CHALMERS
UNIVERSITY OF TECHNOLOGY

PARMA-CC: A Family of Parallel Multiphase Approximate Cluster Combining Algorithms

Journal of Parallel and Distributed Computing 177 (2023) 68–88

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

PARMA-CC: A family of parallel multiphase approximate cluster combining algorithms ☆

Amir Keramatian *, Vincenzo Gulisano *, Marina Papatriantafilou *, Philippas Tsigas *

ARTICLE INFO

Article history:
Received 4 May 2021
Received in revised form 20 September 2022
Accepted 1 February 2023
Available online 20 February 2023

Dataset link: https://github.com/dcs-chalmers/PARMA-CC

Keywords:
Parallel clustering
Approximation
Data structures
Synchronization

ABSTRACT

Clustering is a common task in data analysis applications. Despite the extensive literature, the continuously increasing volumes of data produced by sensors (e.g., rates of several MB/s by 3D scanners such as LIDAR sensors), and the time-sensitivity of the applications leveraging the clustering outcomes (e.g., detecting critical situations such as detecting boundary crossing from a robot arm that could injure human beings) demand for efficient data clustering algorithms that can effectively utilize the increasing computational capacities of modern hardware. To that end, we leverage approximation and parallelization, where the former is to scale down the amount of data, and the latter is to scale up the computation. Regarding parallelization, we explore a design space for synchronization and workload distribution among the threads. As we study different parts of the design space, we propose representative Parallel Multiphase Approximate Cluster Combining, abbreviated as PARMA-CC, algorithms. We show that PARMA-CC algorithms yield equivalent clustering outcomes despite their different approaches. Furthermore, we show that certain PARMA-CC algorithms can achieve higher efficiency with respect to certain properties of the data to be clustered. Generally speaking, in PARMA-CC algorithms, parallel threads compute summaries associated with clusters of data (sub)sets. As the threads concurrently combine the summaries, they construct a comprehensive summary of the sets of clusters. By approximating a cluster with its respective geometrical summaries, PARMA-CC algorithms scale well with increasing data volumes, and, by computing and efficiently combining the summaries in parallel, they enable latency improvements. PARMA-CC algorithms utilize special data structures that enable parallelism through in-place data processing. As we show in our analysis and evaluation, PARMA-CC algorithms can complement and outperform well-established methods, with significantly better timeliness especially when utilizing multiple threads, while still providing highly accurate results in a variety of data sets, even with skewed data distributions, which cause the traditional approaches to exhibit their worst-case behaviour.

1. Introduction

Data clustering, the task of grouping data points into sets of close-by points, is a research thread active since decades. Among many applications and use-cases, clustering algorithms are utilized in safety and management applications that monitor environments to (i) detect areas with high space contention and support decisions to e.g., minimize hazards, plan road networks or schedule transport systems, and (ii) identify objects (e.g., a self-driving vehicle) exhibiting dangerous or critical behaviour (e.g., crossing a geofence or on a collision course with an obstacle). Despite the large body of work on data clustering (e.g., [40, Ch. 11-16] and references therein), deploying such applications, critical in Internet-of-Things- (IoT-) based systems, remain challenging due to requirements such as the following:

- handling large data volumes (for example geolocation data gathered by numerous GPS (Global Positioning System) devices over a period of time and/or readings from LIDAR (Light Detection and Ranging) sensors which scan their surroundings via rotating arrays shooting laser beams, producing several MB/s of *point cloud* data),
- time constraints on data processing,
- efficient data processing for a wide range of data properties.

☆ Work supported by SSF project "FiC: Future Factories in the Cloud", grant GMT14-0032; VR project "HARE: Self-deploying and Adaptive Data Streaming Analytics in Fog Architectures", nr 2016-03800; VR project 'Models and Techniques for Energy-Efficient Concurrent Data Access Designs" Contract nr. 2016-05360.

* Corresponding authors.
E-mail addresses: amir.keramatian@gmail.com (A. Keramatian), vincenzo.gulisano@chalmers.se (V. Gulisano), ptrianta@chalmers.se (M. Papatriantafilou), tsigas@chalmers.se (P. Tsigas).

https://doi.org/10.1016/j.jpdc.2023.02.001

A parallel approach utilizing approximation can open up possibilities to appropriately address the above issues. Approximation reduces the required amount of workload at the expense of ideally small, controllable reduction of accuracy. For example, a recent work proposing MAD-C (Multi-stage Approximate Distributed Cluster-combining) [25] provides evidence regarding the advantages of approximation. MAD-C, being a distributed algorithm for approximating the Euclidean clustering algorithm, multiplicatively reduces the computational workload through approximation at the cost of marginal reduction in the clustering accuracy.

MAD-C's approximation approach aligns with the first part of the "scale down, scale up, scale out" message, summarized by Gibbons in [14], and paves the way to consider the second part, which is about proper utilization of parallelism, already omnipresent in contemporary computing architectures at all levels. To tackle this issue, in this work we address questions regarding the following: Can shared memory boost time efficiency with increased number of threads? Can work-partitioning for parallelization, time-efficiency and high-degree of accuracy co-exist? Furthermore, can adjusting the algorithm according to the data properties improve time-efficiency? Moreover, as IoT applications leverage numerous types of data with variety of different properties, can the latter affect how much the available computational capacity is utilized by an algorithm? These questions are not jointly answered in the literature (cf. also § 9).

To answer the aforementioned questions, we propose a family of Parallel Multiphase Approximate Cluster Combining methods (PARMA-CC). PARMA-CC algorithms are designed to achieve high time-efficiency and parallelization over a spectrum of different properties of data, through proper synchronization. We show how to utilize the shared memory in a way that supports parallel execution of threads sharing the workload. Because of our novel data structures and their algorithmic implementations, several operations require nearly constant time and enable incremental, in-place processing, gradually constructing the final result by connecting pieces of the data structure. We analyse the properties of PARMA-CC algorithms, and we show they all achieve equivalent clustering results. Furthermore, we study the time-efficiency and accuracy of PARMA-CC algorithms, also complementing and comparing with well-established methods such as Euclidean clustering algorithm [36], DBSCAN [11], and PDS-DBSCAN [34]. We supplement the analysis with a detailed experimental study, using both LIDAR and GPS data sets. Our results show efficiency in scaling and in preserving accuracy, even with high numbers of threads and large data sets (that can be challenging for existing clustering algorithms) and give practical evidence for the results in the analysis and the benefits of the different approaches for different properties and correlations of the data features.

The remainder of this paper is organized as follows. In § 2, we discuss the preliminaries. We outline the design ideas behind PARMA-CC algorithms in § 3. We propose the detailed algorithmic description of PARMA-CC algorithms in § 4. In § 5, we describe our proposed data structures and their algorithmic implementations. We theoretically analyze PARMA-CC algorithms in § 6. In § 7, we present a discussion regarding the trade-offs among the PARMA-CC algorithms, further use cases, and some generalizations. We present our empirical evaluation in § 8. We discuss the related work and conclusions in § 9 and § 10, respectively.

2. Preliminaries

2.1. System model and problem description

We consider a multi-core shared-memory system supporting parallel executions of `K` threads, denoted by $t_1, t_2, \cdots, t_K$. Threads access data via `read`, `write` and `read-modify-write` atomic operations. We utilize `CAS`[1] (abbreviating compare-and-swap) and `FAA`[2] (abbreviating fetch-and-add), two commonly used `read-modify-write` atomic operations, supported by all contemporary general purpose processors.

Input Data: `D` denotes the input dataset, a set of `N` points/observations, where each observation contains one or more real-valued features in a metric space (i.e., each feature corresponds to a dimension in the input space), over which distances between points can be calculated. For instance, `D` can be a *point cloud*, i.e., a set of measurements in the 3D space, gathered by one or more LIDAR sensors, or it can contain geolocation data gathered by several GPS trackers over a period of time. It is worth noting that a LIDAR sensor gathers a point cloud by targeting laser beams and measuring the time for the laser beams to get reflected back to the sensor. Furthermore, the sensor typically rotates to give a 360° view [15]. Therefore, a point cloud gathered by such a sensor is *angularly sorted* in time.

Problem Description: Given an input dataset `D`, the goal is to partition `D` into an unknown number of mutually disjoint sets (a.k.a clusters) where the points inside each cluster satisfy some predetermined distance-based or density-based criteria. To that end, we aim for an efficient, scalable parallel approximate solution to assign a clustering label to each point in `D` according to the cluster to which the point belongs. The approximation, used to reduce calculations regarding the enforcement of the distance or density criteria, must have high accuracy. As an end result, each cluster should be characterized by its point set (i.e., the cluster members) and also a *volumetric representation* of the cluster.

Objectives: To solve the aforementioned problem, we aim for a set of highly parallel, time-efficient, and scalable algorithms tailored for different data properties in order to properly utilize the available computational power. Regarding guarantees in presence of concurrency, a common consistency goal is that for every parallel execution, there exists a sequential execution that produces an equivalent result. Furthermore, the algorithms must be able to combine efficiency and accuracy benefits. Regarding efficiency, the evaluation criteria are *completion time* and utilization of parallelism which need to be considered in conjunction with the achieved *accuracy*. Taking into consideration that literature defines scalability in various ways (see for example [29]), we evaluate the parallelism-utilization properties of an approximate concurrent algorithm `A` via a *scaling-factor*, i.e., the ratio of the completion time of the sequential baseline (an exact baseline, as this is the case in the core of the algorithmic approaches here), to the completion time of the algorithm `A` running with `K` threads, for different choices of `K`. The aforementioned metric allows us to study how much the completion time of an approximate parallel algorithm changes with respect to the exact sequential baseline for different number of threads and accuracy values. We measure the accuracy of an approximate parallel algorithm with respect to the results of the exact baseline method using *rand index* [42,26], a commonly used measure of clustering similarity. Given two clusterings of the same set, *rand index* measures the ratio of the number of pairs of elements that are either clustered together or separately in both clusterings, to the total number of pairs of elements.

2.2. Background

For several of the technical parts of the algorithm descriptions, the following algorithmic and concurrency-related terms are use-

[1] `CAS(var, oldVal, newVal)` atomically changes the value stored at `var` to `newVal` if the value stored at `var` is `oldVal` and returns "true" in such a case, else it does not take any effect and returns "false".

[2] `FAA(var, delta)` atomically adds value `delta` to the value stored at variable `var` and returns the value of the variable.

ful to introduce here: A concurrent algorithm is *wait-free* if all the threads can make progress independently of each other. A concurrent implementation of a data object is *linearizable* if the effects of concurrent operations appear instantaneously and are consistent with the sequential specification of the object [20]. An operation implementation is *in-place* if it directly modifies parts of a data structure without making new copies of the latter.

We consider *distance-based* and *density-based* clustering. The points in a distance-based cluster satisfy some minimum distance criteria, and the points in density-based clusters form contiguous region of high-density, separated by contiguous low-density ones. We review PCL-EC (the Point-Cloud-Library's Euclidean clustering algorithm) [36] as a representative of a distance-based clustering. Representing density-based clustering, we cover DBSCAN [11] (Density-Based Spatial Clustering of Applications with Noise) and an established parallel variant, PDSDBSCAN [34]. We refer to PCL-EC and to DBSCAN as exact sequential distance-based and density-based baselines, respectively.

PCL-EC partitions a data set into an a priori unknown number of clusters, so that each cluster has at least $\texttt{minPts}$ points, and within each cluster, each point lies in ϵ-radius neighbourhood of at least another point in the same cluster, for parameters $\texttt{minPts}, \epsilon$. Non-clustered points are identified as *noise*. Using kd-trees for efficient neighbourhood search, PCL-EC's expected and worst-case time complexities are respectively $\mathcal{O}(\texttt{N}\log\texttt{N})$ and $\mathcal{O}(\texttt{N}^2)$, see [36, Ch. 4].

DBSCAN partitions a data set into an a priori unknown number of clusters such that a cluster consists of at least one *core point* and all the points that are *density-reachable* from it. Point p is a core point if it has at least $\texttt{minPts}$ points in its ϵ-radius neighbourhood. Point q is *directly reachable* from p if q lies in the ϵ-radius neighbourhood of p. Point q is density-reachable from p, if q is directly reachable either from p or another core point that is density-reachable from p. Non-core points that are not density-reachable from any core-points are outliers [12]. The expected and worst-case time complexities of DBSCAN are respectively $\mathcal{O}(\texttt{N}\log\texttt{N})$ and $\mathcal{O}(\texttt{N}^2)$ [39].

PDSDBSCAN [34] is a parallel version of DBSCAN. It parallelizes the work through partitioning the points and merging partial clusters consisting of points, maintained via a *disjoint-set* data structure, that facilitates maintaining a collection of disjoint sets supporting in-place *union* and *find* operations [10, Ch. 21.1]

3. The PARMA-CC family of algorithms

Clustering is a global aggregate function, and as such it is far from being an embarrassingly-parallel application; hence, concurrency (parallel tasks working on subsets of data) and synchronization (putting together the results of the data subsets) imply natural trade-offs. We propose PARMA-CC algorithms, abbreviating Parallel Multiphase Approximate Cluster Combining, to explore the design space for parallelism, in conjunction with appropriately designed data structures, to provide alternative options for different scenarios.

For the exposition of the algorithms, we consider LIDAR data as it enables more intuitive descriptions. Nonetheless, the algorithms can process various types of data, and we evaluate them with LIDAR and GPS data.

3.1. High-level view

On one side of the design space, the algorithms in the family target a *coarse-grained synchronization* approach through which operations on disjoint elements are performed in a data parallel fashion, but operations on the shared elements are performed in a mutually exclusive manner. On the other side of the design space, the algorithms target a *fine-grained synchronization* approach through which operations are performed in a fully concurrent fashion in a wait-free manner. The coarse-grained synchronization approach utilizes a scheme for data access control that can take advantage of a work-saving mechanism while the fine-grained synchronization approach eliminates the inherent waiting that is present in the more coarse-grained synchronization one. Furthermore, based on an orthogonal aspect, the algorithms in the family leverage either a *static* or *work-pool* based strategy for workload distribution. Fig. 1 visualizes the aforementioned aspects of the design space.

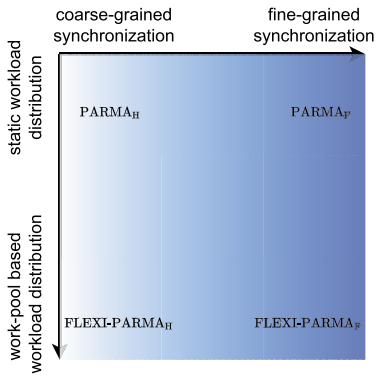


Fig. 1. The design space of PARMA-CC.

For $\texttt{i} \in \{1, \cdots, \texttt{S}\}$, $\texttt{d}_\texttt{i}$s denote mutually disjoint subsets of $\texttt{D}$, where each $\texttt{d}_\texttt{i}$ is a *split* of $\texttt{D}$, and $\texttt{S}$ is the *number of splits*. Each $\texttt{d}_\texttt{i}$ can be, e.g., the i-th chunk of $\texttt{N}/\texttt{S}$ consecutive points in $\texttt{D}$. Fig. 2a shows a hypothetical dataset $\texttt{D}$ being split into four splits. In a PARMA-CC algorithm, $\texttt{K}$ threads in parallel cluster the splits and summarize the locally detected clusters. Afterwards, the threads combine the local summaries to create a holistic summary. Lastly, according to the combined summary, points in $\texttt{D}$ are relabeled. Alg. 1 shows the high-level description of a PARMA-CC algorithm. We will see how each PARMA-CC algorithm is designed based on its position in the design space in Fig. 1. Furthermore, we will see that PARMA-CC algorithms yield equivalent clustering results.

Algorithm 1 Outline of the three phases of a PARMA-CC algorithm.

```
1: let K be the number of CPU threads
2: let d_1, ···, d_S be splits of D
3: let F be an appropriately designed shared data structure
4: for all K threads in parallel do
5:     phase I:
6:         while ∃ d_i to be clustered do
7:             cluster d_i and summarize its local clusters in F
8:             index the summaries in split-summary φ_i
9:             announce the creation of φ_i
10:    phase II:
11:        create objects by detecting and grouping matching summaries in F
12:    phase III (starts when all threads have reached here):
13:        while ∃ d_i to be relabeled do
14:            relabel the points in d_i according to the combined results
```

3.1.1. Challenges

The high-level design in Algorithm 1 implies challenges regarding data structures, workload distribution, and synchronization, outlined in the following:

Data Structures: The design and algorithmic implementation of the shared data structure in Algorithm 1 (denoted by $\mathbb{F}$) must be carried out with regard to concurrent updates by several threads as well as in-place operations to further facilitate scaling-up through shared memory parallelism.

