Parallel DBSCAN using complex grid partitioning

Data Mining Assignment - BDMA - November 2019

Fabrício Ferreira
000489746

Iryna Nazarchuk
000493334

Ledia Isaj
000491124

1. INTRODUCTION

Data clustering, one of the unsupervised learning techniques found in data mining, aims to identify meaningful clusters within a given dataset by grouping instances based on their similarity. Such a technique can be used in different contexts (e.g., image analysis, pattern recognition, epidemiology) and has proven valuable for knowledge discovery. One of the most common algorithms, k-means, is a partitioning method that can find clusters of spherical shape. However, there are some contexts where the clusters present themselves in other forms than circular, and thus, k-means would be ineffective.

The Density-based spatial clustering of applications with noise (DBSCAN) is a density-based data clustering algorithm introduced by Ester et al. (1996) and capable of identifying clusters with arbitrary shapes effectively handling noisy data and outliers. By considering two parameters, a neighborhood radius and a minimum number of points of a dense neighborhood, DBSCAN forms clusters out of adjacent high-density areas.

The original algorithm has quadratic complexity, which can be improved by the adoption of an index structure for accelerating the search of nearest neighbors. Nonetheless, with the evolution of information technology and the increase of the volume of data, efforts are being made to propose other algorithms to improve DBSCAN performance for Big Data contexts. Besides focusing on indexing structures [8], most works are being conducted towards parallel alternatives, since the accession of multi-core processor and distributed computing, by using grid partitioning data into workers [10; 11], MapReduce technique [4; 7] or even taking advantage of graphics processing unit (GPU) [1; 12].

In this work, we discuss the use of parallelization in DBSCAN by implementing the algorithm presented by Sakai et al. (2016) and conducting a performance evaluation comparing to the traditional DBSCAN (without parallelization). The main goal of the assignment, as mentioned in the specification, is to develop the scientific method: reading scientific publications, replicating algorithm implementations, and conducting comparative experiments.

2. IMPLEMENTATION

2.1 Algorithm

The original DBSCAN method operates on data by processing each data point, determining whether it is either a core point, border point, or outlier, and assigning it to a particular spatial cluster. The main drawback of the original algorithm is its computational complexity, which is equal to $O(n^2)$ (without utilizing the index).

In the proposed algorithm [10], the enhancement in the performance is made by building a parallelization model to implement parallel processing of DBSCAN. The dataset is divided into several partitions, which are subsequently used as source data for the extraction of spatial clusters in a parallel manner. The fundamental idea of the new method lies behind partitioning the database into grids. Therefore, each data point is assigned to a specific grid, including the data. In order to conclude if the data that is located near the border is a core point or border point, in the algorithm, all grids extend extra border length of ε to make adjacent grids overlap and hence generate replications.

The overall performance is improved by utilizing complex grid partitioning, which reduces the number of data replications. The complex grid is composed by merging highly dense adjacent grids. This technique reduces the overall number of grids, therefore decreasing the number of replications and making the algorithm more efficient. The processing steps for constructing a complex grid consist of counting the size of each grid and recursively dividing the grid if its size is larger than the proportion of the total number of data points to the number of workers. Afterwards, a "dense" label is assigned to the grid if its size is larger than twice the average of its size; otherwise, the grid is labeled as "non-dense" that later on will be used to form a complex grid. Likewise, each dense grid is coupled with other adjacent dense grids, and the set of them forms a complex grid.

2.2 Application

Concerning the implementation of the proposed algorithm we have developed a Java program, that comprised of such packages: *model*, containing classes for custom data structures; *dbscan*, with the original and parallel DBSCAN implementations and; *test*, with a main method to run the experiment. With the purpose of imposing the abstract data types we constructed custom data structures for grids, complex grids, data points and grid limits, for instance.

