Parallel Maritime Traffic Clustering Based on Apache Spark

Bo Liu
Faculty of Computer Science
Dalhousie University
Halifax, Canada B3H 1W5
boliu@dal.ca

Abstract

Maritime traffic patterns extraction is an essential part for maritime security and surveillance and DBSCANSD is a density based clustering algorithm extracting the arbitrary shapes of the normal lanes from AIS data. This paper presents a parallel DBSCANSD algorithm on top of Apache Spark. The project is an experimental research work and the results shown in this paper is preliminary. The experiment conducted in the paper shows that the proposed method can work well with maritime traffic data although the performance is not satisfying. A discussion about the method's limitation and potential issues is shown at the end of the paper.

key words: Maritime Surveillance; Clustering; Apache Spark

1 Introduction

Trajectory pattern mining has become a hot brunch topic of data mining in these years because of the increasing universality of some location report devices (GPS, AIS, etc.)[7]. A trajectory can be defined as a data type representing the movement of an object. More specifically, a trajectory can be represented by a multidimensional time series[1]. This representation for trajectories make it feasible to apply some traditional machine learning methodologies to deal with trajectory mining problems.

Today maritime transportation represents 90% of global trade volume and therefore the challenges related to safety and security aspects are of high priority[17]. AIS (Automatic Identification System) has been employed by vessels internationally currently. With the AIS information, including both state data (position, speed) and classification data (vessel type and size), it becomes more feasible for AIS anomaly detection. And the first phase for this task is to learn the normal traffic pattern of the vessels from the historical and real time data, which is also an important part in this paper. This is necessary because there exists no predefined lanes, unlike roads built by the countries or some organizations on the lands, in the ocean.

In this paper, I aim to design and implement a parallel version of DBSCANSD [12] which is a density-based clustering algorithm for maritime traffic patterns extraction. The design and implementation are based on Apache Spark [4], a fast and general-purpose cluster computing system for large-scale data processing.

The reason why I choose the topic of Spark is that Spark is a comparatively new technique which has been rapidly adopted by enterprises across a wide range of industries these days. And the community of Spark has become one of the largest open source communities in big data area. So working on the topic is challenging but definitely useful for my future study or work.

One expected contribution of this work is that it can provide a benchmark for researchers who would like to do optimization for DBSCAN[3] implementation on top of Spark. Another contribution is that this is not merely a design for original DBSCAN algorithm, it is a more complicated one for DBSCANSD [12]. This can provide an efficient way for maritime scientists to extract maritime traffic lanes from the AIS data.

To evaluate the results of my work, the first and important thing is the correctness of the implementation. Since the technique of Spark is new and there is no known solution, it is quite hard to design a correct algorithm. And actually this is the part where I spent most of my time. So in the experiment section, I first compare the results between the parallel version and the sequential one for the correctness verification. Then the second part is to evaluate the performance of the parallel one. I apply the algorithm on different sizes of data to compare the time spent for the tasks.

The rest of the paper is organized as follows. The following section introduces Apache Spark and GraphX. Section 3 gives a review of the related work done by other researchers. Section 4 proposes our approach for designing the parallel algorithm. Section 5 presents the results of experimental evaluation. In the final section, we conclude with a summary and discuss our method's limitations and the potential future work.

2 Brief Introduction to Apache Spark

Apache Spark was developed in 2009 in UC Berkeleys AMPLab[1] aimed for building a fast and general engine for large-scale data processing. From the benchmark result, Spark claims it can run 100x faster than Hadoop when Spark uses RAM cache, and 10x faster than Hadoop when running on disk[4]. Spark is developed in Scala and can provide API support for Java, Scala and Python, which makes it easy to build parallel apps.

