Scalable Distributed Trajectory Clustering Using Apache Spark

Stamatis Stefanopoulos[1], Charilaos Akasiadis[2], Nikos Pelekis[3] and Dimitris Zissis[4,5]

[1]University of the Peloponnese, Erythrou Stavrou 28 & Karyotaki, Tripolis, 22131, Greece
[2]II&T, NCSR "Demokritos", Patr. Grigoriou & Neapoleos 27, Agia Paraskevi, 15341, Greece
[3]Department of Statistics & Insurance Science, University of Piraeus, 80, M. Karaoli & A. Dimitriou St., Piraeus, 18534, Greece
[4]Department of Product & Systems Design Engineering, University of the Aegean, Konstantinoupoleos 2, Ermoupolis, Syros, 84100, Greece
[5]MarineTraffic.com

Abstract

Trajectory clustering is an important problem, where position data of mobile objects, such as vehicles and vessels, is analyzed to extract knowledge utilized for a plethora of management tasks. Recently, a vast increase in the production of data gathering devices has taken place, allowing the collection of data in much larger volumes. This challenges the application of existing clustering algorithms, as they are not always able to handle large datasets due to their design. In particular, TRACLUS is one of the most well-known trajectory clustering algorithms that is a generalization of DBSCAN for trajectory line segments. However, due to the iterative approach and the repetitive usage of a spatial index inherited from DBSCAN, TRACLUS's performance degrades as the datasets increase in size and can be extremely slow in some cases. To tackle this shortcoming, we propose a distributed implementation of TRACLUS, built on Apache Spark, that can operate on very large datasets by applying different types of partitioning to the input data. Results from an empirical evaluation on real-world trajectories illustrate that our distributed variant achieves improved runtime performance and clustering efficiency.

Keywords

Trajectory Clustering, Data Mining, Distributed Computing

1. Introduction

Clustering algorithms are valuable components in contemporary data analysis workflows. Being an unsupervised learning method, clustering analysis unveils the structure of the data and can be used as the basis for further learning [1]. The main objective of clustering algorithms is to divide data into groups, where members of the same cluster should be as similar as possible, while members belonging to other clusters should be quite different [2]. Due to the high diversity of the clustering problems properties, as these emerge from the features of the data to be clustered, a wide variety of clustering algorithms exists. Following different workflows, these algorithms incorporate various methods for problem solving, e.g. data partitioning, data hierarchy, fuzzy theory, distributedness, data density, graph theory, grid structures, fractal theory, or other data models [1].

Among these categories, density based clustering is widely used to handle real-world problems, as such algorithms can effectively discover arbitrarily shaped groups of data based on the density of their distribution in the n-dimensional space. From the algorithms of this category, DBSCAN [3] is one of the most well-known, as it comprises a simple, understandable concept that is also resilient to noise. DBSCAN's initial design applies clustering to a spatial distribution of points. However, there are other types of spatial information that would be valuable to consider. Moving object trajectories for example, also include timestamps due to their time series nature and thus constitute more complex data, that can currently be measured using a variety of sensors, for example the GPS of mobile devices that track routes. Clustering analysis helps to extract useful knowledge from such data, and there are many real-world applications where trajectories must be analysed in a density-based manner, such as movement anomaly detection [4] or traffic pattern extraction for vessels [5], vehicles [6, 7] or humans [8].

We focus on a generalisation of DBSCAN designed to cluster mobile object trajectories, through a partitioning-and-grouping procedure. This approach is called TRACLUS [9] and is used to cluster parts of trajectories, line segments in particular. One major drawback of TRACLUS however, is its inability to handle large trajectory datasets, as it does not scale up well, making big data clustering analysis difficult or extremely slow [10]. This is

Proceedings of the Workshop on Big Mobility Data Analytics (BMDA) co-located with EDBT/ICDT 2023 Joint Conference (March 28-31, 2023), Ioannina, Greece
dsc19025@go.uop.gr (S. Stefanopoulos); cakasiadis@iit.demokritos.gr (C. Akasiadis); npelekis@unipi.gr (N. Pelekis); dzissis@aegean.gr (D. Zissis)
https://www.iit.demokritos.gr/el/people/charilaos-akasiadis/ (C. Akasiadis); http://www.unipi.gr/faculty/npelekis/ (N. Pelekis); https://www.syros.aegean.gr/en/staff/professors-and-lecturers/associate-professors/dimitris-zisis (D. Zissis)
0000-0003-0785-4036 (C. Akasiadis); 0000-0001-7205-5703 (N. Pelekis)

mainly due to the continuous use of a spatial index to retrieve the nearby segments, and to the highly sequential and repetitive design that is inherited from the DBSCAN approach. To tackle this issue, paralellization techniques can be employed, especially in the algorithm parts that induce the highest runtime cost. Such techniques could also be integrated to existing analytics engines.

