723254	0.000763	0.002334	0.000000

Table 1: Break down of stages.

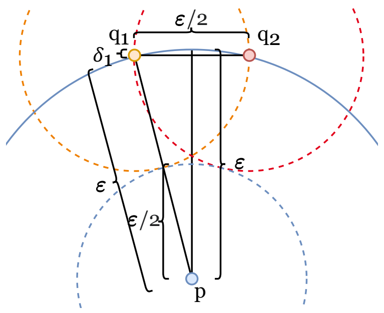
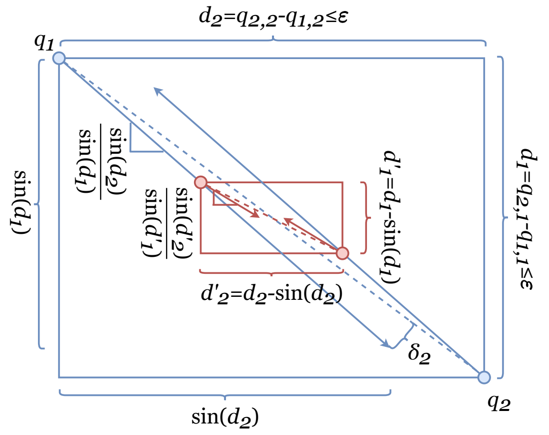


Figure 6: Example of two points q_1, q_2 dragging each other into a third point's p neighborhood.

into the neighborhood. If the location of the points was updated in a straight line, we could compute the extra distance δ_1 as:

$$\begin{aligned} \varepsilon^2 =& (\varepsilon - \delta_1)^2 + (\varepsilon/4)^2 \\ \implies (\varepsilon - \delta_1) =& \sqrt{\varepsilon^2 - (\varepsilon/4)^2} = \varepsilon\sqrt{15/16} \\ \implies \delta_1 =& \varepsilon - \varepsilon\sqrt{15/16}. \end{aligned} \tag{7}$$

However, since the update function, Equation 1, does not update the location of points in a straight line, we need to check a slightly larger extra distance $\delta = \delta_1 + \delta_2$. Since, for each dimension, the points are updated with the average sin of the difference to all other points in the neighborhood, and since sin is not a linear function, the update deviates δ_2 from a straight line. We overestimate the deviation δ_2 by considering the worst-case location and infinitely many points located at this location, i.e., when the points are the furthest apart, the distance along one dimension is as short as possible, and the other is as long as possible. When all points are within $\varepsilon/2$ distances of their neighbors, a point can at most be updated with $\sin(\varepsilon/2)$ along each dimension. This implies that the deviation δ_2 cannot exceed $\delta_2 = \varepsilon/2 - \sin(\varepsilon/2)$, due to Pythagorean theorem, illustrated in Figure 7. Furthermore, since the deviation from a straight line changes between iterations, in a subsequent iteration, the deviation could potentially intersect the ε neighborhood of point p if the slope tilts more towards p. Figure 7 illustrates updating such points at locations q_1, q_2, in two iterations. All information related to the first iteration is colored blue, and the second is colored red. In the first iteration, the point locations differ by $d_1 = q_{2,1} - q_{1,1}$ and $d_2 = q_{2,2} - q_{1,2}$ and the points at location q_1 are updated along the slope $\frac{\sin(d_2)}{\sin(d_1)}$ diverging slightly from the straight dashed line between the two locations. How far the points are updated along this slope depends on the fraction of points α that are located at q_1 compared to q_2. If there is only one point at each location, then $\alpha = 0.5$ and the points will end up close to the middle as in the illustration; else, they will end up close to the location with the most points. After the update, the points at q_1 move



d_2=$q_{2,2}$-$q_{1,2}$≤ε
q_1
$\frac{\sin(d_2)}{\sin(d_1)}$
$\frac{\sin(d'_2)}{\sin(d'_1)}$
$\sin(d_1)$
d'_1=d_1-$\sin(d_1)$
d_1=$q_{2,1}$-$q_{1,1}$≤ε
d'_2=d_2-$\sin(d_2)$
δ_2
q_2
$\sin(d_2)$

Figure 7: Deviation from straight line when updating points q_1, q_2 for two consecutive iterations.