The processing steps of the parallel DBSCAN implementation are as follows: we declare the parameters of the algorithm, such as ε, *minPoints*, number of divisions of the subspaces and number of workers, and then initialize executor for creating a task pool. After that, the grids were

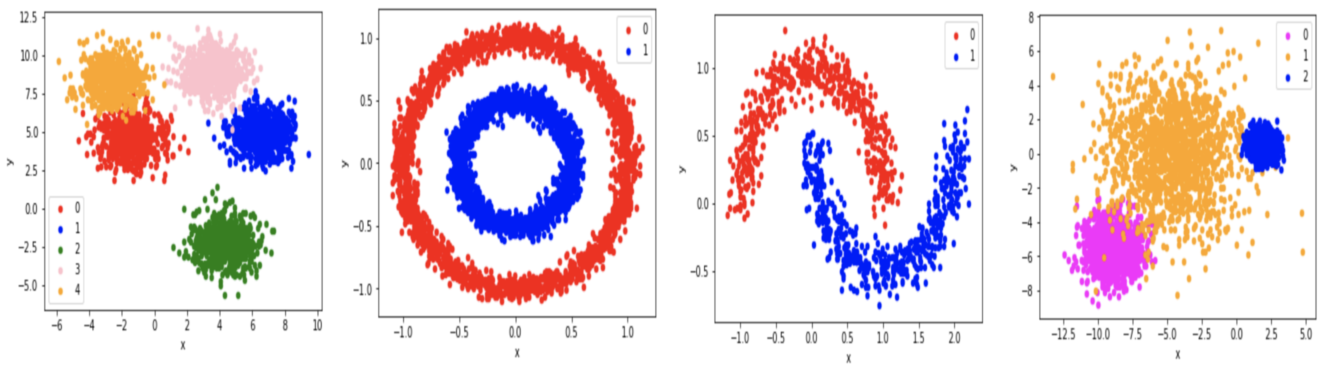

Figure 1: Distribution of each dataset: Blobs (first), Circles (second), Moons (third) and Varied data blobs (fourth)

generated.

For that, first the dimensions of each grid was calculated, based on the number of divisions of the subspaces and the dimension of the data set, including the extension of ε with a small precision correction. Given the coordinate limits of each grid, the points inside them were further included in order to run the clustering algorithm. This can be found on the *generateGrids* function. Subsequently, with the result of generated grids, complex grids were build by merging adjacent dense complex grid, given by the *generateComplexGrids* function.

Each partition in a complex grid is being put in the task pool and submitted to a new thread for execution. For that, a JAVA library called *java.util.concurrent* was used. After the completion of all the tasks, the result of applying DBSCAN to each grid is gathered into a list for the merging, and the task pool is destroyed.

For the merging part, the function *mergeClusters* checks if two clusters share at least one point and then merge them together, creating the final clusters.

2.3 How to run

To run the application it is necessary to:

1. Adjust the file *TestDBSCAN.java* in the *test* package with the desired parameters and input data file. *TestDBSCAN* class acts as *Main* class that contains *main* method
2. Compile the source code
3. Run the TestDBSCAN.java

3. PERFORMANCE EVALUATION

To compare the performance of the parallel version of DBSCAN in comparison to the simple one, an experiment was conducted with different data sets to see how the execution time changes varying the number of samples and number of threads. The experiment was performed on a MacBook Pro with 8 GB of RAM memory and a 2.4 GHz Intel Core i7 processor, with 4 cores. The experiment was run several times (around 3 to 10) to collect the average execution time for each configuration to avoid wrong conclusions caused by outliers. The number of subdivisions was empirically chosen as 4 since for larger values the clusters contained too much noise points (not clustered) and it took longer time for the machine used in this experiment.

3.1 Data Sets

To conduct an experiment for evaluating parallel processing for DBSCAN on a multi-core CPU, artificial data sets with random data were generated, comprising of different size and data distributions. We used data sets of four types: *Blobs*, *Moons*, *Circles* and *Varied data blobs*. Figure 1 shows the datasets. Each data set type was scaled in the range from 1000 to 25000 data. The reason behind generating sets of data with various distribution was to precisely assess the capability of our algorithm implementation to classify the data of different patterns. These types of data distributions are widely used by DBSCAN optimized implementations [4; 7; 10; 11], due to the fact that they include cases where density based approaches are superior towards other (e.g. K-Means). The benefit of artificial data consists of previously known information about the distribution (e.g. standart deviation, number of clusters) before running an experiment, which makes it easier to plot the clusters afterwards to assess the results accuracy. In order to build data sets we utilized Python library for Machine learning called *Scikit-Learn*[5] and *pandas*[2] library for constructing dataframes. Using such methods as *makeBlobs*, *makeMoons* and *makeCircles* of *Scikit-Learn* library every data set was generated with specified parameters for number of samples, number of clusters, dimensionality, percentage of noise, clusters standart deviations, etc. Jupyter notebook[3] *Data generation.ipynb* file was used to perform the data generation.