Motivated by these concerns, in this work we propose two methods to parallelize TRACLUS using the Apache Spark analytics engine.The first approach employs random partitioning of the dataset, while the second performs spatial partitioning, based on the position of the trajectory segments in the two-dimensional space. To show the validity of the results, a distance-based statistical significance measure is used to assess the overall clustering efficiency. This measure can be interpreted as an indicator of the algorithm's confidence regarding the formation of a particular cluster. The proposed distributed variants are both compared against the original, single-threaded implementation of TRACLUS, in terms of runtime and clustering accuracy. Our experimental evaluation illustrates the improved execution time of our approach as input datasets grow in size.

The rest of this paper is organised as follows: Section 2 presents the related work on trajectory clustering and distributed density-based techniques; Section 3 includes a detailed description of the proposed distributed TRACLUS approach. Then, Section 4 presents the experimental evaluation of the methods using real-world datasets and, finally, Section 5 presents the conclusions and some future work directions.

2. Related work

2.1. Trajectory clustering methods

Moving object trajectories is data of a particular form with distinct characteristics, which can be simply described as finite sequences of geolocations with timestamps [11]. Due to this unique form, special algorithms are designed for trajectory analysis that often incorporate trajectory clustering techniques. MovCloud [12] follows the MapReduce paradigm in cloud-based infrastructures, to index trajectory data hierarchically and proceed with the clustering using semantic information along with location coordinates. Considering trajectories as streaming data, the work of [13] utilizes both an online and an offline phase; trajectory clusters are updated dynamically to reflect incoming changes and are clustered to extract the final clustering result respectively. The method introduced in [14] is also based on the MapReduce paradigm to solve distributed tasks, such as the joining of subtrajectories, trajectory segmentation and clustering, and outlier detection. The approach of [15], adopts an indexing scheme for subtrajectories that groups them into nodes, in the philosophy of spatio-temporal partitioning techniques. This way, clustering results are calculated by simply applying query operators to the particular index. The knowledge discovery process presented in [16], extracts popular routes and movement patterns, and is shown to scale with big, real-world vessel mobility data, managing to detect real maritime incidents amongst them.

TRACLUS [9] has been considered as one of the state-of-the art algorithms for trajectory clustering [14, 17], as it is a generalisation of DBSCAN for density-based line segment clustering. TRACLUS partitions the trajectories to line segments before the clustering step, which in turn incorporates a specially designed distance metric. After clusters are extracted, representative trajectories are generated for the ones that fulfil particular criteria. This algorithm also inspired the design proposed by [18] that is an implementation less sensitive to input parameters, as well as [10], where a version of TRACLUS with partitioning and clustering routines designed for GPUs was introduced. In these approaches the trajectories are partitioned to line segments using the MDL principle [19] and are organized in graphs where breadth-first search is executed in parallel during the clustering task. A multi-threaded implementation of TRACLUS is also implemented for benchmarking. However, there is no openly available sourcecode of this variant and, furthermore, this is not a fully parallelized implementation as it only splits the distance between segment pairs related computations, by assigning different threads to different pairs. Instead, the approach that we propose aims to fully parallelize particular phases of the algorithm that influence its overall runtime performance.

2.2. Distributed DBSCAN

Many DBSCAN parallelization methods have been proposed in the literature so far, which are based on various paradigms. Among them, the most popular are the MapReduce-based methods, as the case of [20], which uses the disjoint-set data structure to break the sequential nature of DBSCAN and achieve better load balancing. The method generates a single tree for each point of the dataset and merges those generated at different partitions iteratively to calculate the overall clustering result. In PatchWork [21] the feature space is divided into a grid, and only cells containing more than a threshold of points are sorted by density and are kept for further processing. The most dense cell is considered the first cluster and nearby cells can extend it if their density overpasses a sufficient percentage. In [22] authors proposed a MapReduce method that partitions the dataset without overlapping, and executes DBSCAN in the partitions during the map phase, while it unifies the datasets and re-checks the noise points during the reduce phase.