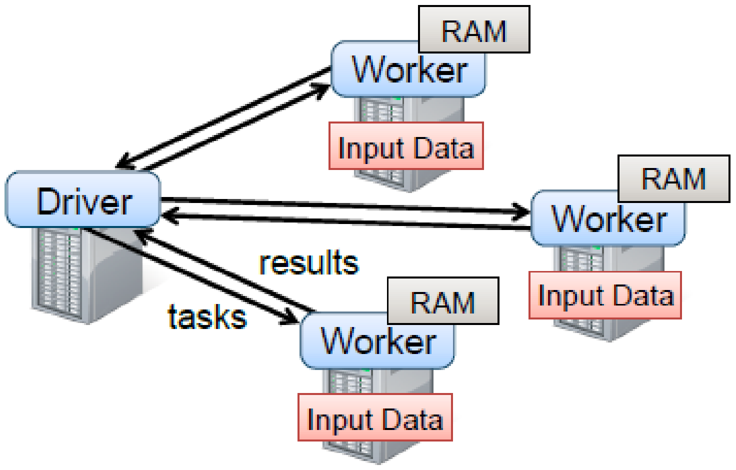


Figure 1: Spark runtime.[20]

[1]https://amplab.cs.berkeley.edu/

A general model of Spark runtime is demonstrated in Figure 1 [19]. To use Spark, developers need to write the high-level control flow of their applications and the driver program can launch multiple workers for the parallel tasks. The data blocks are read from a distributed file system (e.g. HDFS) and partitioned into the RAM of the workers. [19]

At the heart of Sparks architecture is the concept of RDD (Resilient Distributed Datasets). [16] An RDD is a read-only collection of objects partitioned across a set of machines designed for fault tolerance and in-memory cluster computing.

In Spark, each RDD is represented by a Scala object and can be constructed through deterministic operations on either (1) data in stable storage or (2) other RDDs. [19]. We can perform two operations on RDD, transformation and action. Figure 2 shows a lineage involving the two different operations. From the figure we can see that transformation operation will generate a new RDD from an existing one while the action will convert the last RDD into an output to external data sources.[11]. Examples of transformations include *map*, *filter* and *join* and that of action involve *count*, *collect*, *save* and *reduce*.

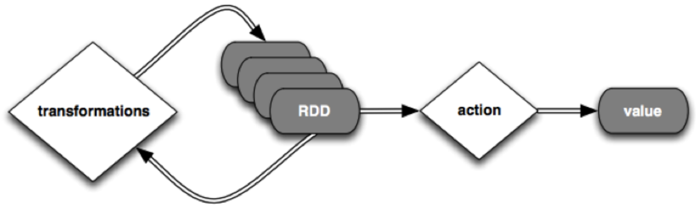


Figure 2: A typical execution sequence with two types of operations on RDDs.[2]

To implement the target of fault tolerance, an RDD stores information about the series of transformations used to build them (their lineage). [2]. More specifically, each RDD object contains a pointer to its parent and information about how the parent was transformed. [20] Thus when an RDD is lost, Spark will recompute the new RDD from its existing parent.

Another essential property of RDD is the support for in-memory computation. We can use *cache* action to alter the persistence of an RDD. When the memory is big enough, *cache* action will make the computed RDD kept in memory for future reuse. However, if the memory is not enough, Spark will recompute the RDD when the program needs it. [20]

However, since RDD is a read-only collection of objects, there are some applications not suitable for RDDs. The applications that make asynchronous fine-grained updates to share state can not benefit from the design of RDDs. [19] One typical example can be an incremental web crawler which needs to update the RDD frequently.

Another component I would like to introduce is GraphX. In this project, a graph-based algorithm was designed based on GraphX. GraphX is a graph computation engine built on top of Spark that enables users to interactively build, transform and reason about graph structured data at scale. [6]

The graph object defined in GraphX consists of directed adjacency structure and user-defined attributes for each vertex or edge.[18]. Just like Spark, we can apply $transformation$ and *action* on the graphs, each transformation can yield a new graph object.

