DBSCAN for PyCOMPSs
A distributed approach

Carlos Segarra
carlos.segarra@bsc.es

Abstract—The abstract goes here.

Index Terms—Computer Society, IEEE, IEEEtran, journal, LaTeX, paper, template.

1 Introduction

This document covers the first implementation of the well-known clustering algorithm, DBSCAN, within the COMPSs framework for distributed computing. The first sections briefly introduce the algorithm and the programming model, followed by the implementation performance and its scalability when used in big clusters.

1.1 The algorithm: DBSCAN

The **D**ensity-**B**ased **S**patial **C**lustering **A**lgorithm with **N**oise (**DBSCAN**) is a clustering algorithm based on point density. It was proposed in 1996 [1] and has become one of the reference techniques for non-supervised learning. To take a first dive into the implementation, a few previous definitions are mandatory.

Definition 1. The **ε-neighborhood** of a point p from a dataset D and a given distance d is

$$N_\varepsilon(p) = \{q \in D : d(p,q) < \varepsilon\}$$

The general approach partitions the set of points in three subsets:

Definition 2.

- A point p is said to be ***core point*** if it has over *minPoints* neighbors within its ε-neighborhood.
- A point p is said to be ***reachable*** if it lies within a ε-neighborhood of a core point in spite of not being one.
- A point p is said to be ***noise*** if it does not fulfill any of the previous definitions.

Definition 3. A **Cluster** C is a subset of a dataset D such that for all $p \in C$:

(i) p is a core-point or reachable
(ii) $\forall p, q \in C, \exists p_1, \ldots, p_r$ fulfilling (i) and $p \in N_\varepsilon(p_1)$, $p_i \in N_\varepsilon(p_{i+1})$ and $p_r \in N_\varepsilon(q)$

Definition 4. The **distance between sets** A and B given a metric d is

$$d(A,B) = \min\{d(a,b) : a \in A, b \in B\}$$

In comparison to its main competitor, **k-means**, the DBSCAN is robust to outliers, it does not require an initial guess of the number of clusters and it is able to detect non-convex clusters as exposed in Figure 1 where the dataset is generated using the *Sklearn* package for datasets, the DBSCAN algorithm is the one presented in this document and the k-means is a personal implementation available on the project's GitHub repository.

1.1.1 S.o.A for distributed implementations

There are a variety of implementations of the DBSCAN raging from the more naive ones to more complex ones. As for comparison, a naive implementation following the exact guidelines of [1] can be found here. One of the most extended and used versions programmed in `Python` is the one by Sklearn.

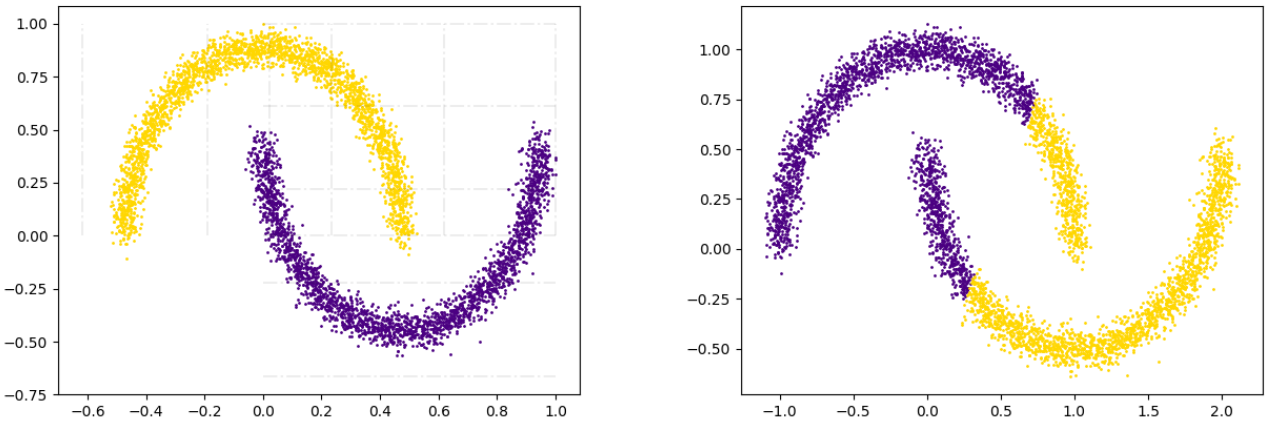
When it comes to distributed implementations of the algorithm, most of them can be summarized in applying an efficient DBSCAN to a chunk of the dataset and using some sort of synchronization or MapReduce. This is partially the approach taken in this project but the algorithm has been reinterpreted in distributed as a whole. See for instance this version, an implementation using Spark.