MR-DBSCAN [23] proposes a three-step MapReduce procedure where the data points are in parallel partitioned, clustered, and then merged and relabelled. This method uses binary space partitioning (BSP) [24], an overlapping scheme to dynamically partition the dataset so that a balanced processing load on the cluster nodes is achieved, while retaining common points for the cluster merging phase. Moreover, a new partitioning method, CBP is introduced, which splits the dataset based on cost criteria. In [25] a similar procedure is followed, where clustering occurs in parallel in small data blocks of equal size and the results are combined during the reduce phase, using intermediate key-value pairs and hierarchical merging of the local clusters. For varying density clustering, VDMR-DBSCAN [26] uses MapReduce to divide the dataset into overlapping groups, perform the clustering step, and merge the resulting local ones according to their density and to partition adjacency criteria.

TRACLUS's relation to DBSCAN, along with its limited scalability and the absence of a published MapReduce-based parallel implementation to handle big data, led us to combine elements of some of the existing parallelization techniques of DBSCAN we discussed. In particular we incorporate BSP [23], the MapReduce strategy [22, 26], and intermediate key-value pairs [25], in order to achieve improved performance for larger datasets.

3. Methodology

We now present our distributed approach that is designed to scale up as the dataset size increases. To begin, the original TRACLUS algorithm consists of three main phases:

Trajectory partitioning. The moving object trajectories are partitioned into line segments, aiming to compress information without losing precision. To achieve this, the *Minimum Description Length* principle (MDL) [19] is used that minimizes the sum of two costs: the encoding cost for a hypothesis and, a second, regarding the data that is encoded according to this hypothesis. In our case, a hypothesis corresponds to a specific subset of all possible partitions of a trajectory.

Line segment clustering. At this phase, the clustering task takes place by applying DBSCAN to line segments, instead of points. A composite function which combines perpendicular, parallel and angle distances is used to calculate the interspace between the line segments and define neighbourhoods.

Representatives generation. The output of the clustering phase is a set of line segment clusters. For each one of them, a representative trajectory is generated based on the line segments belonging to the cluster.

In theory, the single-threaded implementation of TRACLUS is expected to require more time during the *line segment clustering* and the *trajectory partitioning* phases, where all dataset entries are examined. Moreover, the *line segment clustering* follows an iterative procedure, which increases the time complexity as the dataset increases in size. On the other hand, the *representatives generation* phase processes a smaller portion of the dataset, since not every entry is assigned to the final clusters. Thus, this step does not induce significant impacts to the runtime performance. For this reason, only the first and second phases are selected to be parallelized.

Table 1
Algorithm phases and assignment to the different processes.

TRACLUS phase	Random partitioning	Spatial partitioning
Trajectory partitioning	Worker	Worker
Global spatial index	–	Driver
Spatial partitioning	–	Driver
Local spatial indexes	Worker	Worker
Line segment clustering	Worker	Worker
Cluster merging	Driver	Driver
Repr. generation	Driver	Driver

The strategy used for the distributed implementation is based on the design that most Apache Spark programs follow. A driver (main) program initiates processing and a number of workers perform tasks concurrently and independently during the distributed processing (map) phases. The workers store their results in accumulator variables in the driver's memory and are processed by the driver when all workers have finished their jobs (reduce phase). An Apache Spark script may include multiple distributed processing phases. In our implementation, the main phases of the algorithm are shown in Table 1.

3.1. Distributed trajectory partitioning

During the *trajectory partitioning* phase, the dataset is divided randomly at the trajectory level and each worker receives a set of trajectories to process. The results are added to a list accumulator variable, to be unified in a single line segment dataset by the driver program for further processing in the next phases, as shown in Fig. 1.

3.2. Distributed line segment clustering

As already explained, *line segment clustering* is by far the most time-consuming phase , with its time cost growing significantly as the input dataset becomes larger. Thus, a distributed implementation for this part is imperative. We describe the methods used for dataset partitioning, i.e. random and spatial partitioning, and the merging strategies that unify the results produced by the workers.

The DBSCAN algorithm that uses R-tree spatial indexes has a run time complexity of $\mathcal{O}(n \log n)$ [3], where

a. b.