The experiments done in [18] shows that GraphX is 8X faster than a general data-parallel platform (Mahout/Hadoop), but 7X slower than PowerGraph [8]. PowerGraph is a heavily optimized graph-parallel execution engine and one of the fastest open-source graph-parallel frameworks. And the authors in [18] believe that along with more effort done for

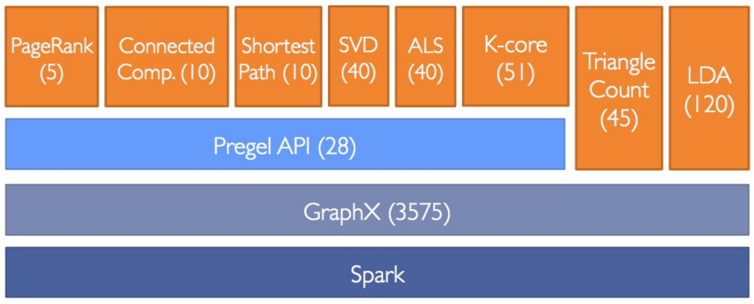


Figure 3: GraphX is a thin layer on top of the Spark general-purpose dataflow framework.[9]

the optimization, the gap between it and PowerGraph will be shortened in the near future.

Since our project is an experimental work, it is meaningful to try this new technology although it has not been optimized perfectly.

3 Review of the Relevant Literature

An unsupervised framework called TREAD is proposed by Giuliana Pallotta *et al.*[14] to detect low likelihood behaviours and predict vessels future positions from maritime traffic data. TREAD is a point-based framework and the traffic routes are built by the waypoints generated by the clustering process, that is, the route objects are directly formed by the flow vectors of the vessels whose paths connect the derived waypoints. This point-based idea has a practical advantage of handling trajectories of unequal length or with gaps, which is quite common in terms of unstable refresh rate of AIS messages. The difference between DBSCANSD[12] and TREAD is that DBSCANSD can consider the whole dataset during clustering phase instead of extracting the waypoints, which can make better use of the data set.

To parallel the algorithm of DBSCAN [3], researchers[15, 10] come up with different strategies for the implementation. In [15], Md. Mostofa Ali Patwary *et al.* employed the disjoint-set data structure to break the access sequentiality of DBSCAN and then use a tree-based bottom-up approach to construct the clusters. The experiments demonstrated that the method could yield a desiable balanced workload distribution. The graph concepts used in this method is also similar to our graph-based idea in our paper. Another work done by Yaobin He *et al.* called MR-DBSCAN [10] is a scalable DBSCAN algorithm using MapReduce. It involves a data partitioning method based on computation cost estimation to achieve a better load balancing even in the context of heavily skewed data set. The idea of data partitioning is effective and it has been proved by their experiments with large data set. Although we have not applied this idea in our work, we plan to implement a new algorithm to improve our algorithm's performance based on the similar method in the future.

4 Our Approach

In this section, I will first give a brief introduction to the DBSCANSD algorithm [12] and then propose our parallel algorithm based on Spark.

4.1 Introduction to DBSCANSD

DBSCANSD is employed for extracting the moving patterns from AIS data in [12]. It is a density based clustering algorithm which extends the original DBSCAN algorithm [3].

Algorithm 1 DBSCANSD

```
1: procedure DBSCANSD($DatasetM$, $eps$, $MinPts$, $MaxDir$, $MaxSpd$)
2:     Mark all points in moving dataset $DatasetM$ as unclassified
3:     $clusterList \leftarrow$ empty list
4:     for each unclassified point $P$ in $DatasetM$ do
5:         Mark $P$ as classified
6:         $neighborPts \leftarrow$ queryNeighborPoints ($DatasetM$, $P$, $eps$, $MinPts$, $MaxDir$,
   $MaxSpd$)
7:         if $neighborPts$ is not $NULL$ then
8:             $clusterList$.add($neighborPts$)
9:     for each cluster $C$ in $clusterList$ do
10:        for each cluster $C'$ in $clusterList$ do
11:            if $C$ and $C'$ are different clusters then
12:                if mergeClusters($C$, $C'$) is TRUE then
13:                    $clusterList$.remove($C'$)
14:    return $clusterList$
15: procedure queryNeighborPoints($data$, $P$, $eps$, $MinPts$, $MaxDir$, $MaxSpd$)
16:    $cluster \leftarrow$ empty list
17:    for each point $Q$ in data do
18:        if distance($P$,$Q$) $< eps$ then
19:            if $|P.SOG - Q.SOG| < MaxSpd$ then
20:                if $|P.COG - Q.COG| < MaxDir$ then
21:                    $cluster$.add($Q$)
22:    if $cluster$.size $> MinPts$ then
23:        Mark $P$ as core point
24:        return $cluster$
25:    return $NULL$
26: procedure mergeClusters($clusterA$, $clusterB$)
27:    $merge \leftarrow FALSE$
28:    for each point $Q$ in $clusterB$ do
29:        if point $Q$ is core point and $clusterA$ contains $Q$ then
30:            $merge \leftarrow TRUE$
31:            for each point $Q'$ in $clusterB$ do
32:                $clusterA$.add($Q'$)
33:            break
34:    return $merge$
```