3.2 Setting the parameters

Setting the minimum number of points `minpoints` and the radius ε was done through performing several experiments and plotting the results. In the case of datasets with round shapes, we used the Silhouette [9] metric as a quality metric for our clusters. The Silhouette $s(i)$ of a cluster i checks the similarity of the points in a cluster $a(i)$ (minimizing the distance within the cluster) and the dissimilarity with the other clusters $b(i)$ (maximizing the distance from the other clusters). The formula of the silhouette is as follows:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} \quad (1)$$

The values that $s(i)$ takes are between -1 and 1. The best scenario would be to have $s(i) = 1$. $s(i) = -1$ means

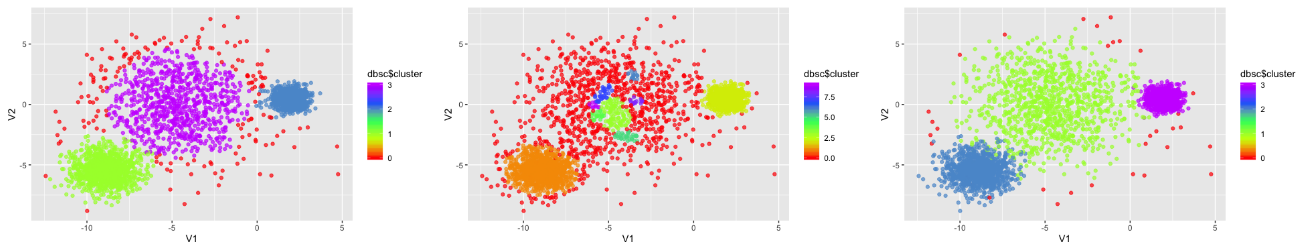

Figure 2: Same dataset with `minpoints`=40, ϵ=1.1 (first), `minpoints`=15, ϵ=0.4 (second), `minpoints`=5, ϵ=1 (third)