Figure 1: Distributed trajectory partitioning. The dataset is split randomly at the trajectory level (a) and is then passed to the workers, which partition the trajectories to line segments in a parallel fashion. All resulting line segments are stored in an accumulator variable managed by the driver program (b).

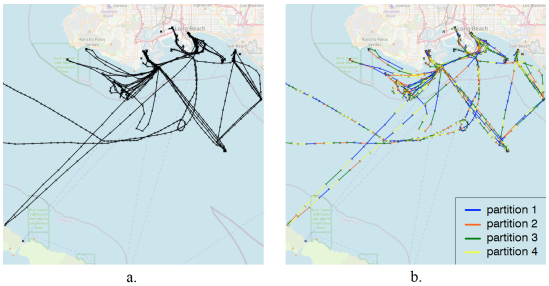
n is the number of the dataset entries. TRACLUS inherits this complexity [9]. Now, the splitting of the dataset into $k > 2$ separate and equal partitions yields a time complexity of:

$$\mathcal{O}(\frac{n}{k} \log \frac{n}{k})$$

As $k \cdot \frac{n}{k} \log \frac{n}{k} < n \log n$, overall, time complexity decreases as k increases.

3.2.1. Random partitioning (dTRACLUS-R)

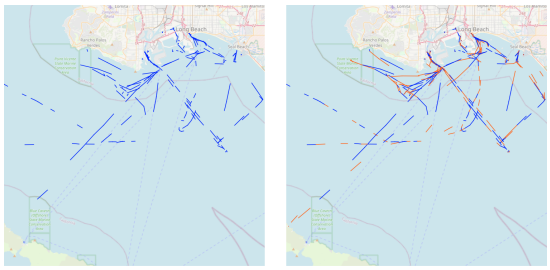
In the case of random partitioning of the line segments, a balanced split of the initial dataset is created, assigning approximately the same number of members to each partition (see, Figure 2). Then, partitions are assigned to separate workers, where local R-tree spatial indexes are created and used during *line segment clustering*.



a. b.

Figure 2: Random dataset partitioning. Dataset line segments (a.) are randomly assigned to partitions (b.). Each partition is colored differently.

Random partitioning creates disjoint sets of line segments, thus any couple of partitions does not contain any common members. This case can render cluster merging impossible after *line segment clustering*, as there will be no line segments belonging to two different clusters to be considered as "bridges", i.e. indicators of meaningful cluster merges. To avoid this, after partitioning, each worker receives *two* partitions of the dataset to build its spatial index (Figure 3). This leads in double-sized spatial indexes per worker (as they are created by two different dataset partitions instead of a single one), containing common line segments between partitions, as the index contains the segments of the corresponding partition and another one. This is useful for merging the local clusters discovered by each worker individually.



a. b.

Figure 3: Example using random partitioning (Figure 2). a. Line segments belonging to the blue partition (partition 1). b. Line segments used for partition's spatial index, contain also line segments from the orange partition (partition 2).

3.2.2. Spatial partitioning (dTRACLUS-S)

In the random partitioning approach, spatial characteristics of the line segments are ignored, making highly probable that segments which belong to the same cluster are assigned to different partitions. Splitting the dataset using spatial criteria leads to a better partitioning of the dataset for the purpose of trajectory clustering, as it takes into account each line segment's position in the two-dimensional space and groups nearby line segments to the same or nearby partitions, keeping the local (neighborhood) density of the dataset unaffected. This leads to more robust DBSCAN performance, as possible cluster candidates are less dispersed, thus less likely to be overseen as noise. For these reasons, we expect more precise results from the spatial rather than the random partitioning. Visual examples of spatial partitioning and the common line segments are given in Fig. 4.

The method used to split the two-dimensional space in partitions is based on the BSP [24] partitioning algorithm. First, all line segments are inserted in a spatial index structure and their minimum bounding rectangle is calculated. This rectangle is split in half recursively, trying to keep approximately the same number of line segments in each split's side. To determine the number of line segments in each side of the split, the previously

mentioned spatial index is used.

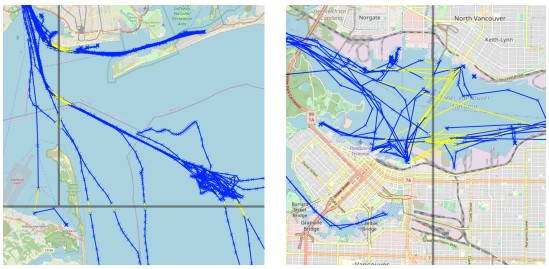


Figure 4: Examples of common line segments between spatial partitions. Spatial partitioning borders are depicted by the grey continuous lines. Common line segments between partitions are colored with yellow.