The key idea of DBSCANSD is that for each point of a cluster the neighborhood of a given radius has to contain at least a minimum number of points with similar SOGs (Speed Over Ground) and COGs (Course Over Ground). With this idea, it can be feasible to handle the case like different types of ships can sail with different velocities in one specific area of the sea. For instance, the cruise speed of a cargo ship can be faster than a fishing ship.[12] Also the situation that the same type of vessels behave differently in relation to direction can also be addressed with the method.

The pseudocode [12] of the algorithm is shown in Algorithm 1. From Algorithm 1, we can easily get that the time complexity of it is $O(n^2)$. The proof of this can be referred in [12].

4.2 Design for Parallel DBSCANSD

In this section, the parallel version of DBSCANSD is proposed. The key idea of the design is to convert the problem to a graph-based one so that GraphX can be applied to the project besides the general Spark API.

Procedure 1 shows the design of our method and Figure 4 demonstrates the whole procedure of this design.

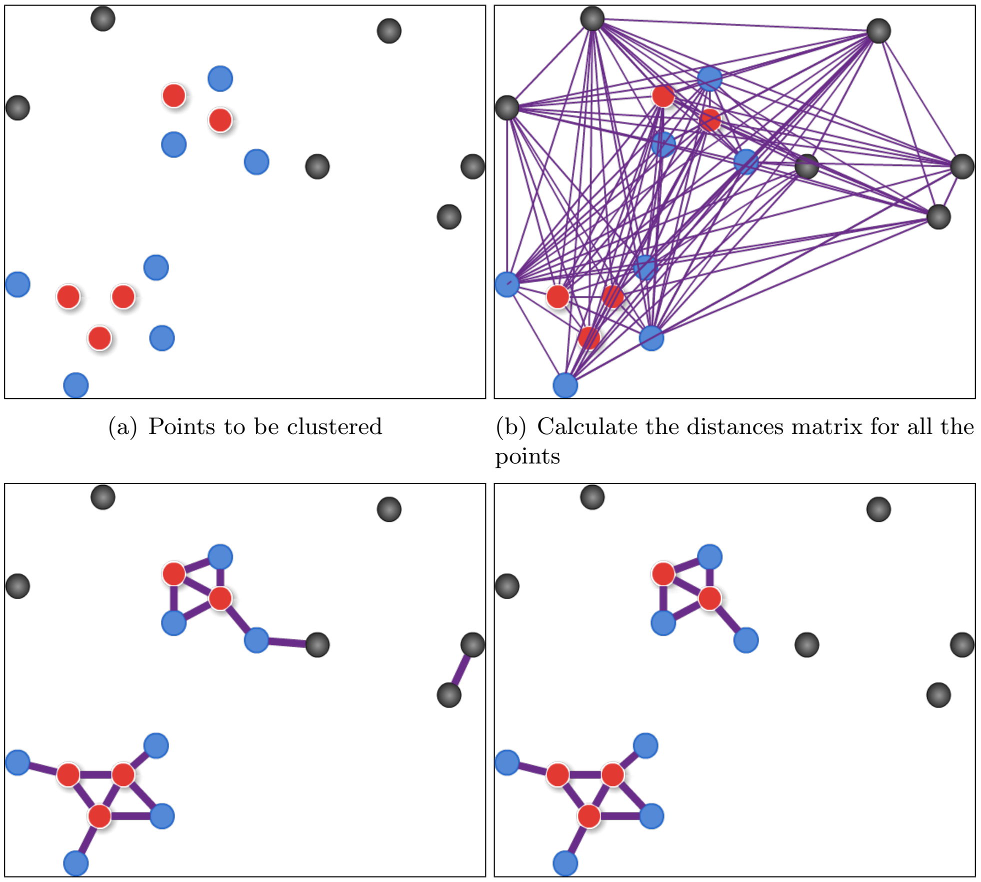
Procedure 1 Design for ParallelDBSCANSD(P, eps, $minNumPts$, $MaxDir$, $MaxSpd$)
Input: a list of all points in the data set, P; two parameters, reachable distance, eps; reachable minimum number of points, $minNumPts$;maximum direction variance, $MaxDir$; maximum speed variance, $MaxSpd$ Output: the resulting clusters, $result$.