Workload Distribution: The workload distribution and *work-partitioning* mechanism among the threads in a PARMA-CC algorithm must

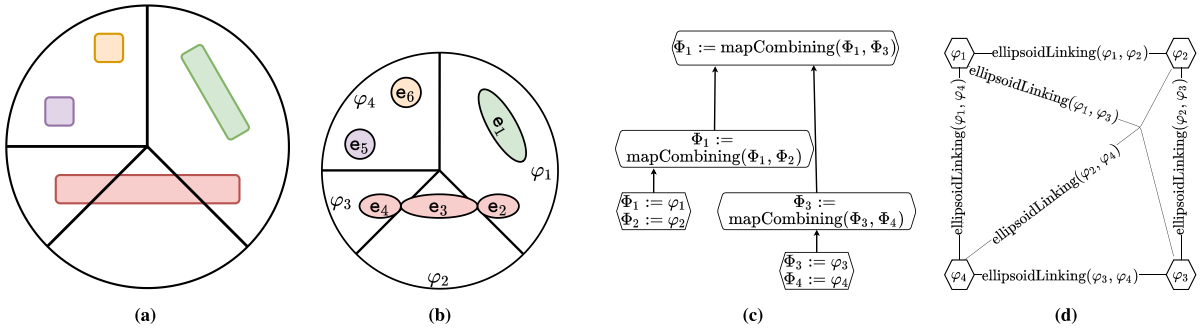


Fig. 2. (a) Dataset split into four splits. (b) Split summaries of local clusters. (c) Hierarchy $\mathbb{H}$ of `mapCombining` operations in a hierarchical PARMA-CC algorithm. (d) `ellipsoidLinking` operations in a flat PARMA-CC algorithm.

facilitate effective collaboration with minimal synchronization and contention overheads.

Synchronization: The synchronization and communication among the threads in all PARMA-CC algorithms must ensure consistency (in the final outcome) despite the diverse algorithmic choices suggested in the design space in Fig. 1.

3.2. Rudiments and definitions

Here we provide general details relevant to all PARMA-CC algorithms. For better intuition, we use a summarization technique utilizing *bounding ellipsoid*s for the exposition of the methods. However, we will see in § 7 that the choice of the summarization technique is orthogonal to the behaviour of PARMA-CC algorithms.

Definition 1. [Objects, Split-summaries, Maps]

- A *local cluster* is a cluster of points identified by a clustering algorithm (e.g., DBSCAN or PCL-EC) performed on a split of the input data. A *bounding ellipsoid* is a volumetric summary of a local cluster.
- A pair of ellipsoids $\langle \mathtt{e}_1, \mathtt{e}_2 \rangle$ can *overlap directly* or *indirectly*; $\mathtt{e}_1$and $\mathtt{e}_2$ directly overlap if $\mathtt{e}_1$ and $\mathtt{e}_2$ geometrically overlap; $\mathtt{e}_1$ and $\mathtt{e}_2$ indirectly overlap if there is an ellipsoid $\mathtt{e}'$ such that both pairs $\langle \mathtt{e}_1, \mathtt{e}' \rangle$ and $\langle \mathtt{e}_2, \mathtt{e}' \rangle$ overlap, either directly or indirectly.
- A *split-summary* $\varphi_\mathtt{i}$ is a set of ellipsoids corresponding to the detected clusters in the i-th split. Fig. 2b shows the split-summaries corresponding to the data splits in Fig. 2a.
- An *object* consists of a set of mutually overlapping ellipsoids. Given an ellipsoid $\mathtt{e}$, let $\mathbb{O}_\mathtt{e}$ denote the object in which $\mathtt{e}$ belongs.
- Two objects *overlap* if there is at least a pair of overlapping ellipsoids (one in each object). Two overlapping objects can get *merged*, forming a bigger object containing all the ellipsoids in the original objects.
- A *map* is a set of objects. Fig. 3 shows several maps.

At the heart of each PARMA-CC algorithm lies a shared data structure called the *ellipsoid forest*, denoted by $\mathbb{F}$ in Algorithm 1. An ellipsoid forest enables *multi-threaded in-place processing and access to ellipsoids, supporting efficient indexing and retrieval of objects in maps and ellipsoids in objects and split-summaries*. At the end of phase I, each ellipsoid, summarizing a local cluster, becomes a *singleton* in the ellipsoid forest upon creation. As the forest evolves in phase II, overlapping ellipsoids get grouped together, i.e., by forming objects.

Definition 2. [Ellipsoid Forest - Extended Disjoint Set Data Structure] We propose to implement the ellipsoid forest as an *extended disjoint-set data structure* [10, Ch. 21], i.e., a data structure that can store disjoint sets of ellipsoids, representing growing *objects*. Here, in a disjoint-set, a *tree* represents an object, and the root of a given tree is called the *representative* of the associated object. Similar to a disjoint-set, an ellipsoid forest supports the following operations: (i) `findRoot` returns the representative of the object containing a given ellipsoid, and (ii) `merge` replaces two given objects with their union.

We propose two extensions of the disjoint-set data structure, resulting in two variants of an ellipsoid forest data structure, in particular through the following:

- operation `mapCombining`, which, given maps $\Phi_\mathtt{i}$ and $\Phi_\mathtt{j}$, for each $\mathbb{O}$ in $\Phi_\mathtt{i}$ and $\mathbb{O}'$ in $\Phi_\mathtt{j}$, merges $\mathbb{O}$ and $\mathbb{O}'$ if they overlap, and it returns a new map that indexes the resulting objects (merged and not merged objects of $\Phi_\mathtt{i}$ and $\Phi_\mathtt{j}$). The operation is to be invoked in a synchronized, hierarchical order, to produce a final map by combining evolving partial maps, and hence we name the extended data structure *hierarchical ellipsoid forest*; or
- operation `ellipsoidLinking`, which, given split-summaries $\varphi_\mathtt{i}$ and $\varphi_\mathtt{j}$, for each pair of ellipsoids $\mathtt{e}$and $\mathtt{e}'$ in $\varphi_\mathtt{i}$ and $\varphi_\mathtt{j}$, if they overlap, merges the objects they are part of, i.e., $\mathbb{O}_\mathtt{e}$ and $\mathbb{O}_{\mathtt{e}'}$. The operation does not return any index, it only updates internal links in the composite data structure. It can be invoked concurrently in an asynchronous fashion, to perform linking between all pairs of split-summaries, and hence we name the extended data structure *flat ellipsoid forest*.

Table 1 summarizes the ellipsoid forests' extended API.

3.2.1. The three phases of a PARMA-CC algorithm

Having introduced the concept of the ellipsoid forest, we give a refined outline of a PARMA-CC-family algorithm.

Phase I: The goal here is to efficiently organize volumetric summaries of local clusters in the shared memory, facilitating efficient operations in phase II. To that end, the threads collaboratively cluster the data splits and create the split-summaries $\varphi_\mathtt{i}$s. The aforementioned steps are outlined in Algorithm 1 l.6-9.

Phase II: The objective in this phase, outlined in Algorithm 1 l. 11, is to concurrently detect and group overlapping ellipsoids in the ellipsoid forest in a scalable manner.

Phase III: This phase's objective is to assign clustering labels to points in `D` such that all points that relate to the same object are given the same label, different from labels of points that belong to other objects. Therefore, to relabel the points associated with the ellipsoids in a certain object, we use the identity of the root of the associated object in the ellipsoid forest, retrieved by `findRoot`. To make sure that the objects in the ellipsoid forest do not change any more when this phase is executed, a thread should start its phase III only after all threads have finished their phase II. The aforementioned steps are outlined in Algorithm 1 l.13-14.

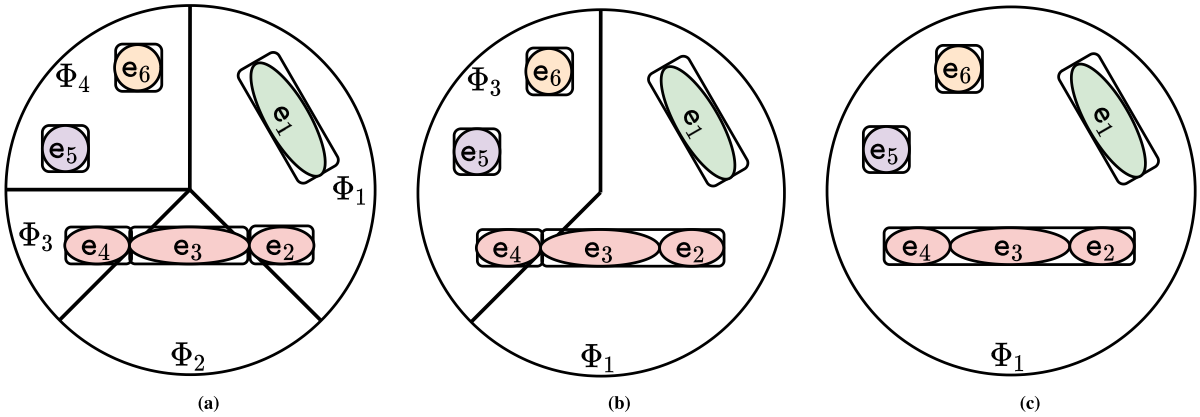


Fig. 3. Maps in a hierarchical PARMA-CC algorithm. Delimiting boxes indicate detected objects in each map. (a) Initial contents of the maps. (b) Maps Φ_1 and Φ_3 after operations Φ_1:=`mapCombining`(Φ_1,Φ_2) and Φ_3:=`mapCombining`(Φ_3,Φ_4), respectively. (c) Φ_1 after operation Φ_1:=`mapCombining`(Φ_1,Φ_3).

Table 1
Ellipsoid Forest's Extended API (the algorithmic implementations are presented in § 5).

operation	forest type	description
`mapCombining`(Φ_i, Φ_j)	hierarchical	for each $\mathbb{O}$ in Φ_i and each $\mathbb{O}'$ in Φ_j, merges $\mathbb{O}$ and $\mathbb{O}'$ if $\mathbb{O}$ and $\mathbb{O}'$ overlap, returns the combined map
`ellipsoidLinking`(φ_i, φ_j)	flat	for each e in φ_i and each e' in φ_j, merges objects $\mathbb{O}_e$ and $\mathbb{O}_{e'}$ if $\mathbb{O}_e$ and $\mathbb{O}_{e'}$ overlap

The algorithmic details of phases I-III are determined based on the choices in the design space of PARMA-CC algorithms. Particularly, the algorithmic details of phases I and III are determined based on the workload distribution aspect of the design space, and the algorithmic details of phase II are determined based on the synchronization aspect of the design space. We elaborate on the two aspects of the design space in the following subsection.

3.3. The design space of PARMA-CC algorithms

We explained in § 3.1 that PARMA-CC algorithms cover a design space concerning two orthogonal aspects: (i) synchronization, and (ii) workload distribution. We study the synchronization and workload distribution aspects of the design space in § 3.3.1 and § 3.3.2, respectively.

3.3.1. Synchronization via hierarchical ellipsoid forest vs synchronization via flat ellipsoid forest

The synchronization aspect of the design space mainly concerns the manner in which overlapping ellipsoids are detected and grouped together. As ellipsoids are stored in the ellipsoid forest, the key element regarding this aspect is the ellipsoid forest. As noted in Definition 2, there are two forest types, named hierarchical and flat. From now on, a hierarchical PARMA-CC algorithm is one that utilizes the hierarchical forest, and a flat PARMA-CC algorithm is one that utilizes the flat forest.

In a hierarchical PARMA-CC algorithm, the order of performing operations is synchronized via a tree $\mathbb{H}$, spanning over nodes that represent maps in the process of constructing the overall outcome (cf. Definition 1). Particularly, every node in $\mathbb{H}$ instructs performing `mapCombining` on two maps. Operations in disjoint branches of $\mathbb{H}$ can be performed concurrently, but in the same branch, the order of performing the operations must follow the hierarchy, starting from the leaves and continuing upwards. Fig. 2c shows a hierarchy applicable on the maps in Fig. 3a, i.e., the maps initiated by the split-summaries in Fig. 2b. Accordingly, Fig. 3b shows the contents of maps Φ_1 and Φ_3 after operations Φ_1:= `mapCombining`(Φ_1,Φ_2) and Φ_3:= `mapCombining`(Φ_3,Φ_4), respectively. Finally, Fig. 3c shows the content of Φ_1 after operation Φ_1:= `mapCombining`(Φ_1,Φ_3). In Fig. 3, the objects in each map are distinguished by delimiting boxes.

In a flat PARMA-CC algorithm, there is no need to synchronize the order of performing operations because `ellipsoidLinking` utilizes asynchronous concurrent linearizable operations (similar to [23]) which makes it possible to perform consistently the `ellipsoidLinking` operations (which are commutative and associative, as we show in the more detailed description sections) in a fully concurrent fashion on all pairs of split-summaries. Fig. 2d outlines the `ellipsoidLinking` operations corresponding to the split-summaries in Fig. 2b. Note that performing the `ellipsoidLinking` operations in Fig. 2d will result in detecting the same objects shown in Fig. 3c.

3.3.2. Basic workload distribution vs flexible workload distribution

Another key aspect of the design space (as we outlined in the beginning of the section and in Fig. 1), is the distribution of the local clustering and local relabeling tasks (i.e., phase I and phase III workload) among the threads. To that end, PARMA-CC algorithms are categorized into two groups. In the first group, which we refer to as *basic*, the workload in phase I and phase III are distributed among the threads in a *work-sharing* [8] style by statically assigning each task to a processor. In the second group, which we refer to as *flexible*, the workload is partitioned into a large number of tasks (larger than the number of threads in the system) available as a *shared pool*, from which the threads compete to take tasks in a *work-stealing* [8] fashion. From now on, a *basic PARMA-CC* algorithm is one that utilizes the basic workload distribution approach, and a *flexi PARMA-CC* is one that utilizes the flexible workload distribution approach.

Basic Workload Distribution: The basic workload distribution assumes a one-to-one relation between the number of threads (`K`) and the number of splits (`S`). Concretely, for each $i \in \{1, \cdots, K\}$, the local clustering and relabeling tasks associated with d_i are performed statically assigned to thread t_i.

Flexible Workload Distribution: Notice that in a basic workload distribution, the duration of performing the local clustering tasks can vary between splits even when they contain the same number of points and the threads are equally fast. The latter is due to the fact that the local clustering algorithm employs a spatial data structure for indexing the points. Consequently, with different distributions of points in each split, the associated cost of using the data structure varies. Hence, the threads that finish their local clustering task

earlier will have to wait in subsequent phases of the algorithm. To alleviate the aforementioned problem, a flexi PARMA-CC algorithm breaks down the local clustering and local relabeling tasks into fine-grained chunks by allowing S to be larger than K. A flexi PARMA-CC algorithm accommodates a shared pool of local clustering and relabeling tasks which can be *booked* by each thread in a wait-free manner.

4. Detailed description of PARMA-CC algorithms

We cover basic and flexi PARMA-CC algorithms in § 4.1 and § 4.2, respectively.

4.1. Basic members of the PARMA-CC family

Hierarchical Parallel Multiphase Approximate Cluster Combining, abbreviated as $\texttt{PARMA}_\texttt{H}$, and Flat Parallel Multiphase Approximate Cluster Combining, abbreviated as $\texttt{PARMA}_\texttt{F}$, are the two basic members of the PARMA-CC family. We introduce $\texttt{PARMA}_\texttt{H}$ in § 4.1.1 and $\texttt{PARMA}_\texttt{F}$ in § 4.1.2. In § 4.2, we discuss how $\texttt{PARMA}_\texttt{H}$ and $\texttt{PARMA}_\texttt{F}$ serve as a basis for the flexi members of the family.[3]

Algorithm 2 $\texttt{PARMA}_\texttt{H}$ algorithm.

```
1: let H be a combine hierarchy
2: let each mapCombining in H be uniquely associated with a thread
3: let S:=K
4: for all thread t_i | i ∈ {1, ··· , K} in parallel do
5:    phase I:
6:       cluster d_i & summarize its local clusters
7:       index the summaries in φ_i
8:       Φ_i:= φ_i
9:       signal the responsible thread on the first level of H that Φ_i is ready
10:   phase II:
11:      if t_i is responsible for mapCombining(Φ_m,Φ_n) then
12:         wait to receive signals that Φ_m and Φ_n are ready
13:         Φ_m:= mapCombining(Φ_m, Φ_n)
14:         signal the responsible thread in the next level of H (if any) that Φ_m is
            ready
15:   phase III (starts when all threads have reached here):
16:      relabel the points in d_i based on their objects
```

Algorithm 3 $\texttt{PARMA}_\texttt{F}$ algorithm.

```
1: let V be {v_{m,n} | m, n ∈ {1, 2, ··· , S}, m < n} (see Definition 3)
2: let S:=K
3: for all thread t_i | i ∈ {1, ··· , K} do
4:    phase I:
5:       cluster d_i & summarize its local clusters
6:       index the summaries in φ_i
7:       for all v ∈ V | v = v_{i,x} or v = v_{x,i} do
8:          atomically increment v //e.g., using FAA
9:    phase II:
10:      while ∃(m, n) | corresponding task to v_{m,n} not booked do
11:         if corresponding task to v_{m,n} is booked // (e.g. using CAS(v_{m,n}, 2, 3))
            then
12:            ellipsoidLinking(φ_m,φ_n)
13:   phase III (starts when all threads have reached here):
14:      relabel the points in d_i based on their objects
```

4.1.1. $\texttt{PARMA}_\texttt{H}$

$\texttt{PARMA}_\texttt{H}$ is a basic PARMA-CC algorithm that utilizes the hierarchical ellipsoid forest (see § 3.3). In $\texttt{PARMA}_\texttt{H}$, each mapCombining in hierarchy H is uniquely associated with a thread which is responsible for performing the associated mapCombining. To make sure that the contents of the maps are finalized before a thread performs its mapCombining, it must wait until it receives *signals* that the associated maps are *ready*. Algorithm 2 gives a high-level view of $\texttt{PARMA}_\texttt{H}$. We study phase I and phase II of $\texttt{PARMA}_\texttt{H}$ in the following. We avoid repeating phase III of $\texttt{PARMA}_\texttt{H}$ because it is identical to the provided details of the corresponding phase in § 3.2.1.