However, higher precision comes with some extra runtime and memory cost. Prior to partitioning, the whole dataset needs to be indexed in a global R-tree whose insert and search operations are equal to $\mathcal{O}(\log n)$. The R-tree is then queried every time a split on the dataset is attempted, to check if each side of the split contains approximately equal number of line segments. Since the search for better splits may require many iterations to converge to the most appropriate one, we stop when a balanced split is found, i.e. the number of line segments in one partition is between $\pm 5\%$ of the line segments in the other partition. Let $i \leq 2$ be the iteration limit, we anticipate an average of $\frac{i}{2}$ tries per split, resulting to 1 in the best case, and $\mathcal{O}(i)$ for the worst.

More formally, if n is the number of the line segments contained in the dataset, p the number of partitions, i the number of maximum tries per split, e the number of extra splits needed in case the number of partitions is not a power of 2, then for the spatial partitioning procedure the time complexity equals to:

$$(p + e - 1) \cdot \frac{i}{2} \cdot \log n$$

Thus, the total runtime complexity for the whole spatial partitioning phase becomes:

$$\mathcal{O}\Big(\big((p + e) \cdot i\big) \cdot \log n\Big)$$

This constitutes an additional effort that is induced by the spatial partitioning method, however this increase is negligible compared to the complexity gain that we have from the complete distributed implementation.

3.3. Merging local clusters

A crucial step for the integrity of the method's results is the merging of the discovered clusters. The process followed for the merging of local clusters into global ones is similar for the random and the spatial partitioning implemetations. The main difference lies in the way common members of the clusters emerge, which are later used for cluster merging.

3.3.1. Random partitioning (dTRACLUS-R)

For the random partitioning implementation, to be able to merge the local clusters we use extra line segments to build the spatial indexes each worker uses. Note though that these line segments are not considered as part of the worker's dataset. This means that these line segments can be returned by querying the spatial index, but are considered as not belonging to the partition dataset. During the *line segment clustering* phase, each time a cluster is expanded by one line segment, this line segment is checked if it belongs to the worker's dataset. If it does not belong to it, the line segment ID and the cluster in which it was assigned are added in a special "duplicates" accumulator to be further processed by the driver program after the end of this distributed phase. When the *line segment clustering* phase is completed, duplicate values are removed from the "duplicates" accumulator and the final merging couples are determined.

3.3.2. Spatial partitioning (dTRACLUS-S)

In the spatial partitioning case, the merging procedure is simpler, as each worker contains the same line segments in both its spatial index and its dataset partition, so there is no need to store possible duplicates in a special-purpose accumulator, as was the case in random partitioning. The common line segments that will be used as "bridges" for cluster merging are already specified by the spatial partitioning phase itself. This happens because line segments are two-dimensional objects that can span one or more spatial partitioning rectangles. This results to few common line segments between the spatial partitions. Thus, in the merging phase, the algorithm searches for these segments to use for the local cluster merging.

The final clusters obtained after the merging phase are expected to be slightly different from the clusters extracted from the single-threaded TRACLUS implementation. This is because in both cases, line segments that would normally be grouped to the same cluster, are dispersed in other partitions. It is possible that some of these partitions might contain so few cluster members, that they are considered as noise by the *line segment clustering* phase and never make it to the *cluster merging* phase. The rejected segments could either be considered as members of bigger clusters, or as common segments used to merge two or more clusters during the merging phase. This deviation is expected to be larger for the random partitioning implementation, as the dispersion of nearby line segments between the resulting partitions

is higher than dispersion caused by spatial partitioning.

3.4. Clustering significance

The final phase generates the cluster representatives, which act as cluster medoids and describe the average movement of the objects included in each discovered cluster. Intuitively, a representative that lies close to the segments of its corresponding cluster, describes it better than one lying further away. Randomly generated data in the wider area of the representative are expected to be more distant to the representative than to the real cluster members, as no spatial criterion is considered for their generation, contrary to the real data selected by spatial neighborhood queries during the clustering phase.