Step (1) Compute a distance matrix (edges) for the points.
Step (2) Build the graph using all the vertices and the edges that are not longer than epsilon.
Step (3) Keep the vertices with degrees not smaller than $minNumPoints$, which are core points.
Step (4) Generate new edges between core points and all points (including core points) and filter the edges whose length are not longer than $epsilon$.
Step (5) Build a new graph using all the vertices and the edges generated from Step 4.
Step (6) Extract all the connected components from the new graph.

Algorithm 2 Parallel DBSCANSD

Input: (1) The input trajectory dataset, $points$;(2) reachable distance, eps; (3) reachable minimum number of points, $minNumPts$; (4)maximum direction variance, $MaxDir$; (5) maximum speed variance, $MaxSpd$
Output: The resulting clusters, $clusteringResult$.

```
1: $vertices \leftarrow$ VertexRDD($points$).cache;      $\triangleright$ convert the input data into RDD
2: $vertexCartesian \leftarrow vertices$.Cartesian($vertices$).cache
3: $edgesAll \leftarrow vertexCartesian$.map(Edge($vertex0$ , $vertex1$)).cache
4: $edges \leftarrow edgesAll$.filter( $|vertex0.SOG - vertex1.SOG| < MaxSpd$ AND $|vertex0.COG - vertex1.COG| < MaxDir$ AND Distance($vertex0$,$vertex1$)$<= eps$)
5: $initialGraph \leftarrow$Graph($vertices$, $edges$).cache
6: $corePoints \leftarrow initialGraph.outDegrees$.filter( $vertex0.outdegree >= minNumPts$ ).cache
7: $finalVerticesCartesian \leftarrow corePoints$.Cartesian($vertices$).cache
8: $finalEdges \leftarrow finalVerticesCartesian$.map(Edge(Distance($vertex0$,$vertex1$)$<= eps$ AND $vertex0$ , $vertex1$)).filter( $|vertex0.SOG - vertex1.SOG| < MaxSpd$ AND $|vertex0.COG - vertex1.COG| < MaxDir$).cache
9: $finalGraph \leftarrow$Graph($corePoints$, $finalEdges$).cache
10: $clusteringResult \leftarrow finalGraph$.connectedComponents().vertices
11: **return** $clusteringResult$
```


(a) Points to be clustered

(b) Calculate the distances matrix for all the points

(c) Discard the edges with length greater than the *epsilon* and rebuild the new graph. Label and store the vertices with degrees not smaller than $minNumPoints$ as core points (Red points).

(d) Build a new graph using all the vertices and the edges generated from Step 4. And two clusters will be generated using a connected component extraction algorithm.

Figure 4: The procedure of extracting the clusters using the parallel version of DBSCANSD. The red points are the core points and the blue points are the border points. The black ones are the noise, which should not be included in the final clustering results.

To implement the idea presented above in Spark, the distance matrix in Step 1 and Step 2 can be represented by a bool type of values to record if two points are close enough with similar speed and direction. The pseudocode of the algoirthm is shown in Algorithm 2.