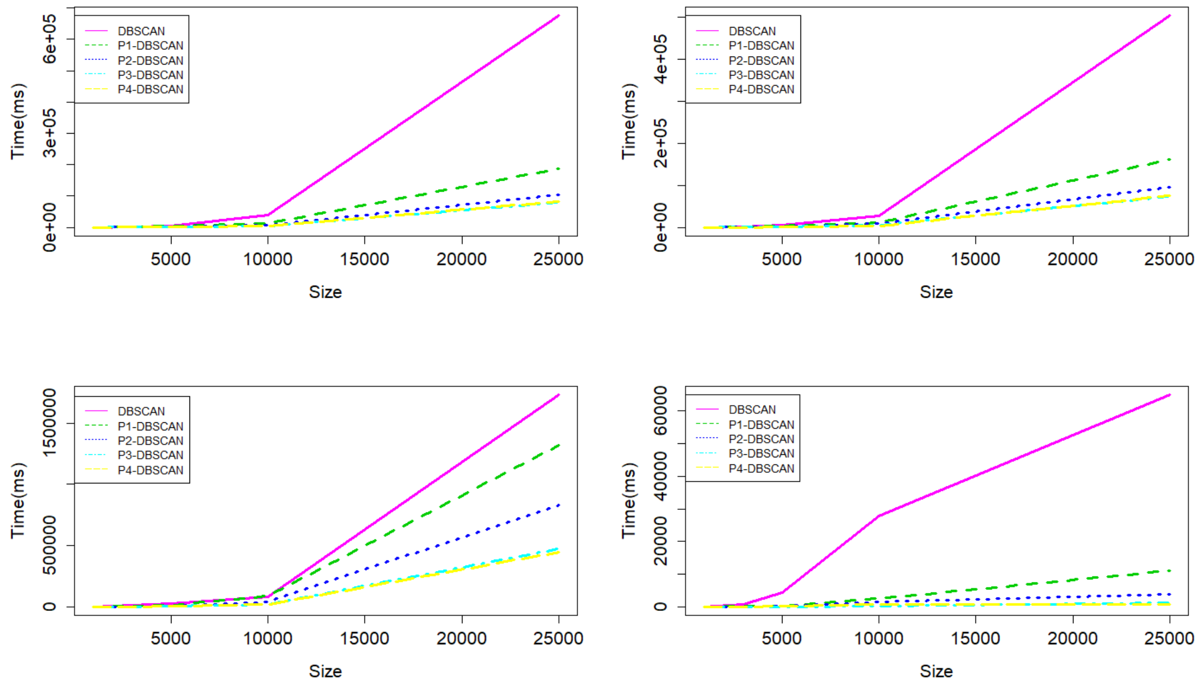

Figure 3: Time execution for DBSCAN, P1DBSCAN, P2DBSCAN, P3DBSCAN, and P4DBSCAN for circle (top-left), moon (top-right), blob (bottom-left), and varied data blobs (bottom-right) shaped datasets

that the points would be better clustered in the neighboring cluster, and $s(i) = 0$ means that the points are in the border between two clusters.

For the round-shaped datasets, we selected those ε values that maximize the mean of all clusters' silhouettes. Let us show an example of how we set the parameters. First, we select a `minpoints` value, e.g. 40 (Fig. 4). Then, for different values of ε, we checked the mean silhouette metric. Finally, we choose that ε that maximizes the silhouette. In Fig. 4, the best ε was 1.1, corresponding to a silhouette of 0.6.

DBSCAN is very sensitive to its parameters. Note that the clusters in Fig. 2 are very clearly identified with `minpoints`=40 and ε=1.1, but when we are more restrictive with `minpoints`=15 and ε=0.4, we get a lot of noise points and several small clusters within. On the contrary, for "loose" parameter values, `minpoints`=5 and ε=1, we tend to put further points in their neighbor clusters.

3.3 Results

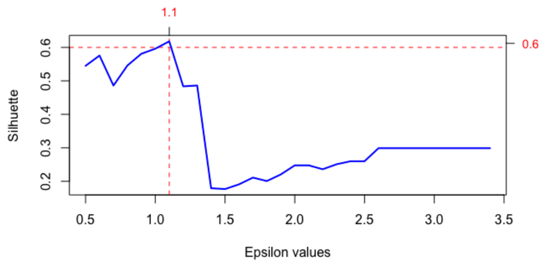

Figure 4: Silhouette for `minpoints`=40

For this experiment, the data is chosen with different types of distribution shapes, density, and size. For each dataset, the sequentially dbscan algorithm is run. Moreover, the parallelized optimized version was chosen to run among

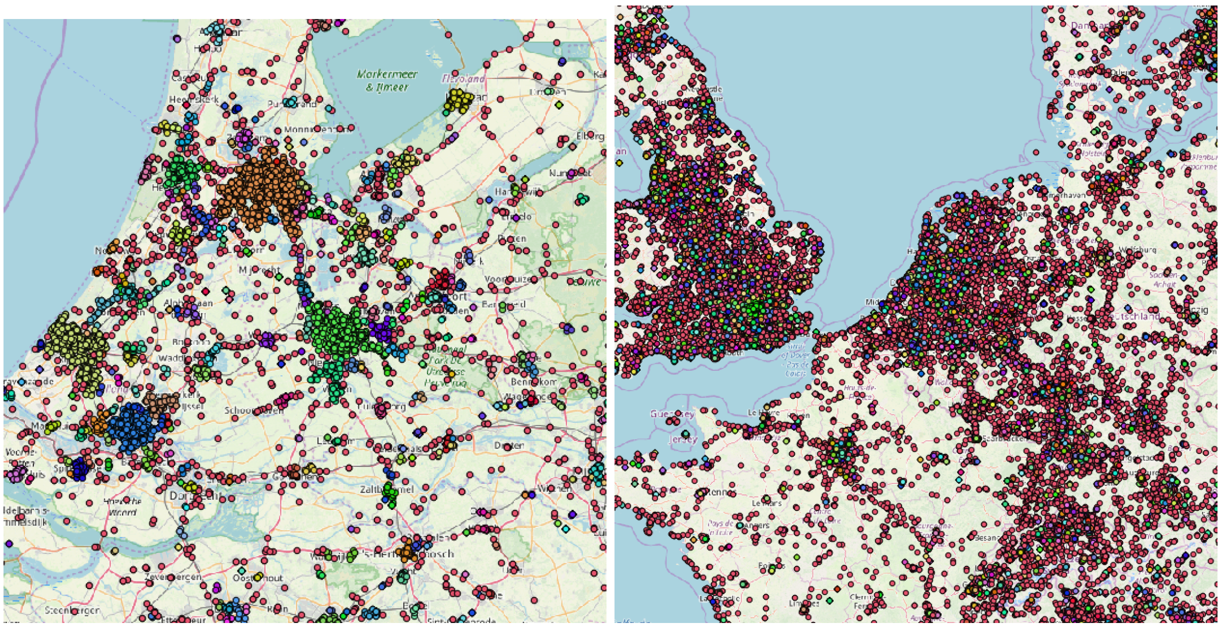
Figure 5: Clusters for the Brightkite checkins