To evaluate the clustering efficiency of the single-threaded version and our proposed distributed one, we employ statistical significance tests, used to determine if the distances of the cluster members to the cluster representative are similar to the distances of random segments.

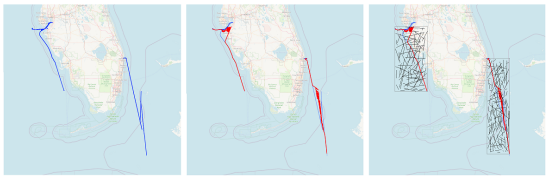


Figure 5: Calculation of clustering significance. Initial dataset (a). Generated representatives (red) with cluster members (real segments) (b). Random (fake) line segments generation inside the representatives' Minimum Bounding Rectangles (c).

To achieve this, for each generated cluster representative, its minimum bounding box is calculated. We generate random line segments inside this box, equal in number to the line segments that belong to the cluster. The next step is to calculate the Fréchet distance between the cluster members and each line segment of the representative, and store the minimum. The process is repeated for the randomly generated line segments. In the end, we come up with two distance distributions, which are tested using Z-test and Kolmogorov-Smirnov test, to obtain a measure of significant difference. This measure indicates if the randomly generated (fake) line segments are significantly further and dissimilar to the representative than the real segments of the cluster and can be used to distinguish between tightly (significant) and loosely (non-significant) connected clusters, acting as a measure of quality for the generated clusters and how well they are described by their representative. An example of this process is shown in Fig. 5.

4. Experimental evaluation

To test the proposed methods, we conducted experiments using a part of NOAA vessels dataset, with data of June 2019. The datasets used for the experiments contained approximately 50, 100, 200, 500, 1000, 2000, 5000 and 10000 trajectories, representing a total of 35K to 6M points. The average length of the datasets' trajectories spanned between 587 to 734 points, having a standard deviation ranging between 311 to 321 points.

Experiments were run on an Apache Spark standalone cluster installed on a VirtualBox virtual machine running Linux Ubuntu 18.04 with 14 CPU cores and 28 Gigabytes of RAM. Our implementation was based on Alex Polcyn's TRACLUS implementation in Python [27], which was properly modified to run in a distributed manner using PySpark according to the approaches we propose in Section 3. For the spatial indexing part, we use the Pyrtree [28], an R-tree implementation written in pure Python. Our implementation is openly available online.[1]

4.1. Line segment clustering runtime

The main goal of the distributed implementation is to make TRACLUS run faster for large trajectory datasets. By evaluating the single-threaded implementation we observe that the *line segment clustering* phase requires most of the overall runtime to complete (Figure 6).

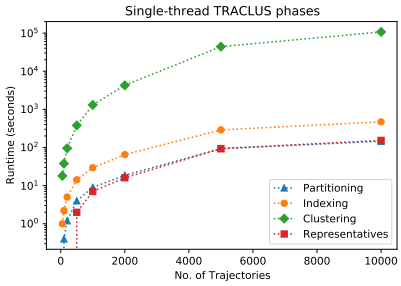


Figure 6: Single-threaded implementation runtimes.

Now, the distributed implementation of the *line segment clustering* phase significantly reduces the total runtime cost, as we can see in Figure 7. This outcome is anticipated based on the complexity analysis of the distributed implementation (cf. Section 3.2). The smaller size of the spatial indexes sent to the workers, enables them to complete respectively smaller processing tasks during the clustering phase in less time.

[1] https://gitlab.com/stamostef/traclus_pyspark

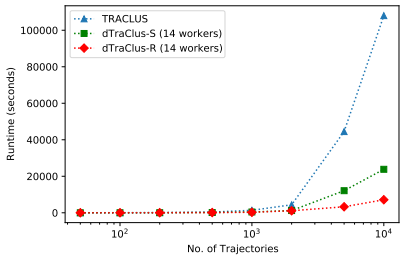


Figure 7: Total runtime performance.

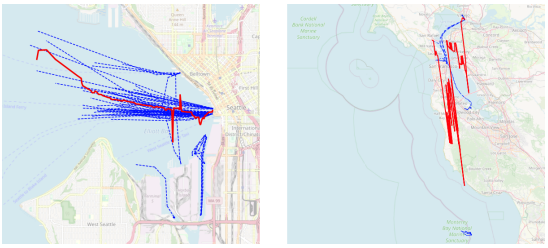


Figure 9: Examples of non-significant clustering. Clusters are loosely connected and the generated representatives (red) fall far away from the cluster members (blue).