In Algorithm 2, we use method of *cache* to keep the internal data into the memory to reduce the time for the computation. And we can see the method such as *Graph*,*Edge* and *ConnectedComponents* provided by GraphX API can make it easier for the developers to programming in Spark.

The overall time complexity for the algorithm is $O(n^2)$ because the 2nd line of the code includes a Cartesian Product operation on the whole data set. This operation takes $O(n^2)$ time and will be the main bottleneck for the whole program. The space complexity for it is $O(n^2)$ since it needs to store the Cartesian Product of the points. This is also a potential issue once the input data set is too big.

5 Experiments

In this section, I first validate the implementation of the Spark-based algorithm and then present some experiments results. I have done a lot more experiments which are not presented here because they have failed. The discussion about it will be put in the next section.

5.1 Correctness validation

As stated in Section 1, correctness is the basic requirement of the program but it took me most of the time. The reason for this is because Spark and GraphX API are new technology and there are not proposed solutions for the issue. A large number of iterations have been involved to achieve a correct implementation with expected output.

To validate the correctness of the algorithm, I select AIS data set from Juan De Fuca Strait area and use both sequential and the parallel version implementations on the data set.

The data set prepared for this experiment composed of 46,000 records (40,000 moving points and 6,000 stopping points distinguished by the SOG threshold 0.5 knot) selected from two-months of trajectory data between November 1 and December 31 in 2012.

After applying the two versions of clustering algorithm, 14 different clusters including 13 moving clusters and 1 stopping cluster are extracted. The result can be seen in Figure 5. Points in blue are the original traffic points while others are the cluster points. The size (total number of the points composing the clusters) of moving clustering results is 25,617.

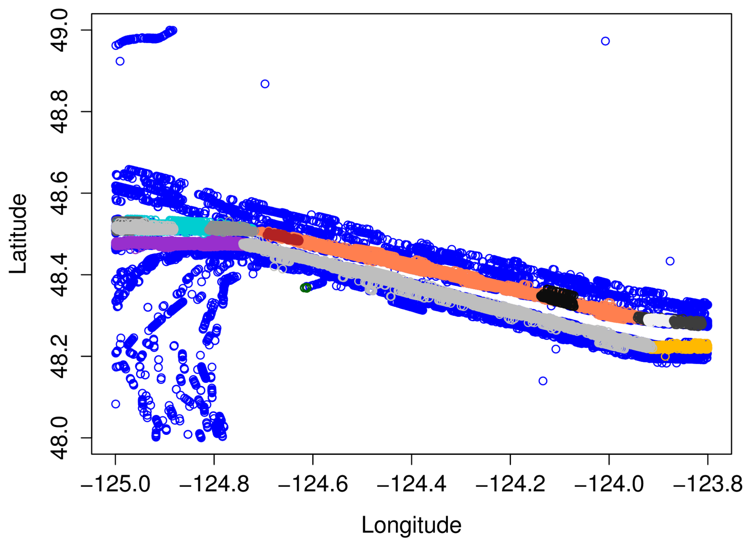


Figure 5: The clustering results of JUAN DE FUCA SRAIT area. Different colors stand for differnt clusters.

5.2 Performance Evaluation

After getting the correct implementation, I start to evaluate the performance of the algorithm. The experiment is done in Amazon EC2[2]. Two m3.large type of instances are selected for the experiment (one as the master instance and the other one is the slave instance). The data sets are uploaded to HDFS and Apache Spark environment is set up based on the the instructions in [5, 13]

The data sets are extracted from the AIS data in the North Pacific Ocean from November 1st to November 30th. 5 different sizes of data sets are extracted and the number of the points in the 5 data sets are 12657, 21494, 40849, 62343 and 87657. The time spent for these 5 data sets using our algorithm is shown in Figure 6. The curve shows a exponential increase while the size expands, which is acceptable because the time complexity of the algorithm is $O(n^2)$.