Phase I: After having created the split-summary φ_i, thread t_i initializes map Φ_i by the content of φ_i, as indicated in Algorithm 2 l.8. Afterwards, t_i signals the responsible thread on the first level of H that Φ_i is ready.

Phase II: Assuming t_i is responsible for mapCombining(Φ_m, Φ_n), after having received the signals that Φ_m and Φ_n are ready, t_i performs the associated mapCombining and updates Φ_m, as indicated in Algorithm 2 l.13. Then, t_i signals the responsible thread in the next level of H (if any) that Φ_m is ready, as shown in Algorithm 2 l.14.

4.1.2. $\texttt{PARMA}_\texttt{F}$

$\texttt{PARMA}_\texttt{F}$ is a basic PARMA-CC algorithm that utilizes the flat ellipsoid forest (see § 3.3). In $\texttt{PARMA}_\texttt{F}$ the elements in the ellipsoid forest are accessed and modified in a fully concurrent manner, i.e. no ordering is required. The latter holds because $\texttt{PARMA}_\texttt{F}$ utilizes the ellipsoidLinking operations to detect and group the overlapping ellipsoids. We will cover the algorithmic implementation of the data structure associated with ellipsoidLinking and their consistency guarantees in the presence of concurrent operations in § 5.3.

In $\texttt{PARMA}_\texttt{F}$, the ellipsoidLinking tasks are distributed among the threads based on the availability of tasks and the availability of unoccupied threads. To that end, each thread, after having performed the local clustering task and having created the associated split-summary, should reveal the availability of the new split-summary and the associated ellipsoidLinking tasks to the rest of the threads, so the unoccupied threads can perform the associated tasks. We propose to utilize an array V, for storing the *status* of the ellipsoidLinking tasks:

Definition 3. Let V be a set of *status* values, each one associated with an ellipsoidLinking task on a certain pair of maps. As ellipsoidLinking is symmetric, V is defined as $\{v_{m,n} | m, n \in \{1, 2, \cdots, S\}, m < n\}$, where $v_{i,j}$ indicates the status of ellipsoidLinking(φ_m, φ_n) and can have any of the following values: 0: when neither φ_m nor φ_n is created (initial value); 1: when one of φ_m or φ_n is ready; 2: when both φ_m and φ_n are ready, but ellipsoidLinking(φ_m, φ_n) is not yet booked, 3: when ellipsoidLinking(φ_m, φ_n) is booked (to be performed by the thread that booked it).

To make sure concurrent updates on V are performed correctly, the threads use *atomic synchronization primitives* to update the status values.

Algorithm 3 outlines $\texttt{PARMA}_\texttt{F}$. We study phase I and phase II of $\texttt{PARMA}_\texttt{F}$ in the following. We avoid repeating phase III of $\texttt{PARMA}_\texttt{F}$ because it is identical to the provided details of the corresponding phase in § 3.2.1.

Phase I: In this phase, after having created split-summary φ_i, thread t_i updates the status values of the affected ellipsoidLinking tasks. To that end, it atomically increments the status values of each affected task (e.g., by performing FAA), as shown in Algorithm 3 l.7-8.

Phase II: A thread in this phase keeps iterating through the status values in V. As the thread finds a task which is not booked yet, it tries to atomically *book* the task (e.g., via CAS operation to change its status from two to three). Upon successful booking, the thread performs the respective task. The aforementioned steps are shown in Algorithm 3 l.10-l.12.

[3] The gray lines in the pseudocodes indicate parts that have been described in previous sections and are marked in this way to facilitate the focus of the different parts of each algorithm.

4.2. Flexi members of the PARMA-CC family

A flexi PARMA-CC targets *flexible workload distribution* among the threads, in particular dividing the local clustering and local relabeling tasks into fine-grained chunks, through allowing S to be larger than K. Following the discussion in § 3.3.2, the flexible workload distribution decreases the potential amount of the threads' waiting time; therefore, it increases the utilization of resources. We introduce FLEXI-PARMA$_H$, and FLEXI-PARMA$_F$, flexi versions of PARMA$_H$ and PARMA$_F$, respectively.

4.2.1. Flexi shared phases

As there is not a one-to-one correspondence between the data splits and the threads, we design a wait-free *booking* mechanism for performing the local clustering and local relabeling tasks, which we explain in the following.

Phase I: The goal here is to perform parallel local clustering of S splits with K threads. Let LCT, abbreviating Local Clustering Tasks, be a boolean array of size S, where each index shows if the associated local clustering task has been performed. The booking mechanism is similar to the one introduced in § 4.1.2. Phase I is shown in Algorithm 5 and Algorithm 4 for the flexi PARMA-CC algorithms. The lines marked by ★ indicate the preparation step for phase II.

Phase III: The goal here is to perform parallel local relabeling of S splits with K threads. To that end, we utilize a boolean array of size S named LRT, abbreviating Local Relabeling Tasks, in the same fashion as explained for LCT. Phase III in Algorithm 5 and Algorithm 4 outline the relabeling steps in flexi PARMA-CC algorithms.

We review the uncovered details of FLEXI-PARMA$_H$ and FLEXI-PARMA$_F$, in § 4.2.2 and § 4.2.3, respectively.

4.2.2. FLEXI-PARMA$_H$

The goal of this algorithm is to reduce the amount of time that a thread waits for its descendants' maps. To that end, this algorithm utilizes an *agile* mechanism to generate the combine hierarchy $\mathbb{H}$ on the fly. Notably, $\mathbb{H}$ is determined based on the order in which the maps become available. The latter is achieved by utilizing a multithreaded queue Q, which holds the indices of the ready maps. As the preparation step (★), for each local clustering task that a thread performs, it inserts the index of the associated map in Q. In phase II, a thread tries to pop two indices from Q. If two indices are popped successfully, then it performs mapCombining on associated maps, and it will insert the index of the resulting map in Q. This process continues until (S -1) mapCombining operations are performed. To that end, the total number of performed tasks is kept as a global variable that gets incremented atomically. When (S -1) tasks are performed, there is only one map index in Q, which indexes all the objects in the forest.

In Algorithm 4 and Algorithm 5 the lines marked by ★ indicate the preparation step for phase II.

4.2.3. FLEXI-PARMA$_F$

This is a flexi version of PARMA$_F$. As the preparation step (★), for each local clustering task that a thread performs, the thread updates the status values of the affected ellipsoidLinking tasks, using the technique explained in § 4.1.2. Algorithm 5 outlines FLEXI-PARMA$_F$.

5. The ellipsoid forest data structures and algorithmic implementation

In this section, we introduce the algorithmic implementation of the ellipsoid forest data structure. We start by introducing the bounding ellipsoid data structure. Afterwards, we study the algorithmic implementation of the hierarchical and flat forests in § 5.2 and § 5.3, respectively.

Algorithm 4 FLEXI-PARMA$_H$ algorithm.

```
1: let LCT and LRT be shared arrays of size S initialized to 0
2: let Q be a multithreaded queue
3: let totalNumberOfCombines be initialized to 0
4: for all K threads in parallel do
5:   phase I:
6:     for splitID ∈ {1, ··· , S} do
7:       if CAS(LCT[splitID], 0, 1) then
8:         cluster d_splitID & summarize its local clusters
9:         index the summaries in φ_splitID
10:        Φ_splitID := φ_splitID
11:        Q.push(splitID)(★)
12:  phase II:
13:    while totalNumberOfCombines < S − 1 do
14:      if Q.tryPop(i, j) == success then
15:        Φ_i := mapCombining(Φ_i, Φ_j)
16:        FAA (totalNumberOfCombines, 1)
17:        Q.push(i)
18:  phase III (starts when all threads have reached here):
19:    for splitID ∈ {1, ··· , S}
20:      if CAS(LRT[splitID], 0, 1) then
21:        relabel the points in d_splitID based on their objects
```

Algorithm 5 FLEXI-PARMA$_F$ algorithm.

```
1: let LCT and LRT be shared arrays of size S initialized with 0
2: let V be {v_{i,j} | i, j ∈ {1, 2, ··· , S}, i < j} (see Definition 3)
3: for all K threads in parallel do
4:   phase I:
5:     for splitID ∈ {1, ··· , S} do
6:       if CAS(LCT[splitID], 0, 1) then
7:         cluster d_splitID & summarize its local clusters
8:         index the summaries in φ_splitID
9:         for all v ∈ V | v = v_{splitID,x} or v = v_{x,splitID} do
10:          FAA(v, 1)(★)
11:  phase II:
12:    while ∃(i, j) | v_{i,j} ≠ 3 do
13:      if CAS(v_{i,j}, 2, 3) then
14:        ellipsoidLinking(φ_i, φ_j)
15:  phase III (starts when all threads have reached here):
16:    for splitID ∈ {1, ··· , S}
17:      if CAS(LRT[splitID], 0, 1) then
18:        relabel the points in d_splitID based on their objects
```

5.1. The bounding ellipsoid data structure

In our design, each (bounding) ellipsoid is instantiated in the shared memory and automatically becomes an element in the forest upon creation. The data structure supporting an ellipsoid contains μ and Σ used to represent the ellipsoid's centroid vector and covariance matrix, respectively (see § 2.2). Furthermore, it also contains certain fields that are required to maintain the membership of an ellipsoid in the forest. To that end, a unique ID is required to identify the ellipsoid in the forest. Furthermore, a *parent* pointer, initialized to null, is utilized to support the structure of the trees in the forest. Moreover, an ellipsoid requires a *next* pointer and a *rank* value. As we explain in § 5.2, the next pointers facilitate efficient enumeration of ellipsoids in objects, and the rank values regulate the heights of the trees resulting from a mapCombining operation.

For a given cluster c, μ and Σ of the associated bounding ellipsoid are respectively the sample mean and sample covariance of the points in c, which can be calculated with $\mathcal{O}(1)$ time complexity. Similarly, the other fields of the bounding ellipsoid data structure can be initialized with $\mathcal{O}(1)$ time complexity. Furthermore, given two ellipsoids, we can determine if they geometrically overlap, using the method described in [2] with $\mathcal{O}(1)$ time complexity.

5.2. The algorithmic implementation of hierarchical ellipsoid forest

As noted in § 3.3.1, in a hierarchical forest, `mapCombining` operations are performed according to a combine hierarchy $\mathbb{H}$, which regulates the concurrent accesses and operations in the forest. Furthermore, the objects are represented by *enhanced* trees in a hierarchical forest. An enhanced tree facilitates iterating through the ellipsoids in the associated object with constant time per ellipsoid. The latter is achieved by making the ellipsoids in an enhanced tree form a *circular linked-list* via the next pointers (see § 5.1).

5.2.1. Compound operation

Operation `mapCombining`: Given maps Φ_i and Φ_j, for each pair of objects $\langle \mathbb{O}, \mathbb{O}' \rangle$, where $\mathbb{O} \in \Phi_i$ and $\mathbb{O}' \in \Phi_j$, `mapCombining` merges $\mathbb{O}$ and $\mathbb{O}'$ if they overlap. Afterwards, the objects in Φ_j get linked to Φ_i. Finally, potential duplicate objects in Φ_i are removed. To that end, an unset flag is associated with the root of every object in Φ_i. Then, every pointer in Φ_i's linked-list is iterated: the flag of the associated object's root gets marked if it is not already marked. Otherwise, the corresponding pointer gets removed from Φ_i's linked-list because another pointer in Φ_i already points to the same object. Algorithm 6 outlines the algorithmic implementation of `mapCombining`.

Enhancements (i) For improved amortized time complexity, we adopt the *path-compression* and *union-by-rank* heuristics [10]; the former flattens the trees, and the latter controls the growth of depth of the trees. To that end, the *rank* value (see § 5.1), initially zero, is assigned to each ellipsoid.

(ii) Note that `mapCombining`(Φ_i, Φ_j) checks all pairs of objects in Φ_i and Φ_j to merge the overlapping ones. Suppose object $\mathbb{O}$ in Φ_i and object $\mathbb{O}'$ in Φ_j do not overlap. To determine this, `mapCombining` has to check all pairs of ellipsoids in $\mathbb{O}$ and $\mathbb{O}'$. To avoid this worst-case behaviour, we propose a work-saving *test* that utilizes *delimiting boxes*.

Definition 4. An object's *delimiting box* is the smallest axis-aligned cuboid encapsulating the ellipsoids in the object.

The delimiting-box test: If the delimiting boxes of objects $\mathbb{O}$ and $\mathbb{O}'$ do not geometrically overlap, then $\mathbb{O}$ and $\mathbb{O}'$ do not overlap, hence effectively saving pairwise checks of the ellipsoids in $\mathbb{O}$ and $\mathbb{O}'$.

Algorithm 6 Operation `mapCombining` in a hierarchical ellipsoid forest.

```
1: procedure mapCombining(Φ_i, Φ_j)
2:   for 𝕆 ∈ Φ_i.list & 𝕆′ ∈ Φ_j.list do
3:     if overlap(𝕆, 𝕆′) then
4:       merge_H(𝕆, 𝕆′)
5:   Φ_i.list.pushAll(Φ_j.list)
6:   unmark all objects in Φ.list
7:   for 𝕆 ∈ Φ.list do
8:     if findRoot_H(𝕆).marked then
9:       Φ.list.remove(𝕆)
10:    else
11:      findRoot_H(𝕆).marked:= 1
12:  return Φ_i
```

5.2.2. Auxiliary operations

In the hierarchical forest, any ellipsoid in an object can be used to represent the object because all the ellipsoids in the object can be accessed via the circular linked-list. Furthermore, the representative (i.e., the root) of the object can be accessed by following the parent pointers. With this note in mind, we introduce the basic operations.

Algorithm 7 Auxiliary operations in a hierarchical ellipsoid forest.

```
1: procedure overlap(𝕆, 𝕆′)
2:   if ¬overlap(𝕆.dBox, 𝕆′.dBox)
       then
3:     return false
4:   for e ∈ 𝕆 & e′ ∈ 𝕆′ do
5:     if e and e′ overlap then
6:       return true
7:   return false
8: procedure merge_H(𝕆, 𝕆′)
9:   e :=findRoot_H(𝕆)
10:  e′ :=findRoot_H(𝕆′)
11:  link_H(e, e′)
12:  swap(e.next, e′.next)
13:  findRoot_H(e).dBox :=
       union(e.dBox, e′.dBox)
14: procedure findRoot_H(e)
15:   if e.parent == ∅ then
16:     return e
17:   else
18:     e.parent:=find-
          Root_H(e.parent)
19:     return e.parent
20: procedure link_H(e, e′)
21:   if e.rank > e′.rank then
22:     e′.parent := e
23:   else
24:     e.parent := e′
25:     if e.rank == e′.rank then
26:       e′.rank := e′.rank + 1
```

Operation `overlap`: Given objects $\mathbb{O}_1$ and $\mathbb{O}_2$, this operation determines if $\mathbb{O}_1$ and $\mathbb{O}_2$ overlap. Algorithm 7 shows the algorithmic implementation of `overlap` with the delimiting box test. Note that Algorithm 7 l.4 utilizes the circular linked-lists of the enhanced trees to iterate over each ellipsoid in constant time.

Operation $\texttt{merge}_H$: Given objects $\mathbb{O}_1$ and $\mathbb{O}_2$, this operation unifies the enhanced trees corresponding to $\mathbb{O}_1$ and $\mathbb{O}_2$ into a single enhanced tree in the hierarchical forest. First of all, the roots/representatives of the two objects are retrieved using the `findRoot` operation. Second, the aforementioned roots are linked via a call to the $\texttt{link}_H$ operation. Third, to make the ellipsoids in the new enhanced tree form a circular linked-list, the circular linked-lists associated with $\mathbb{O}_1$ and $\mathbb{O}_2$ are unified by *swapping* the next pointers of the roots. Finally, the delimiting box of the new object is adjusted so that it encompasses the delimiting boxes of $\mathbb{O}_1$ and $\mathbb{O}_2$. The operation is conducted in-place, avoiding unnecessary data copying or moving

Operation $\texttt{findRoot}_H$: Given an ellipsoid `e`, $\texttt{findRoot}_H$ traverses the chain of parent pointers until it reaches the root of the object in which `e` is a member. A recursive implementation of the $\texttt{findRoot}_H$ operation with the path-compression heuristic is provided in Algorithm 7, where, as the recursion unwinds on a path to a root, the parent pointers start pointing to the root. Furthermore, given an ellipsoid in an object $\mathbb{O}$, $\texttt{findRoot}_H$ returns the root ellipsoid in $\mathbb{O}$.

Operation $\texttt{link}_H$: This operation links two ellipsoids `e` and `e′`, as the roots of two distinct objects, using the union-by-rank heuristic. To that end, $\texttt{link}_H$ sets the parent pointer of the one with the lower rank to the other one (i.e., attaching the shorter tree to the taller tree). If `e` and `e′` have the same rank, then one of them is chosen to be the new root, and its rank gets incremented. Algorithm 7 outlines the $\texttt{link}_H$ operation.

5.3. The algorithmic implementation of flat ellipsoid forest

As we explained in Definition 2, the flat forest extends the disjoint set data structure by the `ellipsoidLinking` operation. The flat forest allows concurrent wait-free execution of `ellipsoidLinking` operations in any order by utilizing fine-grained synchronization primitives as proposed in [23].