4.2. Representatives and significance

A way to determine the quality of the algorithm's output between the presented implementations is the number of the representative trajectories generated at the final phase, compared to the number of representatives generated by the base line (single-threaded) implementation. A number of representatives close to the baseline implies both successful cluster discovery and good performance on local cluster merging. However, as mentioned in Section 3.3, there can be deviations between the single-threaded and the distributed implementation results due to the dispersion of line segments to different partitions.

Evaluation of the clustering significance shows that in most of the cases, significant scores for both the Z-test and Kolmogorov-Smirnov statistical significance tests are achieved. This phenomenon is due to the given TRACLUS hyperparameters (big neighborhood ratio) and the sparsity and nature of the dataset (vessel trajectories) which led to fairly dense clusters, compared to their surroundings. For the committed experiments, the highly significant clusterings illustrate tightly-connected clusters when plotted on a map, as shown in Figure 8.

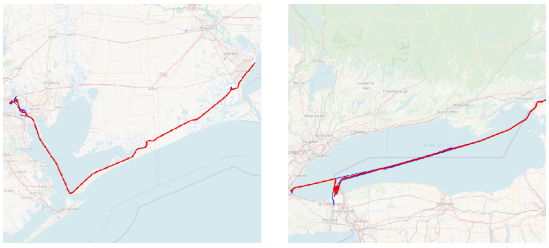


Figure 8: Examples of significant clustering. Clusters are tightly connected and the generated representatives (red) fall close to the cluster members (blue).

Clusterings of no significance were also present in some rare cases. Plotting their structures reveals loosely-connected clusters, often dispersed across multiple "sub-clusters" and not assigned to coherent ones (Figure 9).

This measure of statistical significance can be used for hyperparameter tuning, as a better choice for the *epsilon* TRACLUS parameter and can lead to more significant clustering in cases such as the ones presented in Figure 9. There, the inclusion of the remote line segments in the clusters could be avoided. Moreover, this measure can be used as a filtering criterion to indicate cases where representatives lie away from their corresponding cluster members and should be rejected.

4.3. Overall performance and scalability

We compared the total runtime results of the baseline single-threaded implementation with the distributed ones and, as we can see in Figure 7, the distributed implementations outperform by far the single-threaded one. We can also see that the difference in execution time increases as the datasets grow with respect to the number of trajectories and, consequently, the points they contain. To improve the distributed implementation performance for big data and achieve better scalability, more workers can be added. To take full advantage of the extra workers, the dataset partitions should be equal or more than the number of the available workers.

5. Conclusion and future work

In this paper we proposed two distributed TRACLUS implementations, based on random and spatial partitioning of the initial dataset. Experiments conducted on an Apache Spark cluster revealed that the proposed methods significantly outperform the original single-threaded TRACLUS implementation in terms of execution runtime as the dataset size increases. The clustering results of the distributed implementations maintain an acceptable clustering performance, which was evaluated using statistical significance tests. Possible improvements of the proposed methods could be a faster, possibly distributed, merging strategy for the local clusters. Moreover, studying the performance of the proposed methods with even

larger datasets on multi-node, distributed Apache Spark clusters can reveal more about the scalability of the proposed implementations.

Acknowledgments

This research is supported by European Union's Horizon 2020 RIA programme under grant agreement No 957237, project ENABLING MARITIME DIGITALIZATION BY EXTREME-SCALE ANALYTICS, AI AND DIGITAL TWINS (VesselAI).

References

[1] D. Xu, Y. Tian, A comprehensive survey of clustering algorithms, Annals of Data Science (2015) 165–193.

[2] A. Jain, R. Dubes, Algorithms for clustering data, Prentice-Hall, Inc, Upper Saddle River, 1988.

[3] M. Ester, H. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: Proc. of the Second Int. Conf. on Knowledge Discovery and Data Mining, KDD'96, AAAI Press, 1996, p. 226–231.

[4] R. Barnwal, S. Baride, S. Majumder, S. Ghosh, A density-based algorithm for detecting anomalous trajectories, in: Int. Conf. on Microelectronics, Computing and Comm. (MicroCom), IEEE, 2016, pp. 1–4.