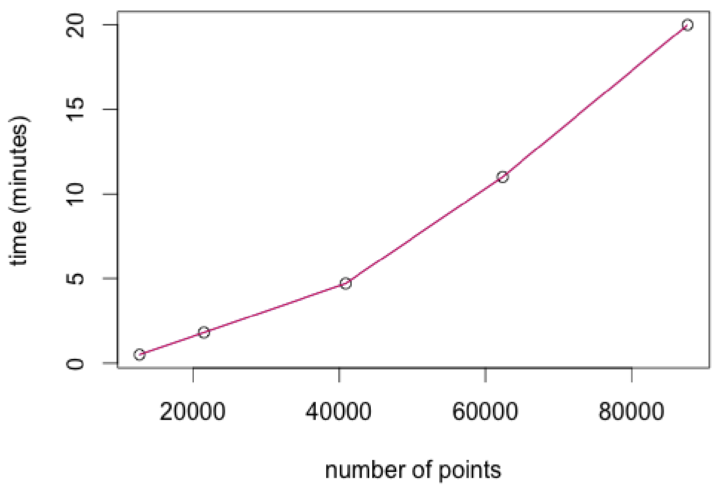


Figure 6: The clustering results of JUAN DE FUCA SRAIT area. Different colors stand for differnt clusters.

6 Discussion and Conclusion

The problem of extracting normal lanes from historical maritime traffic data is the first step of the vessels' behaviours anomaly detection task. And in this paper, we try to design a parallel version of DBSCANSD using Spark.

The experiments done in the previous Section show that the implemented algorithm can work well using Spark APIs. However, I cannot see a big speedup after comparing the results with my sequential implementation. After a comprehensive analysis, I came up with the following possible reasons.

Firstly, the algorithm of DBSCANSD is not an iterative one. More specifically, after caching the data into the memory, there are not many iterations that reuse the data in

[2]http://aws.amazon.com/ec2/

other actions. One example of iterative algorithm is K-Means, K-Means will iteratively partition the data set until the convergence has been reached, which can benefit a lot from the in-memory caching technique.

Secondly, the design of the sequential DBSCANSD is different from that of our Spark-based one. From Algorithm 1 we can see that there exists some cut downs. For example, when the program finds $MinPts$ number of points that are close enough to a particular point, it will stop searching for other points; instead, it will mark the point as core point. However, in the parallel version, we first compute the Cartesian product of all the points, which causes more computations.

Besides, I also do experiments on multiple instances in Amazon EC2, but the results are disappointing. It can take around five minutes to finish a task using merely one instance whereas it takes more than 2 hour to finish using 3 instances. One reason for this can be because the data set is not big enough while it requires a vast computation. Another reason is that Cartesian method employed in the implementation is a wide transformation [16] involving data shuffling which needs much communication overhead between instances. To improve this, I rewrote the algorithm based on broadcast variable. In the new program, the data will be first broadcast to all the worker nodes and then conduct the Cartesian product locally. However, the result has not been improved, instead, it took more time than the current version of program. For example, it takes 4 minutes to finish one task with 4 cores in my local computer using the original code while the broadcast version program takes 8.8 minutes to finish.

Although this experimental work does not have a satisfying performance result, it is still meaningful for me to give this try. The work done can give a good start for other potential optimizations. To improve the performance of this algorithm, I plan to design a new partition-based version. The general idea for this is to first partition the AIS data based on the Geo-location and then apply our clustering algorithm proposed in this paper. If the data can be partitioned well, the time complexity can be reduced to linear which can make it feasible for entire ocean's data clustering. It is challenging but worthy and I put it as my future work.

References

[1] Charu C Aggarwal and Chandan K Reddy. *Data Clustering: Algorithms and Applications.* CRC Press, 2013.

[2] Databricks. *Introduction to Apache Spark.* http://training.databricks.com/workshop/.

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

[4] The Apache Software Foundation. *Apache Spark: Lightning-fast cluster computing.* https://spark.apache.org.

[5] The Apache Software Foundation. *Running Spark on EC2.* https://spark.apache.org/docs/0.9.0/ec2-scripts.html.

[6] The Apache Software Foundation. *Spark GraphX.* https://spark.apache.org/graphx/.