different threads. DBSCAN represents the sequential one, P1DBSCAN the parallelized algorithm run with one thread, P2DBSCAN is the run with two threads, P3DBSCAN with 3 etc. Figure 4 represents the time of execution for each algorithm as the size of the dataset increases.

Figure 4 shows that there is a significant gain in the execution time using the parallelized DBSCAN with different threads, as the size of the data increases. However, if the size of the dataset is relatively small, this gain might not be perceived.

For circle shape distributed data, we can observe that the execution time for the sequential DBSCAN increases dramatically, while when using the parallelized DBSCAN, the clustering can be performed up to 9 times faster. For the moon shape, blobs, varied blobs distributed data, this ratio goes up to 9, 17, and 90, respectively. It was observed that for data relatively distributed, the parallelized algorithm performed much faster compared to the datasets that were really dense.

3.4 Real-world scenario

We tried using DBSCAN with a real-world dataset of Brightkite checkins[1] from all over the world. The dataset initially contained 4,491,143 checkins, but after data cleaning (checking for wrong values for the longitude and the latitude), there were 4,491,074 relevant points. These checkins originate from 58,228 users over the time period Apr. 2008 - Oct. 2010.

The checkins were clustered in 30,330 clusters, and 138,681 points were classified as noise points. Fig. 5 shows the distribution and the clusters. In the figure on the left, we can clearly notice the "hot" spots with the most checkins, while the red-colored points are the noise. From a zoomed-out view in the figure in the right, these checkins cover most of the Utrecht region.

[1]https://snap.stanford.edu/data/loc-Brightkite.html

4. CONCLUSION

The proposed algorithm [10] was used as an implementation of parallel processing of DBSCAN. The experimental results showed that parallelized approach outperforms the original DBSCAN method up to ninety times. However, with small and dense data sets it is laborious to assess the improvement of adopting a parallel approach regarding the obtained results. Moreover, for the small data sets the overhead of creating the data grids might degrade the results.

5. REFERENCES

[1] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha. G-dbscan: A gpu accelerated algorithm for density-based clustering. *Procedia Computer Science*, 18:369 – 378, 2013. 2013 International Conference on Computational Science.

[2] T. Augspurger, W. Ayd, and C. Bartak. Python data analysis library.

[3] D. Avila, M. Bussonnier, and S. Corlay. Project jupyter.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[5] J. V. den Bossche, L. Estève, and A. Gramfort. Scikit-learn: machine learning in python — scikit-learn 0.21.3 documentation.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters 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, pages 226–231. AAAI Press, 1996.

[7] Y. He, H. Tan, W. Luo, S. Feng, and J. Fan. Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, Feb 2014.

[8] K. Mahesh Kumar and A. Rama Mohan Reddy. A fast dbscan clustering algorithm by accelerating neighbor searching using groups method. *Pattern Recogn.*, 58(C):39–48, Oct. 2016.

[9] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[10] T. Sakai, K. Tamura, K. Misaki, and H. Kitakami. Parallel processing for density-based spatial clustering algorithm using complex grid partitioning and its performance evaluation. In *Proceedings of the 2016 International Conference on Parallel and Distributed Processing Techniques and Applications*, United States, 2016. CSREA Press.

[11] H. Song and J.-G. Lee. Rp-dbscan: A superfast parallel dbscan algorithm based on random partitioning. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1173–1187, New York, NY, USA, 2018. ACM.

[12] B. Welton, E. Samanas, and B. P. Miller. Mr. scan: Extreme scale density-based clustering using a tree-based network of gpgpu nodes. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2013.