to $q'_{1,j} = q_{1,j} + \alpha \sin(q_{2,j} - q_{1,j}) \forall i \in [0, d-1]$ and the points at q_2 move to $q_{2,j} = q_{2,j} + (1-\alpha)\sin(q_{1,j} - q_{2,j}) \forall i \in [0, d-1]$. Making the different between the points at the two locations $d'_1 = (q_{1,1} + \alpha \sin(q_{2,1} - q_{1,1})) - (q_{2,1} + (1-\alpha)\sin(q_{1,1} - q_{2,1}))$, $d'_2 = (q_{1,2} + \alpha\sin(q_{2,2} - q_{1,2})) - (q_{2,2} + (1-\alpha)\sin(q_{1,2} - q_{2,2}))$ along the two dimensions. The slope the update follow for the subsequent iteration is therefore $\frac{\sin(d'_2)}{\sin(d'_1)}$. Notice that the behavior is mirrored on the straight dashed line, and it does not matter which side of the line point p is located. We prove this in the 2-dimensional case, but it can similarly be expanded to higher-dimensions. Furthermore, how the points are located in relation to each other can be mirrored such that $0 < d_1 \leq 1$, $0 < d_2 \leq 1$, $d_1 < d_2$, and the points p is below the dashed line; the other cases can be proven analogously. To prove that the updates in the subsequent iterations do not bring the points at q_1, q_2 close to p we must show that the slope increases in the subsequent iterations $\frac{\sin(d_2)}{\sin(d_1)} < \frac{\sin(d'_2)}{\sin(d'_1)}$. We first simplify $d'_j = (q_{2,j} + \alpha\sin(q_{1,j} - q_{2,j})) - (q_{1,j} + (1-\alpha)\sin(q_{2,j} - q_{1,j})) = d_j - \alpha\sin(d_j) - (1-\alpha)\sin(d_j)) = d_j - \sin(d_j)$, implying that $\frac{\sin(d_2)}{\sin(d_1)} < \frac{\sin(d'_2)}{\sin(d'_1)} = \frac{\sin(d_2 - \sin(d_2))}{\sin(d_1 - \sin(d_1))}$. By Lemma 4.5, this is true, and the points can, therefore, not move into the ε neighborhood in subsequent iterations either. This makes the total extra distance $\delta = \delta_1 + \delta_2 = \varepsilon - \varepsilon\sqrt{15/16} + \varepsilon/2 - \sin(\varepsilon/2)$. Furthermore, if there exists a point q within this border, but the minimum bounding rectangle of the points in its neighborhood does not intersect the ε neighborhood of p, then there cannot exist two points in $N_\varepsilon(q)$ that could drag each other into the neighborhood of p. □

REFERENCES

[1] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Renato Ferreira, and Leonardo Rocha. 2013. G-dbscan: A gpu accelerated algorithm for density-based clustering. *Procedia Computer Science* 18 (2013), 369–378.

[2] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering points to identify the clustering structure. *ACM Sigmod record* 28, 2 (1999), 49–60.

[3] Anna Beer, Ekaterina Allerborn, Valentin Hartmann, and Thomas Seidl. 2021. KISS-A fast kNN-based Importance Score for Subspaces.. In *EDBT*. 391–396.

[4] Anna Beer, Nadine Sarah Schüler, and Thomas Seidl. 2019. A Generator for Subspace Clusters.. In *LWDA*. 69–73.

[5] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When is "nearest neighbor" meaningful?. In *International conference on database theory*. Springer, 217–235.

[6] Christian Böhm, Claudia Plant, Junming Shao, and Qinli Yang. 2010. Clustering by synchronization. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 583–592.

[7] Lei Chen, Jing Zhang, Li-Jun Cai, Ting-Qin He, and Tao Meng. 2016. Parallel Synchronization-Inspired Partitioning Clustering. *Journal of Computational and Theoretical Nanoscience* 13, 11 (2016), 8709–8729.

[8] Xinquan Chen. 2018. Fast synchronization clustering algorithms based on spatial index structures. *Expert Systems with Applications* 94 (2018), 276–290.

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

[10] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[11] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H Campbell. 2008. A Parallel Implementation of K-Means Clustering on GPUs.. In *Pdpta*, Vol. 13. 212–312.

[12] Alexander Hinneburg, Daniel A Keim, et al. 1998. An efficient approach to clustering in large multimedia databases with noise. In *KDD*, Vol. 98. 58–65.

[13] Jakob Rødsgaard Jørgensen, Katrine Scheel, and Ira Assent. 2021. GPU-INSCY: A GPU-Parallel Algorithm and Tree Structure for Efficient Density-based Subspace Clustering.. In *EDBT*. 25–36.

[14] Danilo Melo, Sávyo Toledo, Fernando Mourão, Rafael Sachetto, Guilherme Andrade, Renato Ferreira, Srinivasan Parthasarathy, and Leonardo Rocha. 2016. Hierarchical density-based clustering based on GPU accelerated data indexing strategy. *Procedia computer science* 80 (2016), 951–961.

[15] David J Newman, SCLB Hettich, Cason L Blake, and Christopher J Merz. 1998. UCI repository of machine learning databases, 1998.

[16] Sandra Obermeier, Anna Beer, Florian Wahl, and Thomas Seidl. 2021. Cluster Flow-an Advanced Concept for Ensemble-Enabling, Interactive Clustering. *BTW 2021* (2021).

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

[18] Junming Shao, Christian Böhm, Qinli Yang, and Claudia Plant. 2010. Synchronization based outlier detection. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 245–260.

[19] Junming Shao, Xiao He, Christian Böhm, Qinli Yang, and Claudia Plant. 2012. Synchronization-inspired partitioning and hierarchical clustering. *IEEE Transactions on Knowledge and Data Engineering* 25, 4 (2012), 893–905.

[20] Junming Shao, Yue Tan, Lianli Gao, Qinli Yang, Claudia Plant, and Ira Assent. 2019. Synchronization-based clustering on evolving data stream. *Information Sciences* 501 (2019), 573–587.

[21] Junming Shao, Xinzuo Wang, Qinli Yang, Claudia Plant, and Christian Böhm. 2017. Synchronization-based scalable subspace clustering of high-dimensional data. *Knowledge and information systems* 52, 1 (2017), 83–111.

[22] Wenhao Ying, Fu-Lai Chung, and Shitong Wang. 2013. Scaling up synchronization-inspired partitioning clustering. *IEEE Transactions on Knowledge and Data Engineering* 26, 8 (2013), 2045–2057.

[23] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1997. BIRCH: A new data clustering algorithm and its applications. *Data mining and knowledge discovery* 1, 2 (1997), 141–182.