5.3.1. Compound operation

Operation `ellipsoidLinking`: Given split-summaries φ_i and φ_j, this operation checks whether each ellipsoid pair $\langle e, e' \rangle$, where `e` belongs to φ_i and `e′` belongs to φ_j, overlaps. In that case, it merges the objects associated with `e` and `e′`. Algorithm 8 outlines the algorithmic implementation of `ellipsoidLinking`.

Table 2
Table of Notation.

N	≜	number of points in the input dataset	$\|\mathbb{O}\|$	≜	number of ellipsoids in object $\mathbb{O}$
K	≜	number of threads	$\|\|\Phi\|\|$	≜	sum of number of ellipsoids in objects in Φ
S	≜	number of data splits			
γ	≜	number of local clusters in a data split	$\alpha(.)$	≜	inverse Ackermann function

Algorithm 8 Operation `ellipsoidLinking` in a flat ellipsoid forest.

```
1: procedure ellipsoidLinking(φi,φj)
2:   for e ∈ φi.list & e′ ∈ φj.list do
3:     if e and e′ overlap then
4:       merge_F(e, e′)
```

Algorithm 9 Auxiliary operations in a flat ellipsoid forest. The last executed step marked by an asterisk gives the linearization point. Adapted from [23].

```
1: procedure findRoot_F(e)
2:   while e.parent ≠ ∅* do
3:     e:=e.parent
4:   return e
5: procedure merge_F(e, e′)
6:   while true do
7:     e := findRoot_F(e)
8:     e′ := findRoot_F(e′)
9:     if e.ID < e′.ID then
10:      if CAS(e.parent, ∅, e′)* then
11:        return
12:    else if (e.ID == e′.ID)* then
13:      return
14:    else if e.ID > e′.ID then
15:      if CAS(e′.parent, ∅, e)* then
16:        return
```

5.3.2. Auxiliary operations

Operation `findRoot`$_F$: Given an ellipsoid `e`, this operation follows the parent pointers from `e` until it reaches the root of the object in which `e` belongs.

Operation `merge`$_F$: Given two ellipsoids `e` and `e′`, this operation merges the trees in the forest that are associated with `e` and `e′`. First of all, utilizing the `findRoot`$_F$ operation, the representatives of the objects associated with `e` and `e′` are found. Afterwards, the parent pointer of the object representative with the lower `ID` value gets linked to the object representative with the higher `ID` value. As there are concurrent accesses to the elements in the forest, there might be other threads that link the aforementioned parent pointer to another ellipsoid. Therefore, the implementation shown in Algorithm 9 utilizes `CAS` to atomically update the parent pointers. If the `CAS` operation fails when changing a parent pointer, it means that the parent pointer was already changed by some other thread (executing an overlapping `ellipsoidLinking` operation involving some ellipsoid(s) belonging to the same object(s)) in the meantime. Therefore, the roots of the associated objects are recalculated, and then the same mechanism tries to link them. The aforementioned steps continue in the retry loop (shown in Algorithm 9 l.6-l.16) until the roots of the associated objects get linked (by any of the threads). The operation is conducted in-place, avoiding unnecessary data copying or moving.

5.4. Discussion on system aspects

The proposed algorithmic descriptions of PARMA-CC algorithms incorporate a simple *scheduler* [8] for parallel execution of tasks. Such a choice allows us to uncover the algorithm properties of the design space and as well as the behaviour of the ellipsoid forests. In general, parallel execution of tasks in PARMA-CC algorithms can be scheduled using any off-the-shelf parallelization library, such as OpenMP [22], TBB [45], and Cilk [7]. Using such parallelization libraries is orthogonal to the scope of this work as it introduces new aspects and trade-offs to the study. Nevertheless, using such libraries can facilitate scheduling and executing finer parallel tasks. For instance, each invocation of `ellipsoidLinking` operation be decomposed into several parallelization tasks. As shown in Algorithm 8, there is no dependency between the iterations of the for loop in `ellipsoidLinking`; therefore, each iteration of the aforementioned for loop can be performed in parallel.

6. Analysis

We provide an analytical study of PARMA-CC algorithms.
Notation: Let γ be an upper bound on the number of locally detected clusters in a split of data. For an object $\mathbb{O}$, let $|\mathbb{O}|$, i.e., size of $\mathbb{O}$, be the total number of ellipsoids in $\mathbb{O}$. Considering a hierarchical PARMA-CC algorithm, for a map Φ, let $\|\Phi\|$ the number of all the ellipsoids in Φ. Table 2 summarizes the notations in this section.

6.1. Ellipsoid forest analysis

6.1.1. Hierarchical ellipsoid forest

Lemma 1. *[Adapted from Lemma 21.13 in [10]] The worst-case and amortized time complexity of each* `findRoot`$_H$ *operation is respectively* $\mathcal{O}(\log(\gamma S))$ *and* $\mathcal{O}(\alpha(\gamma S))$*, where* $\alpha(.)$ *is the inverse Ackermann function.*

Note that $\alpha(.)$ is a very slowly growing function where $\alpha(x) < 5$ for $x < 10^{80}$.

Lemma 2. *The worst-case time complexity of each* `overlap` *operation on objects* $\mathbb{O}_1$ *and* $\mathbb{O}_2$ *is* $\mathcal{O}(|\mathbb{O}_1||\mathbb{O}_2|)$.

Lemma 3. *The worst-case and amortized time complexity of each* `merge`$_H$ *operation is respectively* $\mathcal{O}(\log(\gamma S))$ *and* $\mathcal{O}(\alpha(\gamma S))$.

Proof. A `merge`$_H$ calls (i) three `findRoot`$_H$ operations, (ii) one `link`$_H$ operation, and (iii) swapping the values of two pointers. According to Lemma 1, the worst-case and amortized time complexity of (i) is respectively $\mathcal{O}(\log(\gamma S))$ and $\mathcal{O}(\alpha(\gamma S))$. (ii) and (iii) are performed with $\mathcal{O}(1)$ time complexity.

Lemma 4. *The worst-case and amortized time complexities of each* `mapCombining` *operation on* Φ_i *and* Φ_j *are bounded from above by* $\mathcal{O}\left(\log(\gamma S).||\Phi_i||.||\Phi_j||\right)$ *and* $\mathcal{O}\left(\alpha(\gamma S).||\Phi_i||.||\Phi_j||\right)$*, respectively.*

The above follows from deriving an upper bound on the summation of time complexities of operations `overlap` and `merge`$_H$ as performed by the `mapCombining` operation. Note that the bound for the amortized complexity is loose for two reasons: (i) As soon as $\mathbb{O}$ and $\mathbb{O}'$ are found to have overlapping ellipsoids, `overlap` returns true without further investigation of the remaining cases, see Algorithm 7 l.5-6. (ii) When objects $\mathbb{O}$ and $\mathbb{O}'$ do not overlap, with high probability, the delimiting boxes of $\mathbb{O}$ and $\mathbb{O}'$ do not overlap either; therefore, saving the comparisons of ellipsoids in $\mathbb{O}$ and $\mathbb{O}'$.

6.1.2. Flat ellipsoid forest

Lemma 5. *[adapted from Theorem 1 in [23]] Any concurrent execution order of* $\texttt{findRoot}_F$ *and* $\texttt{merge}_F$ *is linearizable and wait-free.*

Lemma 6. *[adapted from Theorem 2 in [23]] The probability that each* $\texttt{findRoot}_F$ *and each* $\texttt{merge}_F$ *perform* $\mathcal{O}(\log(\gamma \texttt{S}))$ *steps is at least* $1 - \frac{1}{\gamma \texttt{S}}$.

Lemma 7. *The expected asymptotic time complexity of each* $\texttt{findRoot}_F$ *and each* $\texttt{merge}_F$ *is* $\mathcal{O}(\log(\gamma \texttt{S}))$.

Proof. Based on Lemma 6, the probability that each $\texttt{findRoot}_\texttt{F}$ and each $\texttt{merge}_\texttt{F}$ perform $\Theta(\gamma \texttt{S})$ steps (the maximum possible height of a tree in the ellipsoid forest) is at most $\frac{1}{\gamma \texttt{S}}$. Therefore, the expected time complexity of each $\texttt{findRoot}_\texttt{F}$ and $\texttt{merge}_\texttt{F}$ is less than or equal to $(1 - \frac{1}{\gamma \texttt{S}})\mathcal{O}(\log(\gamma \texttt{S})) + \frac{1}{\gamma \texttt{S}}\Theta(\gamma \texttt{S})$, yielding the bound $\mathcal{O}(\log(\gamma \texttt{S}))$.

Lemma 8. *The expected asymptotic time complexity of each* `ellipsoidLinking` *operation is* $\mathcal{O}\left(\gamma^2 \log(\gamma \texttt{S})\right)$.

Proof. Consider `ellipsoidLinking` on two given maps. Due to the linearity of expectation, the expected time complexity of `ellipsoidLinking` is the sum of expected time complexities of the $\texttt{merge}_\texttt{F}$ operations that it performs. Consider two given maps. The maximum number of times that `ellipsoidLinking` can perform the $\texttt{merge}_\texttt{F}$ on the two maps is at most $\mathcal{O}(\gamma^2)$ times (the number of pairs of ellipsoids in the two maps), where each $\texttt{merge}_\texttt{F}$ has expected time complexity of $\mathcal{O}(\log(\gamma \texttt{S}))$ according to Lemma 7.

6.2. Safety and completeness properties of PARMA-CC algorithms

Lemma 9. *Operations and* `mapCombining` *and* `ellipsoidLinking` *satisfy the commutative and associative properties.*

The above follows from the descriptions and the algorithmic implementations introduced in § 5.2 and § 5.3.

Lemma 10. *For any concurrent execution of a PARMA-CC algorithm, there exists a sequential execution that produces an equivalent result.*

Proof. We argue how to build an equivalent sequential execution corresponding to a concurrent execution of a PARMA-CC algorithm. Similar to the concurrent execution, the equivalent sequential algorithm splits the input dataset into $\texttt{S}$ splits and operates in three matching phases, except for the synchronization details, which are not needed in the equivalent sequential execution. Regarding phase I, the signalling mechanism in $\texttt{PARMA}_\texttt{H}$ (shown in Algorithm 2 l.9), updating the status values in $\texttt{PARMA}_\texttt{F}$ (shown in Algorithm 3 l.7-8), insertions in $\texttt{Q}$ in $\texttt{FLEXI-PARMA}_\texttt{H}$ (shown in Algorithm 4 l.11), and updating the status values in $\texttt{FLEXI-PARMA}_\texttt{F}$ (shown in Algorithm 5 l.9-10) are not needed in the equivalent sequential algorithm. Note that besides the aforementioned synchronization details, the rest of the operations in phase I of a PARMA-CC execution are performed in a data parallel fashion. Therefore, in phase I, the equivalent sequential algorithm can perform the local clustering tasks and create the split-summaries in any arbitrary order. The same argument also holds regarding phase III of the equivalent sequential algorithm. We explain how to construct the rest of the equivalent sequential execution (i.e., phase II) for the hierarchical and flat PARMA-CC algorithms in the following:

Hierarchical: A hierarchical PARMA-CC algorithm performs `mapCombining` (see Algorithm 6) operations in the hierarchical forest according to hierarchy $\mathbb{H}$, where $\mathbb{H}$ can either be predetermined as in $\texttt{PARMA}_\texttt{H}$ or be dynamically determined as in $\texttt{FLEXI-PARMA}_\texttt{H}$. In either case, `mapCombining` operations are performed with respect to the following rules: (P1) `mapCombining` operations corresponding to disjoint subtrees in $\mathbb{H}$ can be performed in parallel. (P2) `mapCombining` operations which have an ancestor-descendant relation in $\mathbb{H}$ never modify the same sets in the forest simultaneously. (P3) Each ellipsoid belongs to only one object both before and after a `mapCombining` operation. (P4) All pairs of objects that have overlapping ellipsoids are merged in the final map. Therefore, in phase II, the equivalent sequential algorithm can sequentially perform `mapCombining` operations following any arbitrary hierarchy $\mathbb{H}$ and get the same set of objects because operation `mapCombining` satisfies the commutative and associative properties (see Lemma 9).

Flat: The threads in a flat PARMA-CC algorithm perform `ellipsoidLinking` (see Algorithm 8) operations on all pairs of split-summaries (i.e., elements of $\texttt{V}$ as defined in Definition 3). We show in the following that any arbitrary (due to concurrency) inter-leaving of `ellipsoidLinking` operations, results in the same set of objects.

1. Each `ellipsoidLinking` operation in $\texttt{V}$ is performed exactly once. The latter holds because each thread tries to atomically book available an `ellipsoidLinking` operation via performing `CAS` on the corresponding status value (see Definition 3), as long as there are available `ellipsoidLinking` operations left.
2. As the threads perform `ellipsoidLinking` operations, concurrent executions of operation $\texttt{merge}_\texttt{F}$ might be performed.
3. As any concurrent execution of $\texttt{merge}_\texttt{F}$ is linearizable (see Lemma 5), and operation `ellipsoidLinking` satisfies the commutative and associative properties (see Lemma 9), the same set of objects gets formed in the ellipsoid forest regardless of linearization of the $\texttt{merge}_\texttt{F}$ operations.

Based on the sequence of arguments in (1), (2), and (3), any concurrent execution of `ellipsoidLinking` operations on all pairs of split-summaries results in the same set of objects. Therefore, in phase II, the equivalent sequential algorithm can sequentially perform `ellipsoidLinking` in any arbitrary order and get the same set of objects.

Definition 5 *(The Completeness Property).* An ellipsoid forest satisfies the completeness property when the following condition holds for each pair of ellipsoids $\langle \texttt{e}_\texttt{i}, \texttt{e}_\texttt{j} \rangle$ in the forest: The pair $\langle \texttt{e}_\texttt{i}, \texttt{e}_\texttt{j} \rangle$ is directly or indirectly overlapping (see Definition 1) if and only if there exists an object $\mathbb{O}$ such that $\texttt{e}_\texttt{i} \in \mathbb{O}$ and $\texttt{e}_\texttt{j} \in \mathbb{O}$.

Lemma 11 *(Completeness in* $\texttt{PARMA}_\texttt{H}$*). By the end of phase II in* $\texttt{PARMA}_\texttt{H}$*, the completeness property holds in the associated hierarchical ellipsoid forest.*

Proof. We first prove the statement in the following direction: If the pair $\langle \texttt{e}_\texttt{i}, \texttt{e}_\texttt{j} \rangle$ is directly or indirectly overlapping, then, by the end of phase II, there exists an object $\mathbb{O}$ such that $\texttt{e}_\texttt{i} \in \mathbb{O}$ and $\texttt{e}_\texttt{j} \in \mathbb{O}$. To that end, consider phase I, when $\texttt{e}_\texttt{i}$ is a member of $\mathbb{O}_{i'}$ in map $\Phi_\texttt{i}$, and $\texttt{e}_\texttt{j}$ is a member of $\mathbb{O}_{j'}$ in map $\Phi_\texttt{j}$. If the pair $\langle \texttt{e}_\texttt{i}, \texttt{e}_\texttt{j} \rangle$ is directly overlapping, then $\mathbb{O}_{i'}$ and $\mathbb{O}_{j'}$ are merged when `mapCombining` operation is performed on the maps containing $\texttt{e}_\texttt{i}$ and $\texttt{e}_\texttt{j}$ (such a `mapCombining` operation is guaranteed to exist as $\mathbb{H}$ is a spanning tree, see § 3.3.1). On the other hand, suppose the pair $\langle \texttt{e}_\texttt{i}, \texttt{e}_\texttt{j} \rangle$ is indirectly overlapping via ellipsoid $\texttt{e}_\texttt{k}$

(i.e., ellipsoids in each of the pairs $\langle e_i, e_k\rangle$ and $\langle e_j, e_k\rangle$ are directly overlapping), where e_k belongs to object $\mathbb{O}_{k'}$, at the end of phase I, in Φ_k. After mapCombining operations are performed on Φ_i, Φ_j, and Φ_k in the order specified by $\mathbb{H}$, there will be an object containing e_i, e_j, and e_k. The latter holds regardless of the hierarchy specified by $\mathbb{H}$ because the mapCombining operation satisfies the commutative property (see Lemma 9). This argument can be extended inductively to cover all the cases in which e_i and e_j are indirectly overlapping.

Now we prove the statement in the opposite direction: If there exists an object $\mathbb{O}$ such that $e_i \in \mathbb{O}$ and $e_j \in \mathbb{O}$, then the pair $\langle e_i, e_j\rangle$ is directly or indirectly overlapping. Towards a contradiction, suppose the pair $\langle e_i, e_j\rangle$ is neither directly nor indirectly overlapping, but $e_i \in \mathbb{O}$ and $e_j \in \mathbb{O}$. The latter implies that, at some point in phase II, the mapCombining operation combined non-overlapping objects, a contradiction.