[5] I. Kontopoulos, I. Varlamis, K. Tserpes, A distributed framework for extracting maritime traffic patterns, Int. Journal of Geographical Information Science 35 (2020) 1–26.

[6] X. Li, J. Han, J. Lee, H. Gonzalez, Traffic density-based discovery of hot routes in road networks, in: Advances in Spatial and Temporal Databases: 10th Int. Symposium, 2007, pp. 441–459.

[7] H. Munaga, M. Sree, J. Murthy, Dentrac: A density based trajectory clustering tool, Int. Journal of Computer Applications 41 (2012) 17–21.

[8] D. Das, D. Mishra, Unsupervised anomalous trajectory detection for crowded scenes, in: IEEE 13th Int. Conf. on Industrial and Information Systems (ICIIS), 2018, pp. 27–31.

[9] J. Lee, J. Han, K. Whang, Trajectory clustering: A partition-and-group framework, SIGMOD '07, ACM, New York, NY, USA, 2007, p. 593–604.

[10] H. Mustafa, C. Barrus, E. Leal, L. Gruenwald, Gtraclus: A local trajectory clustering algorithm for gpus, in: 2021 IEEE 37th Int. Conf. on Data Engineering Workshops (ICDEW), 2021, pp. 30–35.

[11] P. Sun, S. Xia, G. Yuan, D. Li, An overview of moving object trajectory compression algorithms, Mathematical Problems in Engineering 2016 (2016) 1–13.

[12] S. Ghosh, S. K. Ghosh, R. Buyya, Movcloud: A cloud-enabled framework to analyse movement behaviors, in: IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom), 2019, pp. 239–246.

[13] Z. Li, J. Lee, X. Li, J. Han, Incremental clustering for trajectories, in: Database Systems for Advanced Applications: 15th Int. Conf., 2010, pp. 32–46.

[14] P. Tampakis, N. Pelekis, C. Doulkeridis, Y. Theodoridis, Scalable distributed subtrajectory clustering, IEEE Int. Conf. on Big Data (2019).

[15] N. Pelekis, P. Tampakis, M. Vodas, C. Doulkeridis, Y. Theodoridis, On temporal-constrained sub-trajectory cluster analysis, Data Mining and Knowledge Discovery 31 (2017).

[16] D. Zissis, K. Chatzikokolakis, G. Spiliopoulos, M. Vodas, A distributed spatial method for modeling maritime routes, IEEE Access 8 (2020) 47556–47568.

[17] O. L. Hsu, C. Lee, Common sub-trajectory clustering via hypercubes in spatiotemporal space, IEEE Access 8 (2020) 23369–23377.

[18] C. Jiashun, A new trajectory clustering algorithm based on traclus, in: Int. Conf. on Computer Science and Network Technology, 2012, pp. 783–787.

[19] E. Keogh, S. Lonardi, C. Ratanamahatana, Towards parameter-free data mining, 2004, pp. 206–215.

[20] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. Liao, F. Manne, A. Choudhary, A new scalable parallel dbscan algorithm using the disjoint-set data structure, in: Int. Conf. on High Performance Comp., Networking, Storage and Analysis, 2012, pp. 1–11.

[21] F. Gouineau, T. Landry, T. Triplet, Patchwork, a scalable density-grid clustering algorithm, in: Proc. of the 31st Annual ACM Symposium on Applied Computing, ACM, 2016, p. 824–831.

[22] X. Hu, L. Liu, N. Qiu, M. Li, A mapreduce-based improvement algorithm for dbscan, Journal of Algorithms & Computational Technology 12 (2017).

[23] Y. He, H. Tan, W. Luo, S. Feng, J. Fan, Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data, Front. of Comp. Science 8 (2014).

[24] M. J. Berger, S. H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, IEEE Transactions on Computers C-36 (1987) 570–580.

[25] X. Fu, S. Hu, Y. Wang, Research of parallel dbscan clustering algorithm based on mapreduce, Int. Journal of Database Theory and Application 7 (2014) 41–48.

[26] S. Bhardwaj, S. Dash, Vdmr-dbscan: Varied density mapreduce dbscan, in: Big Data Analytics: 4th Int. Conf., volume 9498, 2015, pp. 134–150.

[27] A. Polcyn, traclus_impl, https://github.com/apolcyn/traclus_impl, 2016.

[28] A. S. Peleg, pyrtree, https://github.com/Rhoana/pyrtree, 2018.