1.2 The framework: COMPSs

The COMPSs framework is a programming model designed to ease the development of applications for distributed architectures. The user programs a sequential code and defines the so-called `tasks`. The COMPSs runtime infers the dependencies and schedules the executions basing on the resources available. The model is developed in Java but has bindings for both `C` and `Python`. `PyCOMPSs` is the model chosen to develop the application. In order to mark a function as a task for the scheduler to take it into account, a small function decorator must be added. `COMPSs` is complemented by a set of tools for facilitating the monitoring and the post-mortem performance analysis (see Section 3).

1.3 The test zone: Mare Nostrum 4

The Mare Nostrum 4 is the fourth generetion of supercomputers located in the Barcelona Supercomputing Center - Centro Nacional de Supercomputacin (BSC-CNS) its current Linpack Rmax Performance is 6.2272 Petaflops and it has been running since early July 2017. It is equipped with



(a) Clusters found by DBSCAN.

(b) Clusters found by k-means.

Fig. 1. Comparison of the clusters provided by DBSCAN and k-means over set containing non-convex clusters.

48 racks with 3456 nodes. Each node has two Intel Xeon Platinum chips with 24 processors per unit.

2 Implementation Proposal

The algorithm implemented is a reinterpretation of the one exposed in [1], it is still though completely equivalent. The implementation can be found in this GitHub repository.

2.1 Step-by-Step analysis

Data Pre-Processing

Firstly, data is normalized by dividing each axis by the maximum value along it. This is done to prevent different factors from being really unbalanced and to ease the input parameters choice.

```
def normalizeData(dataFile):
    """
    Given a dataset, divide each dimension
    by its maximum
    :param dataFile: path to the original .
    txt
    :return newName: new name of the data
    file containing the normalized data.
    """
```

Adaptative Partitioning

Secondly, the dataset is partitioned depending on the point density. Initially the three partitioning points among each axis are the minimum value, the arithmetic mean and the maximum value. Iteratively, an additional point is added in the middle of the interval containing the higher amount of points. By performing this partition, excessive unbalancing of points per square can be avoided.

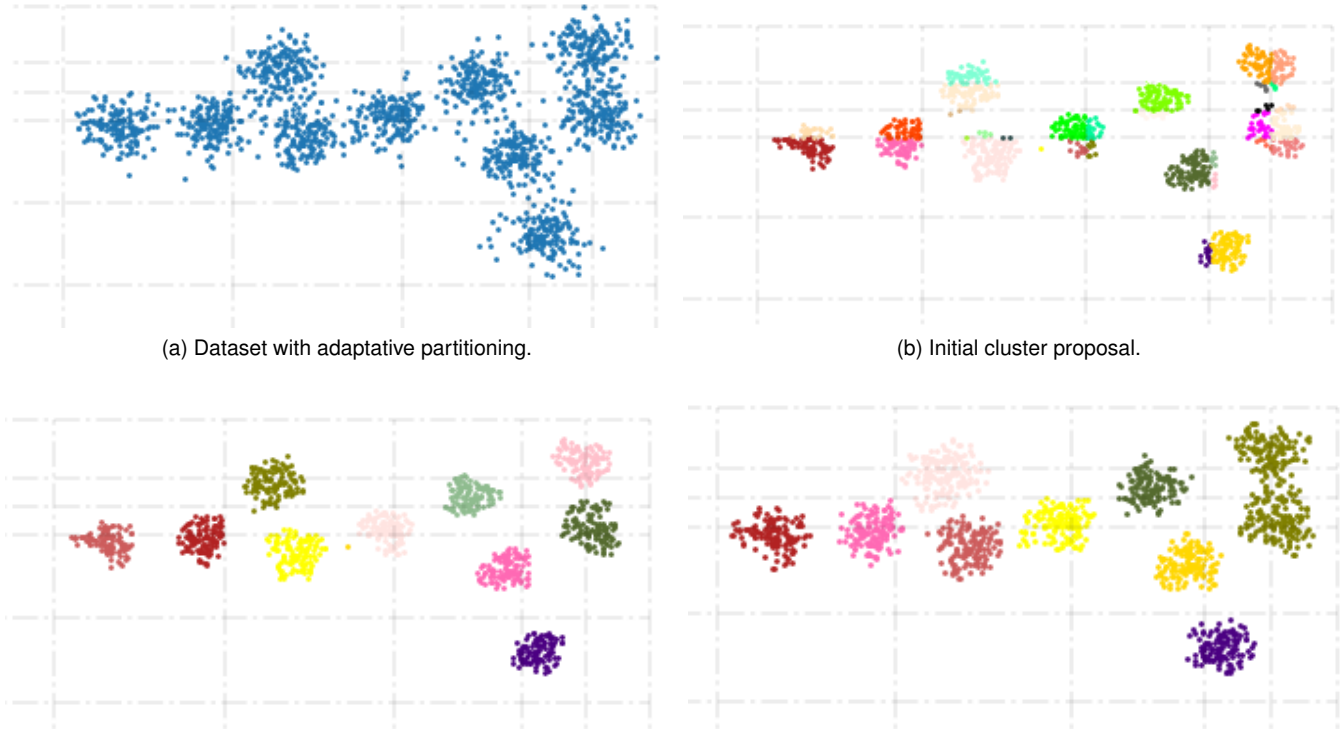
```
def partitionSpace(dataset, fragSize,
    epsilon):
    """
    Gives a space partition basing on point
    density
    :returns fragData:      dict with (key,
    value)=(spatial id, points in the region
    )
    :returns rangeToEps:    list with the #
    squares until epsilon distance
    """
```