Lemma 12 *(Completeness in $\texttt{PARMA}_F$). By the end of phase II in $\texttt{PARMA}_F$, the completeness property holds in the associated flat ellipsoid forest.*

Proof. We first prove the statement in the following direction: If the pair $\langle e_i, e_j\rangle$ is directly or indirectly overlapping, then there exists an object $\mathbb{O}$ such that $e_i \in \mathbb{O}$ and $e_j \in \mathbb{O}$. To that end suppose $e_i \in \varphi_i$ and $e_j \in \varphi_j$. If the pair $\langle e_i, e_j\rangle$ is directly overlapping, then, through the call to ellipsoidLinking(φ_i, φ_j), e_i and e_j will become members of the same object. On the other hand, if the pair $\langle e_i, e_j\rangle$ is indirectly overlapping with just an ellipsoid $e_k \in \varphi_k$ in between (i.e., ellipsoids in each of the pairs $\langle e_i, e_k\rangle$ and $\langle e_j, e_k\rangle$ are directly overlapping), then there will be an object containing e_i and e_j (as well as e_k) after ellipsoidLinking(φ_i, φ_k) and ellipsoidLinking(φ_j, φ_k) are completed. This argument can be inductively extended to cover all the cases in which e_i and e_j are indirectly overlapping.

The proof in the opposite direction is made with contradiction, similar to the one provided in the proof of Lemma 11.

Lemma 13 *(Completeness in Flexi PARMA-CC Algorithms). At the end of phase II, the ellipsoid forest in a flexi PARMA-CC satisfies the completeness property.*

Proof. For a given hierarchical/flat flexi PARMA-CC algorithm operating on S splits, consider a hierarchical/flat basic PARMA-CC algorithm that operates with K=S threads. The two algorithms produce equivalent ellipsoid forests because both mapCombining and ellipsoidLinking satisfy the commutative property. Therefore, the ellipsoid forest at the end of phase II of the flexi PARMA-CC algorithm satisfies the completeness property similar to the basic PARMA-CC algorithm (based on Lemma 11 and Lemma 12).

Corollary 1. *With fixed minPts, ϵ, and S, PARMA-CC algorithms yield the same clustering for the same input dataset.*

6.3. Completion time behaviour of PARMA-CC algorithms

Here we analyze the completion time behaviour of the algorithms in the PARMA-CC family.

Assumptions:

- As D can contain several hundreds of thousands of points, but the number of splits is limited to a few hundreds, we assume $N \gg S \geq K$. Furthermore, we assume $N \gg \gamma$ because a local cluster can typically contain a large number of points.
- The local clustering algorithm (for instance PCL-EC or DBSCAN) uses a kd-tree to perform ϵ-neighbourhood queries.
- The total number of ellipsoids in an ellipsoid forest (γS) is smaller than 10^{80} for all the possible use-cases. Therefore, all occurrences of the inverse Ackermann function are substituted with $\mathcal{O}(1)$.

Lemma 14. *The following statements hold regarding the completion time of different phases of a PARMA-CC algorithm:*

- *The expected completion time of phase Iis $\mathcal{O}(\frac{N}{K}\log(\frac{N}{S}))$.*
- *The expected completion time of phase II of a hierarchical PARMA-CC algorithm is $\mathcal{O}(\gamma^2 S^2)$.*
- *The expected completion time of phase II of a flat PARMA-CC algorithm is $\mathcal{O}\left(\frac{\gamma^2 S^2}{K}\log(\gamma S)\right)$.*
- *The expected completion time of phase III in a hierarchical PARMA-CC algorithm is $\mathcal{O}(\frac{N}{K})$.*
- *The expected completion time of phase III in a flat PARMA-CC algorithm is $\mathcal{O}\left(\frac{N}{K}\log(\gamma S)\right)$.*

Proof. We prove each statement in the following:

- We prove the statement for basic and flat PARMA-CC algorithms:
  - In case of a basic PARMA-CC algorithm ($S = K$): As the workload is distributed evenly among the K threads, the expected completion time of a data split clustering is $\mathcal{O}(\frac{N}{K}\log(\frac{N}{K}))$.
  - In case of a flexi PARMA-CC algorithm ($S > K$): There are S local clustering tasks to be shared by K threads, and as each split contains N/S points, the expected completion time of a data split clustering is $\mathcal{O}(\frac{N}{S}\log(\frac{N}{S}))$. Therefore, the expected completion time of K threads concurrently performing local clustering is $\mathcal{O}\left(\frac{S}{K}\frac{N}{S}\log(\frac{N}{S})\right) = \mathcal{O}(\frac{N}{K}\log(\frac{N}{S}))$.

  Other computational steps in phase I of a PARMA-CC (i.e., fitting bounding ellipsoids, applying synchronization primitives, pushing elements into a queue) are asymptotically dominated by $\mathcal{O}(\frac{N}{K}\log(\frac{N}{S}))$.
- Summing up the amortized time complexities of all mapCombining operations (see Lemma 4), the expected total time complexity of all mapCombining operations is bounded from above by $\mathcal{O}(\gamma^2 S^2)$, which is a loose bound based on the proof of Lemma 4.
- In phase II of a flat PARMA-CC algorithm, there are $\mathcal{O}(S^2)$ ellipsoidLinking operations which are shared by K threads running in parallel. Considering the fine granularity of the ellipsoidLinking operations, each thread performs $\mathcal{O}(S^2/K)$ ellipsoidLinking operations. Applying the linearity of expectation over the expected time complexity of each ellipsoidLinking (given in Lemma 8) yields the result.
- We prove the statement for basic and flat PARMA-CC algorithms:
  - In case of a basic PARMA-CC algorithm ($S = K$): As the workload is distributed evenly among the K threads, each thread relabels $\mathcal{O}(\frac{N}{K})$ points.
  - In case of a flexi PARMA-CC algorithm ($S > K$): There are S local relabeling tasks to be shared by K threads, and as each split contains N/S points, each thread relabels $\mathcal{O}\left(\frac{S}{K}\frac{N}{S}\right) = \mathcal{O}(\frac{N}{K})$ points.

  The amortized time complexity of relabeling each point is $\mathcal{O}(1)$ because finding the root of the associated tree, via performing

Table 3
Algorithms in the PARMA-CC family (SP stands for synchronization primitives).

Algorithm	Ellipsoid Forest/Combine Order	Basic/Flexi	Synchronization			Preferred Data Properties
			Phase I	Phase II	Phase III	
$\texttt{PARMA}_\texttt{H}$	Hierarchical/predetermined	Basic	-	SP	-	arbitrarily ordered
$\texttt{PARMA}_\texttt{F}$	Flat/dynamic	Basic	-	SP	-	spatio-temporal (e.g. LIDAR)
$\texttt{FLEXI-PARMA}_\texttt{F}$	Flat/dynamic	Flexi	SP	SP	SP	spatio-temporal (e.g. LIDAR)
$\texttt{FLEXI-PARMA}_\texttt{H}$	Hierarchical/dynamic	Flexi	Ready Queue + SP	SP	SP	arbitrarily ordered

$\texttt{findRoot}_\texttt{H}$ (Lemma 1), and then retrieving the root's ID are the only required steps for each point. The expected completion time can be driven by taking a summation over the amortized time complexities of relabeling each point.

- This statement is proven similar to the previous one. Due to the linearity of expectation, summing the expected completion times of $\mathcal{O}(\frac{\texttt{N}}{\texttt{K}})$ $\texttt{findRoot}_\texttt{F}$ operations (given in Lemma 7) yields the result.

Observation 1. *Lemma 14 indicates the following trade-off between the expected completion time of phase I and the expected completion time of phase II: the dominating factor in the completion time of a PARMA-CC algorithm, i.e., the local clustering in phase I, can be reduced by increasing* K *and/or* S*. However, too large values for* K *and/or* S *increase the expected completion time of phase II.*

Theorem 1. *The expected completion time of a PARMA-CC algorithm under the given assumptions is* $\mathcal{O}(\frac{\texttt{N}}{\texttt{K}}\log(\frac{\texttt{N}}{\texttt{S}}))$.

Proof. The theorem follows from taking the dominating asymptotic term in Lemma 14.

6.4. On shared memory accesses and contention

As discussed in § 5.1, the operations on the data structure are in-place, avoiding unnecessary copies and moves of data. Regarding contention on the shared memory in different PARMA-CC algorithms, note that there is none in $\texttt{PARMA}_\texttt{H}$ because the computations that each thread performs follow a predetermined partial order that ensures that concurrent operations touch disjoint data only. The number of occasions in which shared memory contention can take place in $\texttt{FLEXI-PARMA}_\texttt{H}$ is proportionate to S, i.e., the number of shared tasks. On the other hand, the number of shared tasks in flat PARMA-CC algorithms is proportionate to $\texttt{S}^2$, determined by the number of `ellipsoidLinking` operations. Note that this shared memory contention discussion is complementary to the expected completion time analysis in Lemma 14, which, among other factors, takes into account the expected number of retries that have to be performed because of memory contention, where necessary.

Note when X threads concurrently perform a `CAS` operation on a memory location, only one of them succeeds and $\texttt{X}-1$ threads fail. Therefore, to measure memory contention, we consider the average ratio of failed CAS operations to the total number of invoked CAS operations. Exact measurements for the ratio of failed CAS operations to the total number of invoked CAS operation are data-dependent and execution-dependent. In the worst-case, the aforementioned ratio can be as large as $1-\frac{1}{\texttt{K}}$, indicating one successful `CAS` against $\texttt{K}-1$ unsuccessful `CAS` for all the invocations. In the algorithmic implementation of PARMA-CC, some invocations of `CAS` operations can be avoided; e.g., a thread does not need to invoke a `CAS` to book a local clustering task which is already booked or completed (see Algorithm 4 l.7 and Algorithm 5 l.6). The same also applies for booking local clustering tasks and `ellipsoidLinking` operations. We empirically measure the aforementioned ratio in § 8.5.

7. Discussion on the utilization and the building components of the algorithms

7.1. On which PARMA-CC algorithm to choose

Let *inter-split overlap* refer to the amount of overlap between local clusters in different splits. With high inter-split overlap, utilizing the hierarchical forest can result in a higher scaling-factor compared to utilizing the flat forest. First of all, as the inter-split overlap increases, the average number of ellipsoids in different objects increases. Consecutively, the computational savings of the delimiting-box test increase as a result of skipping the comparison of ellipsoids in non-overlapping objects. On the other hand, the amount of concurrent updates on overlapping elements in a flat forest is directly proportional to the inter-split overlap. Therefore, threads performing `ellipsoidLinking` in a flat forest have to retry (see Algorithm 9 l.6) for more number of times for successful linking as the inter-split overlap increases.

Observation 2. *With the splitting mechanism outlined in § 3.2 and input data having a* spatio-temporal *locality (e.g., an angularly sorted LIDAR point cloud), the inter-split overlap is low. Therefore, we expect the flat PARMA-CC algorithms to scale better under the aforementioned conditions. On the contrary, we expect the hierarchical PARMA-CC algorithms to scale better on arbitrarily ordered datasets that exhibit high inter-split overlap. Table 3 summarises the PARMA-CC algorithms.*

7.2. Use cases implying extensions

PARMA-CC's summaries can be used to efficiently answer a range of queries. We demonstrate the latter by studying two common queries, for which we study and compare how PARMA-CC and a classical approach can be used.

Predicting the clustering label of a new point q based on the existing clusters: The latter can be useful in evolving sets. A classical approach might decide about q's clustering label by considering the clustering labels of q's nearest neighbours in D. Using a kd-tree, nearest neighbour queries have expected and worst-case time complexities of $\mathcal{O}(\log \texttt{N})$ and $\mathcal{O}(\texttt{N}^2)$, respectively. On the other hand, the approach leveraging the summaries of a PARMA-CC algorithm can assign q the unique ID of $\mathbb{O}_\texttt{q}$'s root, where $\mathbb{O}_\texttt{q}$ is the object in which q geometrically falls. The latter is shown as operation `predictLabel` in Algorithm 10 l.1-6. As the total number of ellipsoids is $\gamma\texttt{S}$, `predictLabel`'s worst-case time complexity is $\mathcal{O}(\gamma\texttt{S})$. Note that, in general, $\gamma\texttt{S}$ is much smaller than N.

Approximating the distance of a given point q to the nearest point in cluster c: With time complexity $\mathcal{O}(|\texttt{c}|)$, a classical approach calculates the distance of each point in c to q and returns the smallest one. On the other hand, the approach levering PARMA-CC's summaries computes the distance of q to each ellipsoid in $\mathbb{O}_\texttt{c}$, where $\mathbb{O}_\texttt{c}$ is a PARMA-CC object corresponding to cluster c. The latter is shown as operation `distanceToObject` in Algorithm 10. The distance between a point and an ellipsoid is determined in $\mathcal{O}(1)$ using the method in [35]. Therefore, `distanceToObject`'s time complexity is $\mathcal{O}(|\mathbb{O}|)$.

Algorithm 10 Answering queries using PARMA-CC's summaries.

```
1: procedure predictLabel(q)
2:     for i ∈ {1, ··· , S} do
3:         for e ∈ φ_i.list do
4:             if q falls within e then
5:                 return findRoot(e).ID
6:     return noise
7: procedure distanceToObject(𝕆, q)
8:     return min_{e∈𝕆} {distance between q and e}
```

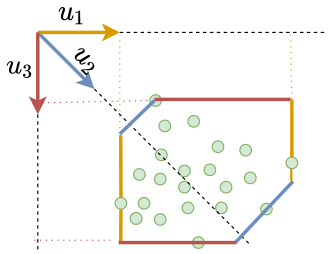


Fig. 4. Polyhedron fitting.

7.3. On volumetric summarization methods

Besides the bounding ellipsoid summarization method, PARMA-CC algorithms can utilize other geometric summarization methods such as axis-aligned bounding boxes (AABBs) [41] or oriented bounding boxes (OBBs) [16]. More generally, we consider *bounding polyhedrons*. Fig. 4, shows an example cluster of points (represented by green circles) and a corresponding bounding polyhedron. We characterize a bounding polyhedron by a set of normal vectors u_i for $i \in \{1, \cdots, F\}$, where each u_i is a normal vector to two parallel faces in the bounding polyhedron.

Fitting a bounding polyhedron around a local cluster c: The minimum and maximum values of the orthogonal projections of points in c onto vector u_i determine respectively the *left* and *right* faces associated with normal vector u_i. Note that the orthogonal projection of a point onto a vector is simply calculated by their dot product. The example in Fig. 4 utilizes three normal vectors u_1, u_2, and u_3. Note that the left and right faces associated with each normal vector are shown in the same colour as the normal vector.

Determining the normal vectors: The normal vectors u_is can either be chosen randomly or in a systematic way to uniformly sample the unit sphere in the space. The number of vectors, F, determines the granularity of the volumetric approximation, i.e., increasing F increases the approximation accuracy but increases the cost of computing the bounding polyhedron. However, with fixed F, the cost of fitting a bounding polyhedron is constant per point in c.

Determining whether two bounding polyhedra geomterically overlap: Two polyhedra P_1 and P_2 are geometrically overlapping if and only if, for all u_is, the intervals containing the left and right faces in P_1 and P_2 overlap.

8. Evaluation

We here empirically evaluate PARMA-CC algorithms. The parameters of the study are the following: the number of threads (K), the number of splits (S), the size of the input data (N), the number of objects in the input data, the degree of inter-split overlap in the input data, and the local clustering algorithm.

8.1. Experiment setup

We study the completion time of PARMA-CC algorithms in accordance with the expected completion time analysis given in Theorem 1. To understand the parallelization utilization behaviour of PARMA-CC algorithms, we evaluate their scaling-factors, i.e., the ratio of the completion time of the exact sequential baseline to the completion time of PARMA-CC algorithms, as a function of K and S. We also empirically examine the expectations raised in Observation 2 regarding the behaviour of the algorithms on datasets with different degrees of inter-split overlap. Moreover, we study the ratio of the local clustering time to the completion time of the algorithms in accordance with the analytical results in Lemma 14 to gain insight on how the different phases contribute to the total completion time. Furthermore, we evaluate the clustering accuracy of PARMA-CC algorithms against the exact baseline using *rand index*.[4] Finally, we complement the shared memory access and contention analysis in § 6.4 by empirically measuring the average ratio of failed CAS operations to the total number of invoked CAS operations.

We provide the evaluation results corresponding to PARMA-CC algorithms that utilize PCL-EC (see § 2) as the local clustering algorithm. Moreover, in order to evaluate PARMA-CC algorithms utilizing a density-based local clustering algorithm, we study the scaling-factor and the accuracy of basic PARMA-CC algorithms utilizing DBSCAN as the local clustering algorithm. Accordingly, we compare the scaling-factor of the aforementioned algorithms with that of PDSDBSCAN (see § 2), which for the latter is the same as its speedup. By default, the presented results and discussions refer to PARMA-CC algorithms that utilize PCL-EC as the local clustering algorithm, unless otherwise stated.

Evaluation data: We use both LIDAR and GPS datasets. Regarding LIDAR data, we study a random subset of the point clouds in the KITTI dataset [13], collected by a Velodyne laser scanner in urban driving. We also study a random subset of the Ford Multi-AV Seasonal dataset [1] which is collected by a fleet of vehicles in a variety of conditions. Regarding GPS data, we choose a random subset of points in the Mopsi route dataset [30], which contains GPS readings (in terms of latitude and longitude) gathered by various users doing a wide range of activities (e.g., walking, cycling, skiing, taking a boat) mostly in Finland. We also study a random subset of the GeoLife GPS Trajectories dataset [48,46,47], containing densely recorded GPS readings by several users mostly in Beijing city. Furthermore, we study randomly shuffled versions of the GeoLife and Mopsi datasets, exhibiting high inter-split overlap. Table 4 gives an overview of each bench-marked dataset along with its inter-split overlap characterization.