[7] Fosca Giannotti, Mirco Nanni, Fabio Pinelli, and Dino Pedreschi. Trajectory pattern mining. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 330–339, New York, NY, USA, 2007. ACM.

[8] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[9] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[10] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, 2014.

[11] Ricky Ho. *Spark: Low latency, massively parallel processing framework.* http://horicky.blogspot.ca/2013/12/spark-low-latency-massively-parallel.html.

[12] Bo Liu, Erico N.de Souza, Stan Matwin, and Marcin Sydow. Knowledge-based clustering of ship trajectories using density-based approach. In *2014 IEEE International Conference on Big Data.*

[13] Chris Paciorek. *An Introduction to Using Distributed File Systems and MapReduce through Spark.* http://www.stat.berkeley.edu/scf/paciorek-spark-2014.html.

[14] Giuliana Pallotta, Michele Vespe, and Karna Bryan. Vessel pattern knowledge discovery from ais data: A framework for anomaly detection and route prediction. *Entropy*, 15(6):2218–2245, 2013.

[15] Md Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. A new scalable parallel dbscan algorithm using the disjoint-set data structure. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[16] Abhinav Sharma. *Apache Spark.* http://dowhilezero.blogspot.ca/2014/07/apache-spark.html.

[17] Michele Vespe, Ingrid Visentini, Karna Bryan, and Paolo Braca. Unsupervised learning of maritime traffic patterns for anomaly detection. In *Data Fusion & Target Tracking Conference (DF&TT 2012): Algorithms & Applications, 9th IET*, pages 1–5. IET, 2012.

[18] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[19] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[20] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

Appendix

Logs For Project Progress

I started my work from setting up the environment for Spark. Because I think it should be better for me to learn what is Spark through programming. However, it is not that easy to do these configuration things. Around 3 days have been spent on this. Here is just one example, to compile Spark code in Java or Scala, one choice is to use Maven. So I followed the instructions online to download and install maven in my own laptop. Then I wrote a test program to test if it can work well. The whole process took me around 2 hours. Then to make it work in EC2, I registered for an account and then spent a whole night for configuration between Spark and EC2.

The next thing is to learn the APIs supported by Spark. It supports Java, Scala and Python. Since I used Java before, I fist decided to write some Java programs at the beginning.

After trying to write some simple programs in Java API provided by Spark, I found this was a terrible experience. First reason for this is that Spark was written in Scala instead of Java. As a result, you can write a program which is just 2 lines in Scala but you may require 8 lines in Java. Secondly, since Spark is such a new technique, there are not many tutorials online. Then even though I can find some tutorials, they are taught in Scala API instead of Java API, which makes this course project more challenging and infeasible. So I decided to learn Scala for Spark.

To learn some basic syntax for Scala took me around half day and it took me another half day to do some practice with Scala, such as writing some toy programs.

After these trivial but time-consuming tasks, I can finally start to design my Parallel program using Spark. The design spent me around 1 day but the implementation spent me incomputable hours. This was the most horrible part to me because I was not familiar with Scala and GraphX although I spent one day on leaning the syntax. So I had to try again and again until it could give me a correct output. I really have no idea how many hours I spent on this. But fortunately, it could work finally.

But after finding my Spark program could not achieve a satisfying performance result, I started to read the original papers for Spark to figure out the possible reasons. To understand Spark more deeply it took me at least two days.

Then I tried to use broadcast variable supported by Spark to improve the performance of my code. It took me over 6 hours to implement a no-bug program. However, the result was still not improved. Then the other day, I tried again to do multiple modifications to the original program using broadcast scheme and it took me another half day for doing experiments to compare the results. I found in this way it could give an improvement if more cores were used. However, the final results could still not be improved compared to the original implementation using Cartesian directly.

Finally, I decided to give up because it was the time for writing the report for the course. I am not good at writing, so it took me 3 days on this report.

There are more things I did not mention, but I believe the whole process took me at least 150 hours although the final result is not satisfying. I think I learn quite a lot these days thanks to the challenging topic.