Due to the theoretical possible large amount of points per cluster, an extra scale parameter is added (`numParts`), each worker will be assigned to a `numParts` part of each square resulting of the partition.

Initial Neighbor Retrieval

Once the dataset is correctly split, each worker performs a core point retrieval, please note *core points* retrieval. To optimize this query, each worker knows which square he has been assigned to and consequently only looks for possible neighbors in all the adjacent squares within ε distance. With this core points, a first cluster proposal is made (a cluster is a subset of the points assigned to that worker, however each worker has access to its neighbor squares points to determine whether a points neighborhood contains enough people). In the example of Figure 2, `numParts` has been fixed to 1 so that each square is assigned to one and only one worker, otherwise intermediate results would be difficult to visually understand.

```
def partialScan(corePoints, square, epsilon,
     minPoints, fragData, fragSize, numParts
    , rangeToEps):
    """
    Looks for all the core points (over
    minPoints neighbours) inside a certain
    square.
    :inout corePoints:  list were core
    points found are appended.
    :param square:      space region where
    core points are looked for.
    :param fragData:    dict containing
    space partition and its points.
    :param rangeToEps:  for each square,
    number of neighbors until epsilon
    distance.
    """
```



(a) Dataset with adaptative partitioning.

(b) Initial cluster proposal.

(c) Clusters after synchronization.

(d) Final output.

Fig. 2. Plotting of the current result at each step of the algorithm.

Cluster Synchronization

As a result of 2.1, a huge list of clusters is obtained. To sync the corresponding results the clusters are reinterpreted as nodes in a graph and an adjacency matrix is built. The (i, j)-th element in the matrix will be true if and only if the distance between the i-th and the j-th cluster is lower than ε, see Definition 4. Equivalently, each synced cluster will correspond to a connected component of the graph determined by the adjacency matrix (here a synced cluster refers to a cluster post-synchronization). Using a merge-find set, this dependencies are quickly sorted and the connected components easily found. Since the initial number of clusters found depends on the amount of partition points chosen and on the data distribution, an extra scale parameter is introduced to prevent the algorithm of scheduling too many tasks (one per comparison). This scale parameter, $numComp$ determines how many cluster vs cluster comparisons a single task performs.

```
def syncClusters(clusters, epsilon, numParts):
    """
    Returns a matrix of booleans. Pos [i,j]=1 <=> clusters -i and -j should be merged.
    :inout clusters:            list of all clusters and their points.
    :param numComp:             number of comparisons per worker (i.e tasks).
    :return possibleClusters:   adjacency matrix.
    """
```

Cluster Expansion

Once all the information is updated, the current result is a set of clusters of core-points.A second neighbor retrieval is performed, assigning a part of each cluster to a worker, to check for reachable points and detect possible cluster merging. Finally results are exported to a text file and, if selected, clusters are plotted and the plot saved to file.

```
def expandCluster(clusters, fragData, epsilon, minPoints, fragSize, numParts, rangeToEps):
    """
    Expands all clusters contained in a list of clusters.
    already established clusters.
    """
```

The whole process is summarized in Figure 2.