Preprocessing: By imposing a simple threshold, we filter the ground (floor) points in the point clouds (otherwise scene objects are connected via the ground). Each filtered point cloud in the KITTI dataset contains about 40,000 points, and each filtered point cloud in our subset of the Ford Multi-AV dataset contains between 150,000 and 300,000 points. Regarding the GPS datasets, we filter sequential duplicate points (the points that correspond to when GPS readings were logged while the user was stationary). Our filtered subset of the GeoLife and Mopsi datasets contain more than 1.4 million and 1.2 million points, respectively. The sizes of the bench-marked datasets are shown in Table 4.

Parameters: Our purpose of clustering LIDAR datasets is to detect scene objects, and our purpose of clustering GPS datasets is to detect areas attracting a lot of users. To that end, we choose ϵ and minPts to attain valid ground truth by the baselines. We identified that the baseline achieves reasonable clustering of scene objects with $\epsilon = 0.7$ and minPts = 10 for the KITTI dataset, and so it does with $\epsilon = 0.5$ and minPts = 100 for the Ford Multi-AV dataset. Furthermore, the baseline achieves reasonable clustering of GPS readings with $\epsilon = 0.1$ and minPts = 500 for the Mopsi dataset. On the other hand, as the GeoLife dataset contains much

[4] https://github.com/bjoern-andres/partition-comparison.

Table 4
Summary of the bench-marked datasets, showing the characteristics and the chosen clustering parameters for each dataset.

dataset	KITTI	FORD	GEOLIFE, shuffled GEOLIFE	MOPSI, shuffled MOPSI
number of points	40,000	150,000-300,000	1.4 million	1.2 million
inter-split overlap	low	low	medium, high	medium, high
(ϵ, minPts)	(0.7, 10)	(0.5, 100)	(0.001, 500), (0.001, 500/S)	(0.1, 500), (0.1, 500/S)
number of splits (S)	{2, 3, 4, 5, 10, 15, 20, 30, 36, 40, 50, 60, 70, 140}		{2, 3, 4, 5, 10, 15, 20, 30, 36, 40, 50, 60, 70, 100, 200, 400, 600}	
number of threads	{2, 3, 4, 5, 10, 15, 20, 30, 36, 40, 50, 60, 70}			

Table 5
Highlights of average elapsed completion time (in seconds) for different methods and datasets with a variety of parameters.

dataset		KITTI	FORD		GeoLife	Mopsi	
PCL		0.3598	1.8459		950	9829	
K=20	$PARMA_H$	0.0756	0.2304		21.6	91.7	
	$PARMA_F$	0.0762	0.2063		22.5	100.1	
	$FLEXI\text{-}PARMA_H$	0.0259	0.1074	S=140	1.3	3.24	S=600
	$FLEXI\text{-}PARMA_F$	0.0224	0.1016		1.5	3.5	
K=40	$PARMA_H$	0.0429	0.1363		13.7	35.2	
	$PARMA_F$	0.0418	0.1387		14.5	38.3	
	$FLEXI\text{-}PARMA_H$	0.0241	0.0800	S=140	0.94	2.06	S=600
	$FLEXI\text{-}PARMA_F$	0.0178	0.0803		0.97	2.16	
K=70	$PARMA_H$	0.0335	0.1038		4.9	19.9	
	$PARMA_F$	0.0601	0.1096		6.7	25.3	
	$FLEXI\text{-}PARMA_H$	0.0411	0.0827	S=140	0.8	1.54	S=600
	$FLEXI\text{-}PARMA_F$	0.0295	0.0756		0.85	1.58	

denser recordings, the baseline achieves reasonable clustering with parameters $\epsilon = 0.001$ and minPts = 500 for this dataset. For the LIDAR datasets, we execute flexi PARMA-CC algorithms choosing 20, 40, 70, and 140 for S. For the GPS datasets, as they contain much more points than the LIDAR datasets, we choose S among 100, 200, 400, and 600. We perform the experiments with up to 70 threads, except for the experiments in which S is less than 70, where we choose up to S threads. As the distribution of the points in randomly shuffled datasets becomes uniform among the splits, we adjust minPts with respect to S by using minPts/S. The aforementioned adjustment is a common practice, e.g., [17]. The chosen clustering parameters are summarized in Table 4 for each dataset.

Evaluation setup: We implemented PARMA-CC algorithms[5] in C++ and used GNU scientific library for matrix algebra. We used POSIX threads for multi-threaded programming. We used PCL's implementation of PCL-EC [37]. We employed elapsed real time to measure completion times. Experiments were run on a 2.10 GHz Intel(R) Xeon(R) E5-2695 system with 36 cores on two sockets (18 cores per socket, each core supporting two hyper-threads) and 64 GB memory in total, running Ubuntu 16.04. We only used hyper threading when there were more threads than the actual number of cores.

8.2. Completion time and the scaling-factor of PARMA-CC algorithms

Graphs plotted in the left Y-axes of Fig. 5 show the scaling-factor of PARMA-CC algorithms for different datasets with varying choices of K and S for the basic and flexi PARMA-CC algorithms. Besides, some highlights of the completion times are presented in Table 5 using varying number of threads and splits. Furthermore, the results of PCL-EC, as an exact sequential baseline, are included for the reference.

The results show that, with appropriate choices of S and large enough number of threads, PARMA-CC algorithms can be several orders of magnitude faster than the exact sequential baseline. Furthermore, the scaling-factor of PARMA-CC algorithms demonstrates a super-linear behaviour with respect to K or S for the GeoLife and Mopsi datasets. As both GeoLife and Mopsi datasets have highly skewed distributions, the complexity of the exact sequential baseline is $\mathcal{O}(N^2)$. The latter holds because the spatial data structure used to find ϵ-neighbourhoods is not able to operate efficiently with skewed data distributions. On the other hand, PARMA-CC algorithms reduce the completion time of the local clustering quadratically in K or S, by splitting the data and by approximation.

We notice that with a large enough choice of S, a flexi PARMA-CC algorithm achieves a higher scaling-factor than its basic counterpart. For example, with S being 600, the scaling-factor of a flexi PARMA-CC algorithm is about 6 times that of a basic PARMA-CC algorithm, as shown in Fig. 5c and Fig. 5s. Moreover, for each dataset, we observe that the scaling-factor of the flexi PARMA-CC algorithms tends to increase with greater S values. The latter is in accordance with Observation 1, stating the effect of increasing S on decreasing the completion time of local clustering. Furthermore, similar to the basic PARMA-CC algorithms, we observe super-linear growth of the scaling-factor for flexi PARMA-CC algorithms on the GeoLife and Mopsi datasets. Regarding smaller sets of data, we observe that increasing the number of threads beyond a certain point does not further decrease the execution time, as there is less work to be done and the benefit from distributing is opposed by the cost of coordination (Fig. 5m, Fig. 5q, Fig. 5n, Fig. 5r).

8.2.1. Spatio-temporal properties and the scaling-factor of PARMA-CC algorithms

On the KITTI and FORD datasets (with low inter-split overlap), $FLEXI\text{-}PARMA_F$ achieves the highest scaling-factor. The latter is shown in the left Y-axes in Fig. 5q-5r, and it is in accordance with Observation 2. On the other hand, $FLEXI\text{-}PARMA_H$ achieves the highest scaling-factor on the GeoLife and Mopsi datasets, see the zoomed graphs Fig. 6c and Fig. 6d, respectively.

The left Y-axes in Fig. 6a and Fig. 6b respectively show the scaling-factor of the basic PARMA-CC algorithms on the randomly shuffled GeoLife and Mopsi, datasets exhibiting high inter-split overlap. The results show $PARMA_H$ typically achieves a higher scaling-factor than $FLEXI\text{-}PARMA_F$ on the randomly shuffled datasets. The latter is in accordance with Observation 2. Another important observation is that PARMA-CC algorithms typically achieve higher scaling-factors on the randomly shuffled datasets. The latter holds because the splits of a randomly shuffled dataset contain approximately similar distributions, alleviating the worst-case behaviour of spatial data structures such as kd-tree.

8.2.2. Approximate DBSCAN clustering and the scaling-factor of PARMA-CC algorithms

The left axes in Fig. 7a-7d show the average scaling-factor of the basic PARMA-CC algorithms that utilize DBSCAN as the local clustering algorithm on the KITTI, FORD, GeoLife, and Mopsi datasets. For comparison, the speed-up of PDSDBSCAN (an exact parallel DBSCAN algorithm reviewed in § 2) is also provided in Fig. 7. Note that PDSDBSCAN fails to produce a proper clustering in Fig. 7c and crashes as it runs out of memory in Fig. 7d. The results show that PARMA-CC algorithms facilitate multiple times

[5] https://github.com/dcs-chalmers/PARMA-CC.

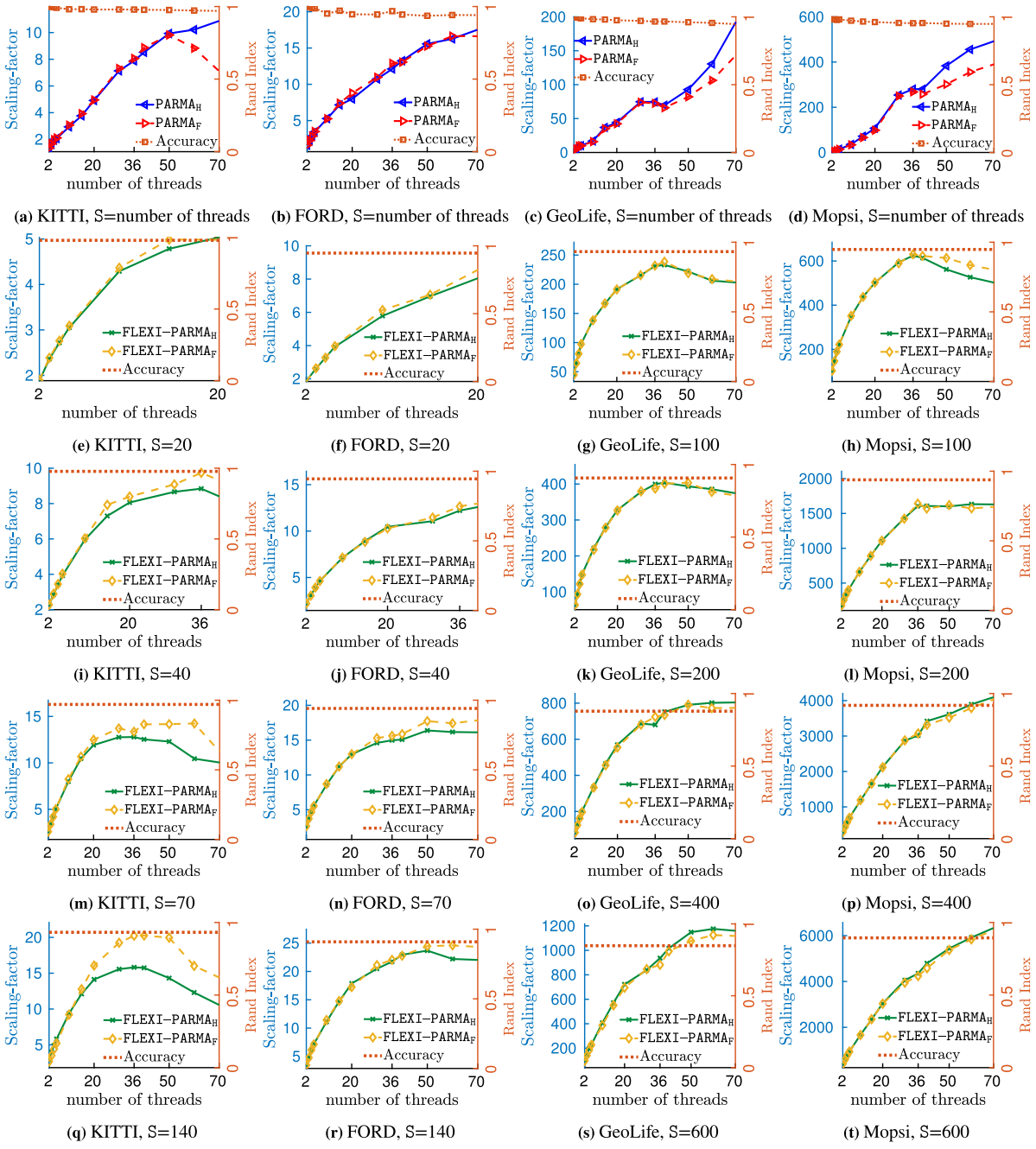


Fig. 5. The scaling-factor and rand index clustering accuracy of PARMA-CC algorithms. The scaling-factor of a PARMA-CC algorithm running with `K` threads is defined as the ratio of the completion time of the exact sequential baseline to the completion time of PARMA-CC algorithm running with `K` threads. Given a fixed `S`, all PARMA-CC algorithms achieve the same clustering accuracy.

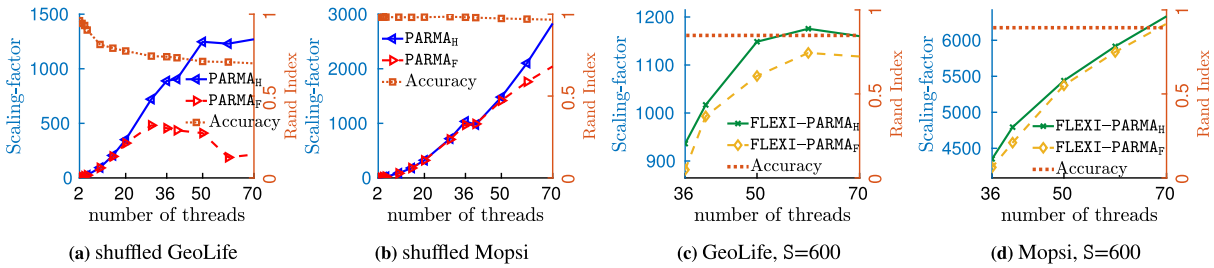
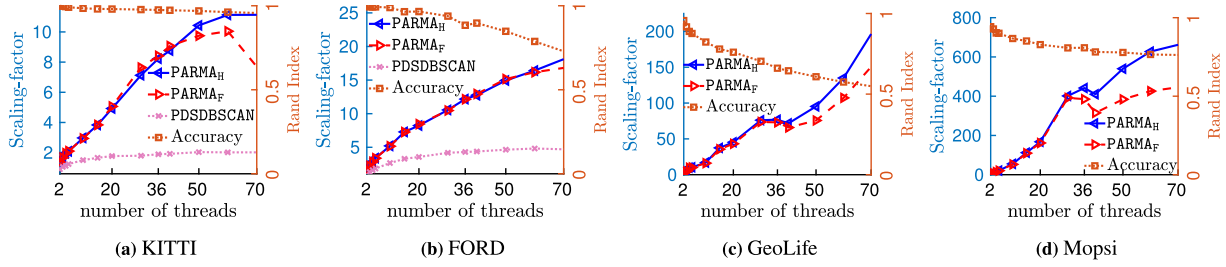
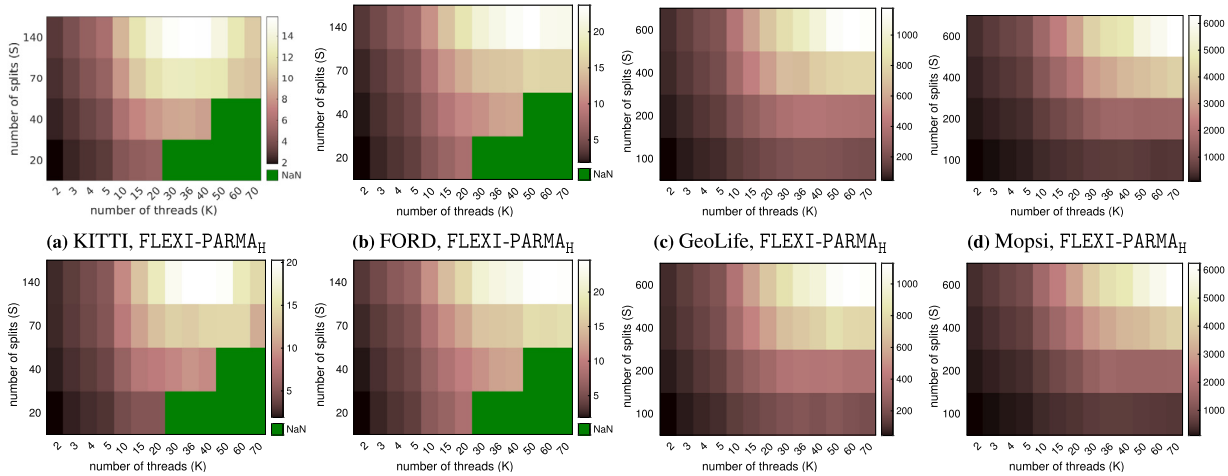


Fig. 6. The scaling-factor and rand-index clustering accuracy of basic PARMA-CC algorithms on the shuffled GeoLife and Mopsi datasets in (a) and (b). Zoomed scaling-factor and accuracy of flexi PARMA-CC algorithms on GeoLife and Mopsi datasets in (c) and (d).



(a) KITTI (b) FORD (c) GeoLife (d) Mopsi

Fig. 7. The scaling-factor and rand index clustering accuracy of basic PARMA-CC algorithms utilizing DBSCAN as the local clustering algorithm. For comparison, the speed-up of PDSDBSCAN (an exact parallel DBSCAN algorithm) is proved. In (c), PDSDBSCAN does not produce a proper DBSCAN clustering. In (d), PDSDBSCAN crashes as it runs out of memory. The right Y-axes show the clustering accuracy of basic PARMA-CC algorithms.



(a) KITTI, FLEXI-PARMA$_H$ (b) FORD, FLEXI-PARMA$_H$ (c) GeoLife, FLEXI-PARMA$_H$ (d) Mopsi, FLEXI-PARMA$_H$

(e) KITTI, FLEXI-PARMA$_F$ (f) FORD, FLEXI-PARMA$_F$ (g) GeoLife, FLEXI-PARMA$_F$ (h) Mopsi, FLEXI-PARMA$_F$

Fig. 8. The scaling-factor of FLEXI-PARMA$_H$ and FLEXI-PARMA$_F$ as a function of K and S (for K $\leq$ S) demonstrated by heat maps for different datasets. Moving between the columns of each heat-map indicates the effect of parallelization, and moving between the rows of each heat-map indicates the effect of approximation.

faster DBSCAN clustering through utilization of approximation and parallelization.

8.2.3. *Effects of parallelization and approximation on the scaling-factor of PARMA-CC algorithms*

We have seen so far how PARMA-CC algorithms utilize approximation and parallelization to gain in timeliness. We here aim to gain insight into the effects of approximation and parallelization on the scaling-factor. To that end, the heat maps in Fig. 8 summarize the scaling-factor behaviour of FLEXI-PARMA$_H$ and FLEXI-PARMA$_F$ for KITTI, FORD, GeoLife, and Mopsi datasets as a function of the number of threads and the number of splits via heat-maps (brighter colours indicate greater scaling-factors and the green parts correspond to the cases in which K is larger than S). Note that PARMA-CC algorithms yield equivalent clustering results with a fixed value of S (see Corollary 1). Therefore, moving between the columns of each heat-map indicates the effect of parallelization (i.e., number of threads), and moving between the rows of each heat-map indicates the effect of approximation.

8.3. *Relative ratio of local clustering to the completion time*

Fig. 9 shows the ratio of the duration of longest phase Ito the completion time in PARMA-CC algorithms. In all cases, with small values for K and S, ratio is very close to one because the local clustering phase constitutes the most significant duration in a PARMA-CC algorithm. Generally, for each dataset, as K or S increases, the aforementioned ratio decreases accordingly. The latter indicates the presence of two opposing phenomena. Firstly, the local clustering tasks get distributed more evenly among the workers, resulting in higher scaling-factor values. On the other hand, as indicated in Observation 1, too large values for K and/or S can increase the expected completion time of phase II, resulting in smaller scaling-factor values. In § 8.2, we empirically studied the joint effects of the aforementioned opposing phenomena on the overall completion time and the scaling-factor of PARMA-CC algorithms.

8.4. *Clustering accuracy*

The right Y-axes in Fig. 5a and Fig. 5b show the average accuracy of basic on the KITTI and FORD datasets, respectively. Furthermore, Fig. 5c and Fig. 5d show the accuracy of basic PARMA-CC algorithms on the GeoLife and Mopsi datasets, respectively.

Similarly, the right Y-axes in Fig. 5e-5t show the clustering accuracy of the flexi PARMA-CC algorithms for varying choices of S for each dataset. Note that, in each case, with a fixed value of S, PARMA-CC algorithms achieve the same clustering accuracy, as noted in Corollary 1.

The right Y-axes in Fig. 6a and Fig. 6b show the accuracy of basic PARMA-CC algorithms on the shuffled GeoLife and shuffled Mopsi, respectively.

The right Y-axes in Fig. 7a, Fig. 7b, Fig. 7c, and Fig. 7d shows the accuracy of basic PARMA-CC algorithms utilizing DBSCAN as the local clustering algorithm on the KITTI, FORD, GeoLife, and Mopsi datasets, respectively.

The results show that, although as S increases, the clustering accuracy of PARMA-CC algorithms gradually decreases, in most

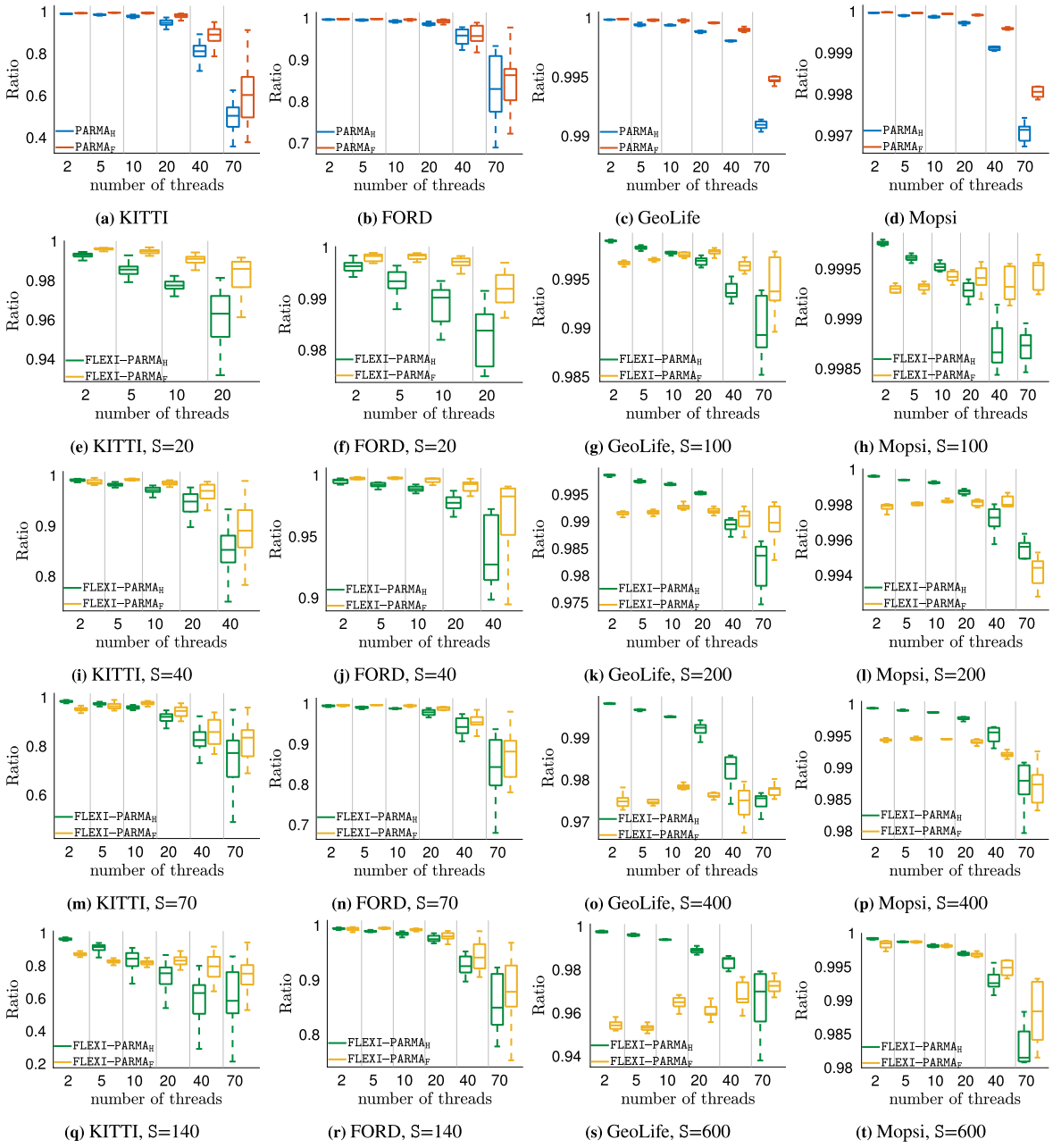


Fig. 9. Ratio of the duration of longest phase Ito the completion time in PARMA-CC algorithms.

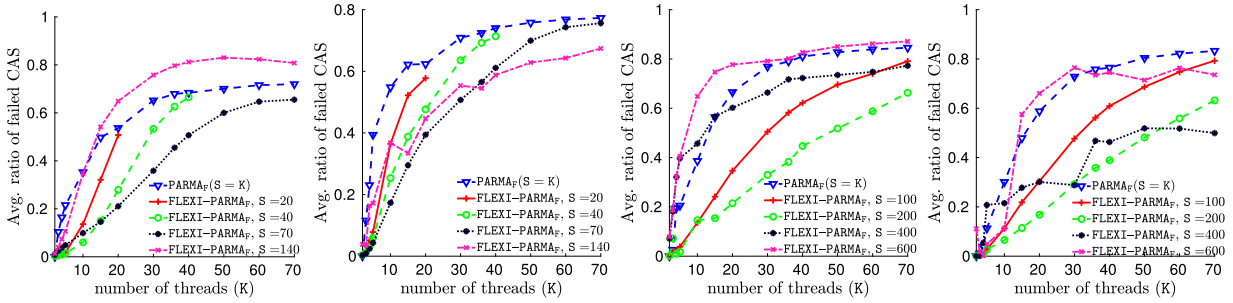
cases it stays high and this is due to the summarization properties of the bounding ellipsoids. Furthermore, PARMA-CC algorithms that utilize the Euclidean clustering algorithm are able to better keep up the accuracy compared to PARMA-CC algorithms that utilize DBSCAN.

8.5. Shared memory contention

As mentioned in § 6.4, there is no shared memory contention in $\texttt{PARMA}_{\texttt{H}}$, and the number of occasions in which shared memory contention can take place in $\texttt{FLEXI-PARMA}_{\texttt{H}}$ is significantly smaller than that of flat PARMA-CC algorithms (the aforementioned statements are also supported by the empirical measurements in Appendix A). Therefore, we focus on shared memory contention in flat PARMA-CC algorithms. Fig. 10 shows shared memory contention in those algorithms, as the average ratio of failed CAS operations to the total number of invoked CAS operations for $\texttt{K} \leq \texttt{S}$. Specifically, Fig. 10a, Fig. 10b, Fig. 10c, and Fig. 10d show the shared memory contention in $\texttt{PARMA}_{\texttt{F}}$ and $\texttt{FLEXI-PARMA}_{\texttt{F}}$ on the KITTI, FORD, GeoLife, and Mopsi datasets, respectively. The results show that shared memory contention in $\texttt{PARMA}_{\texttt{F}}$ is higher than that of $\texttt{FLEXI-PARMA}_{\texttt{F}}$, with a few exceptions. Furthermore, the results suggest that contention in $\texttt{FLEXI-PARMA}_{\texttt{F}}$ gets lower by choosing larger values of S, as it increases the number of shared tasks. However, the contention increases again if the chosen number of splits is too large for the amount of data, indicating that too large S values should be avoided for proper use of the algorithms.

8.6. Summary of the empirical evaluation

We studied timing and accuracy performance of PARMA-CC algorithms in a variety of situations. We saw how PARMA-CC algorithms can also have super-linear scaling-factors, as a result of approximation, when the datasets are skewed. We also saw that



(a) KITTI (b) FORD (c) GeoLife (d) Mopsi

Fig. 10. The average ratio of failed CAS operations to the total number of invoked CAS operations in flat PARMA-CC algorithms (for K ≤ S).

the local clustering is the dominant factor in the execution of a PARMA-CC algorithm. To that end, we noted that flexi PARMA-CC algorithms yield higher scaling-factors by increasing S (the number of data splits), up to a point justified by the volume of the data (the splits should not become too small, else the benefits of work-partitioning get counter-balanced by the overhead to coordinate the latter). Furthermore, with lower inter-split overlap, we observed that the flat PARMA-CC algorithms yield higher scaling-factors than the hierarchical PARMA-CC algorithms, and we noticed that the hierarchical PARMA-CC algorithms achieve higher scaling-factors when the inter-split overlap is high. We also showed in practice the trade-off between S and the scaling-factor when data is not too big, as well as the clustering accuracy of the algorithms, showing the advantage of suitable choice of S for utilizing the advantageous properties of PARMA-CC algorithms.

9. Related work

PARMA$_{\text{H}}$, the first parallel multiphase approximate clustering combining algorithm, was introduced and explored in [24]. The present paper extends the study of PARMA-CC algorithms by considering a design space along two orthogonal aspects. The first aspect considers how the threads synchronize and collaborate, and the second aspect considers how the workload gets distributed among the threads. As a result, the present paper introduces optimized algorithms targeting different places in the design space. As suggested by the extensive empirical evaluation, different PARMA-CC algorithms can be utilized according to certain properties of the data to be clustered.

In the following, we present three categories of clustering algorithms relevant to PARMA-CC algorithms.

CAT1 The methods in this category can directly be embedded in PARMA-CC algorithms as a local clustering algorithm to gain the parallelism benefits of PARMA-CC algorithms. For example, instead of DBSCAN, DENCLUE [21], STING [44], or OPTICS [4], and their approximate variants (see **CAT3**) can be employed. The algorithms in this category can utilize spatial data structures such as kd-trees [6], Octrees [31], R-trees [19], M-trees [9], and navigating nets [28]. Similarly, PARMA-CC algorithms can also incorporate the utilization of such spatial data structures in the local clustering phase. Moreover, with appropriately formed input, one can also employ Lisco [33], which is a single-pass continuous version of PCL-EC with faster ϵ-neighbourhood radius search via exploiting the angularly sorted readings of a LIDAR sensor.

CAT2 These methods boost the performance of classical clustering algorithms such as DBSCAN through parallelization. For instance, Highly Parallel DBSCAN [17], HPDBSCAN, is an OpenMP/MPI hybrid algorithm that redistributes the points to distinct computational units that perform the local clustering tasks. Then, the local clusters that need to get merged are identified, and thus appropriate cluster relabeling rules get generated, broadcasted, and applied locally. HPDBSCAN offers good scalability; however, when the data is skewed, its performance degrades severely. On the other hand, PARMA-CC algorithms can deal significantly better with skewed data as shown in the empirical evaluation. Moreover, PARMA-CC algorithms' approach to utilize the shared memory via in-place operations is more efficient that OpenMP's relaxed consistency memory model in which multiple copies of the same data might exist [22]. G-DBSCAN [3] is a parallel version of DBSCAN using GPU that employs a graph structure for indexing data. Other efforts on parallelizing DBSCAN employ a master-slave architecture, e.g., [5]. Nevertheless, PARMA-CC algorithms follow the orthogonal approach of scaling up before scaling out.

CAT3 Methods in this category sacrifice clustering accuracy to gain performance. For example, ρ-approximate DBSCAN [39], and STING (also in **CAT1**) which are both grid-based methods. The former gives a result that is *sandwiched* between those of DBSCAN with parameters $(\epsilon, \texttt{minPts})$ and $(\epsilon(1+\rho), \texttt{minPts})$, for an arbitrary small ρ. With a constant input dimensionality d, ρ-approximate DBSCAN has an expected $\mathcal{O}(\text{N})$ complexity. [12]. However, the number of neighbouring cells, $\mathcal{O}(1+(1/\rho)^{d-1})$, grows exponentially with the number of dimensions [39]. STING builds a hierarchical grid structure that divides the spatial area into rectangular cells, at a different resolution per level. Each cell summarizes the points it contains, thus approximating the clustering result of DBSCAN. With a smaller granularity step, the approximation gets better, but the number of bottom layer cells increases. Moreover, same as other grid-based methods, the number of grid cells increase exponentially with the number of input dimensions. Other methods integrate approximate nearest neighbour search techniques (e.g., those based on locality sensitive hashing) into DBSCAN, e.g., [39]. Another approximation approach is to cluster sampled data. To that end, for example, the dynamic (biased) sampling method in [27] can be utilized. The aforementioned techniques can as well be embedded in PARMA-CC algorithms.

10. Conclusions

To address the problem of parallel approximate distance- and density-based clustering, we explored a design space for synchronization and workload distribution among the threads. To cover different parts of the design space, we proposed representative PARMA-CC algorithms. We analytically and empirically provided evidence regarding capabilities of PARMA-CC algorithms to balance scaling and accuracy as well as to tolerate skewed data distributions. Furthermore, our studies show that certain properties in the input dataset can determine which PARMA-CC algorithm to choose for the best performance. Moreover, we showed that all PARMA-CC algorithms yield equivalent clustering results. We saw, furthermore, that the approximation technique can result in super-linear scaling-factor in the number of threads, with only marginal loss in accuracy. In general our results show that high-quality approx-

imate clustering can be several orders of magnitude faster than exact clustering. Based on the results of our extensive study of PARMA-CC algorithms, we provide some general guidelines related to parallel approximate data processing in the following:

- Regarding parallelization: In addition to the nature of the data processing task, some intrinsic properties of data also influence the required amount of synchronization among the threads. Fine-grained synchronization techniques are beneficial until certain threshold, but when heavy synchronization is needed, lock-based data parallel approaches can be more efficient. For example, several threads in a flat PARMA-CC algorithm can concurrently merge overlapping ellipsoids in split-summaries. Nonetheless, if the number of overlapping ellipsoids is large (which is a factor determined by the input data), then a large portion of fine-grained synchronization primitives will fail due to contention; consequently, the corresponding threads will need to retry. On the other hand, a thread in a hierarchical PARMA-CC algorithm can merge as many overlapping objects as required without any interruption, while other threads can in parallel merge the ellipsoids in mutually disjoint sets of objects.
- Regarding data structures: The choice of the data structures (and the computational complexity of the required functionalities) should be in accordance with the data processing task. For example, PARMA-CC algorithms utilize a union-set data structure supporting efficient `union` and `find` operations, which is in accordance with the *agglomerative* [32] nature of PARMA-CC algorithms. On the other hand, for a *divisive* [38] data clustering approach, the union-set data structure is probably not a good choice as it does not support efficient separation of sets. From an algorithmic implementation point of view, it is beneficial if the data structures support in-place operations utilizing pointer manipulation techniques.
- Regarding skewed data distributions: Classical data indexing methods used for data clustering can result in quadratic complexity in terms of the size of data under skewed data distributions. Our results show approximation can be a key idea for alleviating the challenges imposed by high skewness. Furthermore, our work shows that splitting a highly skewed data into a number of portions with similar distributions, and performing the required computation on the portions separately and then aggregating the results can reduce the required workload. Despite its approximate nature, our results show such an approach can attain high clustering accuracy.

We expect that PARMA-CC algorithms can facilitate pipeline processing of point clouds, especially combined with stream-processing oriented data structures as proposed in [18,43] and given the discussion about possible use-cases and associated queries in the respective section. Considering the observed parallelism-induced benefits of PARMA-CC algorithms, a possible future venue of studies and experiments is to adapt PARMA-CC algorithms to GPU enabled systems.

CRediT authorship contribution statement

Amir Keramatian: Conceptualization, Methodology, Gathering Data, Software, Writing, Reviewing and Editing. **Vincenzo Gulisano:** Conceptualization, Methodology, Supervision, Writing, Reviewing and Editing. **Marina Papatriantafilou:** Conceptualization, Methodology, Supervision, Writing, Reviewing and Editing. **Philippas Tsigas:** Conceptualization, Methodology, Supervision, Writing, Reviewing and Editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Software and data are available on github: https://github.com/dcs-chalmers/PARMA-CC

Appendix A

Fig. 11 shows the number of occasions in which shared memory contention can take place in PARMA-CC algorithms as the average number of failed `CAS` operations. The results show that memory contention in $\texttt{PARMA}_{\texttt{H}}$ is significantly lower than flat PARMA-CC algorithms.

References

[1] Siddharth Agarwal, Ankit Vora, Gaurav Pandey, Wayne Williams, Helen Kourous, James R. McBride, Ford multi-av seasonal dataset, CoRR, arXiv:2003.07969 [abs], 2020.
[2] Salvatore Alfano, Meredith Greer, Determining if two solid ellipsoids intersect, J. Guid. Control Dyn. 26 (2003) 106–110.
[3] Guilherme Andrade, Gabriel Spada Ramos, Daniel Madeira, Rafael Sachetto Oliveira, Renato Ferreira, Leonardo C. da Rocha, G-DBSCAN: a GPU accelerated algorithm for density-based clustering, in: Vassil, N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, Peter M.A. Sloot (Eds.), Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013, in: Procedia Computer Science, vol. 18, Elsevier, 2013, pp. 369–378.
[4] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, Jörg Sander Optics, Ordering points to identify the clustering structure, in: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, SIGMOD '99, ACM, New York, NY, USA, 1999, pp. 49–60.
[5] Domenica Arlia, Massimo Coppola, Experiments in parallel clustering with DBSCAN, in: Rizos Sakellariou, John A. Keane, John R. Gurd, Len Freeman (Eds.), Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference, Manchester, UK, August 28-31, 2001, Proceedings, in: Lecture Notes in Computer Science, vol. 2150, Springer, 2001, pp. 326–331.
[6] Jon Louis Bentley, K-d trees for semidynamic point sets, in: Proc. of the 6th Symp. on Comp. Geometry, Berkeley, CA, USA, June 6–8, 1990, ACM, 1990, pp. 187–197.
[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Yuli Zhou, Cilk: An efficient multithreaded runtime system, J. Parallel Distrib. Comput. 37 (1) (1996) 55–69.
[8] Robert D. Blumofe, Charles E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (5) (1999) 720–748.
[9] Paolo Ciaccia, Marco Patella, Pavel Zezula, M-tree: an efficient access method for similarity search in metric spaces, in: VLDB'97, Proc. of 23rd Int. Conf. on Very Large Data Bases, M. Kaufmann, 1997, pp. 426–435.
[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009.
[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA, AAAI Press, 1996, pp. 226–231.
[12] Junhao Gan, Yufei Tao, Dbscan revisited: mis-claim, un-fixability, and approximation, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, ACM, New York, NY, USA, 2015, pp. 519–530.
[13] Andreas Geiger, Philip Lenz, Christoph Stiller, Raquel Urtasun, Vision meets robotics: the KITTI dataset. I, J. Robot. Res. 32 (11) (2013) 1231–1237.
[14] Phillip B. Gibbons, Big data: scale down, scale up, scale out, in: 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015, IEEE Computer Society, 2015, p. 3.
[15] Craig L. Glennie, Derek D. Lichti, Static calibration and analysis of the velodyne HDL-64E S2 for high accuracy mobile scanning, Remote Sens. 2 (6) (2010) 1610–1624.
[16] Stefan Gottschalk, Ming C. Lin, Dinesh Manocha, Obbtree: a hierarchical structure for rapid interference detection, in: John Fujii (Ed.), Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1996, New Orleans, LA, USA, August 4-9, 1996, ACM, 1996, pp. 171–180.

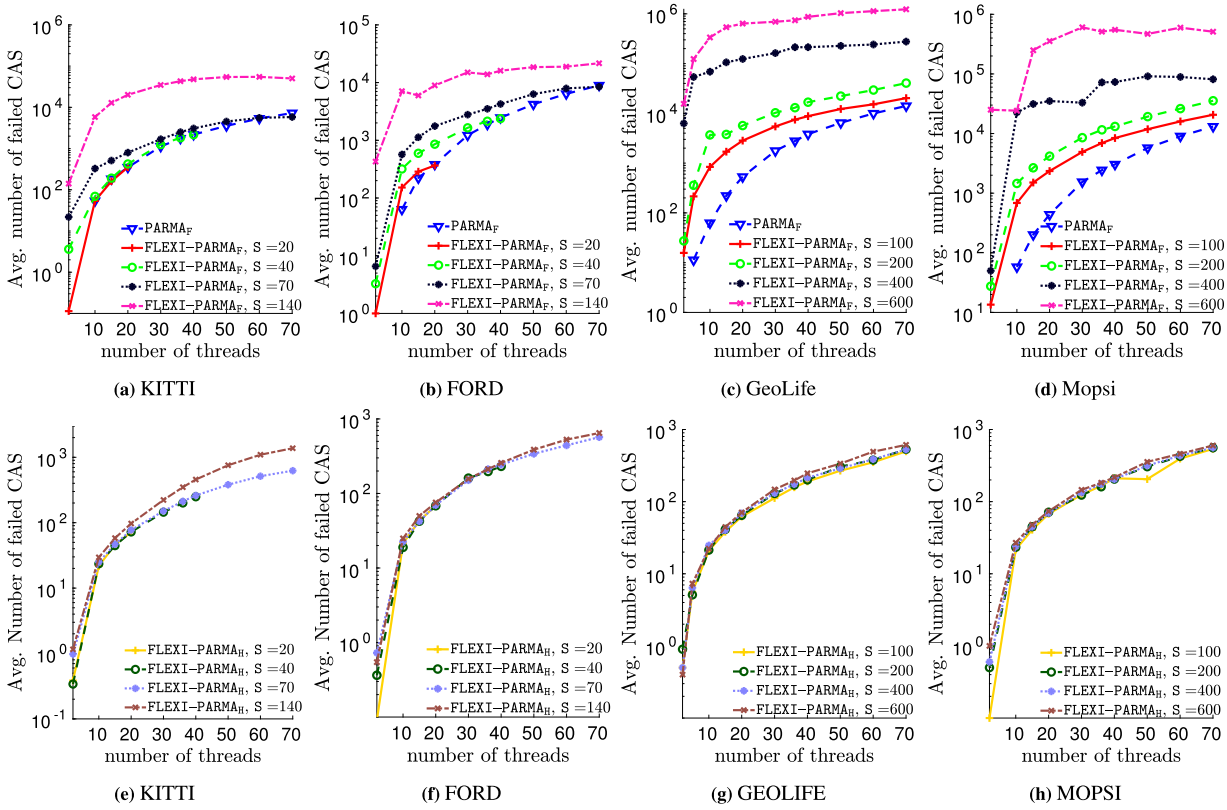


Fig. 11. The average number of failed CAS operations in PARMA-CC algorithms (for $K \leq S$).

[17] Markus Götz, Christian Bodenstein, Morris Riedel, Hpdbscan: highly parallel dbscan, in: Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC '15, ACM, New York, NY, USA, 2015, pp. 2:1–2:10.

[18] Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafilou, Philippas Tsigas, Deterministic real-time analytics of geospatial data streams through scalegate objects, in: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, ACM, 2015, pp. 316–317.

[19] Antonin Guttman, R-trees: A dynamic index structure for spatial searching, in: Proc. SIGMOD'84, ACM Press, 1984, pp. 47–57.

[20] Maurice Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. 13 (1) (January 1991) 124–149.

[21] Alexander Hinneburg, Daniel A. Keim, An efficient approach to clustering in large multimedia databases with noise, in: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98), New York, USA, August 27-31, 1998, AAAI Press, 1998, pp. 58–65.

[22] Jay P. Hoeflinger, Bronis R. de Supinski, The openMP memory model, in: Matthias S. Müller, Barbara M. Chapman, Bronis R. de Supinski, Allen D. Malony, Michael Voss (Eds.), OpenMP Shared Memory Parallel Programming - International Workshops, IWOMP 2005 and IWOMP 2006, Eugene, OR, USA, June 1-4, 2005, Reims, France, June 12–15, 2006, in: Lecture Notes in Computer Science, vol. 4315, Springer, 2005, pp. 167–177.

[23] Siddhartha V. Jayanti, Robert E. Tarjan, A randomized concurrent algorithm for disjoint set union, in: George Giakkoupis (Ed.), Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016, ACM, 2016, pp. 75–82.

[24] Amir Keramatian, Vincenzo Gulisano, Marina Papatriantafilou, Philippas Tsigas, PARMA-CC: parallel multiphase approximate cluster combining, in: Proceedings of the 21st International Conference on Distributed Computing and Networking, ICDCN 2020, New York, NY, USA, Association for Computing Machinery, 2020.

[25] Amir Keramatian, Vincenzo Gulisano, Marina Papatriantafilou, Philippas Tsigas, MAD-C: multi-stage approximate distributed cluster-combining for obstacle detection and localization, J. Parallel Distrib. Comput. 147 (2021) 248–267.

[26] Margret Keuper, Evgeny Levinkov, Nicolas Bonneel, Guillaume Lavoué, Thomas Brox, Bjoern Andres, Efficient decomposition of image and mesh graphs by lifted multicuts, in: ICCV, 2015.

[27] George Kollios, Dimitrios Gunopulos, Nick Koudas, Stefan Berchtold, Efficient biased sampling for approximate clustering and outlier detection in large data sets, IEEE Trans. Knowl. Data Eng. 15 (5) (September 2003) 1170–1187.

[28] Robert Krauthgamer, James R. Lee, Navigating nets: simple algorithms for proximity search, in: Proc. of the Fifteenth Annual ACM-SIAM Symp. on Discrete Algorithms, SODA 2004, SIAM, 2004, pp. 798–807.

[29] V.P. Kumar, A. Gupta, Analyzing scalability of parallel algorithms and architectures, J. Parallel Distrib. Comput. 22 (3) (1994) 379–391.

[30] Radu Mariescu-Istodor, Pasi Fränti, Grid-based method for GPS route analysis for retrieval, ACM Trans. Spatial Algorithms Syst. 3 (3) (2017) 8:1–8:28.

[31] Donald Meagher, Geometric modeling using octree encoding, Comput. Graph. Image Process. 19 (1) (1982) 85.

[32] Daniel Müllner, Modern hierarchical, agglomerative clustering algorithms, CoRR, arXiv:1109.2378 [abs], 2011.

[33] Hannaneh Najdataei, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, Continuous and parallel lidar point-cloud clustering, in: 38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018, IEEE Computer Society, 2018, pp. 671–684.

[34] Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, Alok Choudhary, A new scalable parallel dbscan algorithm using the disjoint-set data structure, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, Washington, DC, USA, IEEE Computer Society Press, 2012.

[35] Stephen B. Pope, Algorithms for ellipsoids, Cornell Univ., Rep. No. FDA, 2008.

[36] Radu Bogdan Rusu, Semantic 3d object maps for everyday manipulation in human living environments, Künstl. Intell. 24 (4) (Nov 2010) 345–348.

[37] Radu Bogdan Rusu, Steve Cousins, 3d is here: point cloud library (PCL), in: IEEE International Conference on Robotics and Automation, ICRA 2011, Shanghai, China, 9-13 May 2011, IEEE, 2011, pp. 9–13.

[38] Sergio M. Savaresi, Daniel L. Boley, Sergio Bittanti, Giovanna Gazzaniga, Cluster selection in divisive clustering algorithms, in: Robert L. Grossman, Jiawei Han, Vipin Kumar, Heikki Mannila, Rajeev Motwani (Eds.), Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002, SIAM, 2002, pp. 299–314.

[39] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, Xiaowei Xu, Dbscan revisited, revisited: why and how you should (still) use dbscan, ACM Trans. Database Syst. 42 (3) (July 2017) 19:1–19:21.

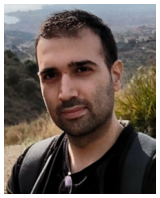
[40] Sergios Theodoridis, Konstantinos Koutroumbas, Pattern Recognition, Fourth Edition, 4th edition, Academic Press, Inc., Orlando, FL, USA, 2008.

[41] Gino van den Bergen, Efficient collision detection of complex deformable models using AABB trees, J. Graph. GPU Game Tools 2 (4) (1997) 1–13.

[42] Silke Wagner, Dorothea Wagner, Comparing clusterings - an overview, Technical Report 4, Universität Karlsruhe (TH), 2007.

[43] Ivan Walulya, Dimitris Palyvos-Giannas, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, Philippas Tsigas Viper, A module for

communication-layer determinism and scaling in low-latency stream processing, Future Gener. Comput. Syst. 88 (2018) 297–308.
[44] Wei Wang, Jiong Yang, Richard R. Muntz, Sting: a statistical information grid approach to spatial data mining, in: Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97, Morgan Kaufmann Publishers Inc., 1997, pp. 186–195.
[45] Thomas Willhalm, Nicolae Popovici, Putting intel threading building blocks to work, in: 1st Int. Workshop on Multicore Software Eng., IWMSE '08, ACM, 2008, pp. 3–4.
[46] Yu Zheng, Quannan Li, Yukun Chen, Xing Xie, Wei-Ying Ma, Understanding mobility based on GPS data, in: Hee Yong Youn, We-Duke Cho (Eds.), UbiComp 2008: Ubiquitous Computing, 10th International Conference, UbiComp 2008, Seoul, Korea, September 21-24, 2008, Proceedings, in: ACM International Conference Proceeding Series, vol. 344, ACM, 2008, pp. 312–321.
[47] Yu Zheng, Xing Xie, Wei-Ying Ma. Geolife, A collaborative social networking service among user, location and trajectory, IEEE Data Eng. Bull. 33 (2) (2010) 32–39.
[48] Yu Zheng, Lizhu Zhang, Xing Xie, Wei-Ying Ma, Mining interesting locations and travel sequences from GPS trajectories, in: Juan Quemada, Gonzalo León, Yoëlle S. Maarek, Wolfgang Nejdl (Eds.), Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009, ACM, 2009, pp. 791–800.



Amir Keramatian received the B.Sc. degree in Computer Engineering from Isfahan University of Technology, the M.Sc. degree in Artificial Intelligence from Sharif University of Technology, and the Ph.D. degree in Computer Science and Engineering from the Networks and Systems Division at Chalmers University of Technology. Amir's research interests include distributed and parallel processing, data structures, algorithms, pattern recognition, signal processing, and machine learning.

Vincenzo Gulisano Associate Professor in the Networks and Systems Division at Chalmers University of Technology. His research focuses on data processing and distributed / parallel / elastic and fault-tolerant data streaming. Dr. Vincenzo Gulisano holds a Ph.D. in Computer Science from the Polytechnic University of Madrid, Spain.

Marina Papatriantafilou Associate Professor, Chalmers Un.; earlier with the Max-Planck Inst. for Computer Science, Saarbruecken and CWI, Amsterdam, Ph.D. degree from the Computer Science and Informatics Dept., Patras Un. Member of Network of National Contacts ACM-WE NeNaC. Research interests: efficient and robust parallel, distributed, stream processing and applications in multiprocessor, multicore and distributed, cyberphysical systems; synchronization, consistency, fault-tolerance.

Philippas Tsigas received the B.Sc. degree in mathematics and the Ph.D. degree in computer engineering and informatics from the University of Patras, Greece. He was at the National Research Institute for Mathematics and Computer Science, Amsterdam, The Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrucken, Germany, before. At present, he is a professor in the Department of Computing Science at Chalmers University of Technology, Sweden. His research interests include concurrent data structures and algorithmic libraries for multiprocessor and many-core systems, communication and synchronization in parallel systems, power aware computing, fault-tolerant computing, autonomic computing, and scalable data streaming.