2.2 Dependency Graph

The COMPSs framework introduced in Section 1.2 incorporates a dependencies graph generating script, to improve the comprehension on the application behavior.

In Figure 3 two main bottlenecks in the execution can be observed (besides the last synchronization which is unavoidable). Relating to the step by step description, the first one corresponds to the initial neighbor retrieval and the first cluster proposal. One might think that this step might not need a synchronization point since the cluster vs cluster comparison does not require all the clusters to be found, just the two required to be compared. Even though this might sound true in theory, the fact that the points from a square need to be sub-partitioned again for size issues and the fact that in order to build the adjacency matrix one

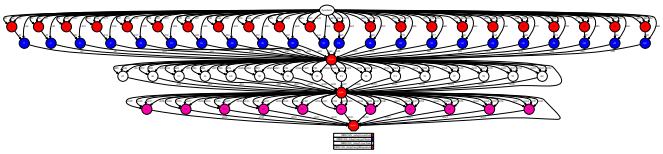


Fig. 3. Dependency graph for a small execution.

must know the number of nodes his graph will have make removing this synchronization point nearly impossible (at least with this implementation). However there are as well possible workarounds. For instance, a `MAX_CLUSTS` variable could be defined and all the sizes prefixed. This goes however against one of the DBSCAN's main principle, that no information about the number of clusters is required to begin the execution. Further development on improvement proposals is exposed in Section 4.1. The second bottleneck comes after the initial cluster proposal is synced. Once we know for each cluster which cluster it can be merged too, it is mandatory to wait for all the information processed since some information might be lost and second and most important, adding points to clusters back and forth can be much more time consuming than waiting for the adjacency matrix to be fully computed and appending points only once. Both workarounds have been implemented, proving to behave worse.

2.3 About equivalence

Basing on the definitions given in Section 1.1 and the implementation exposed in Section 2.1. It is pretty clear that the algorithm performs completely equivalently to the one stated in [1]. Locally at worker level, the algorithm is exactly the same, the only difference is that only a chunk of the whole dataset is used. When syncing two different situations might appear.

- **Clusters in different squares.** This is one of the most common situations. A natural cluster divided by the space partition performed at 2.1. If both clusters should really be merged, then at least one point from each cluster must be at a distance lower than ε and if so, the syncing process will merge them. This situation is illustrated in Figure 4. Bear in mind though that after 2.1 only clusters of core points are found. What if the linking point between two clusters is not a core point?
- **Clusters related by a non-core point.** According to the reference paper, if two point-dense regions lie within a ε-neighborhood of the same point, even if this point might only have two neighbors, then this two clusters must be merged. In this implementation this situation is particularly fragile since initially only core points are retrieved. This dependencies are taken care off in 2.1. This situation is illustrated in Figure 5.

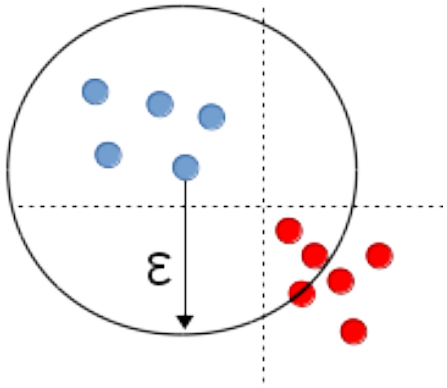


Fig. 4. A cluster divided by the chunking process.

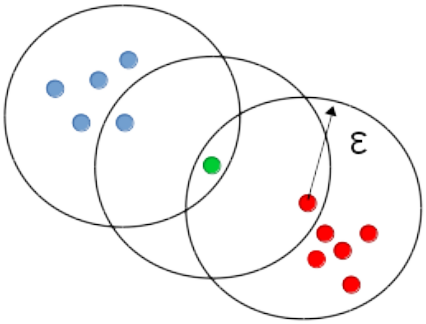


Fig. 5. Cluster linked by a non-core point.

3 Performance

3.1 Runtime Metrics

To measure the runtime, for different dataset sizes (all three dimensional but with different number of points) the algorithm's execution time has been measured. The time measurement is performed inside the python method using the built-in tools and printed through the standard output. For the same dataset, a batch of executions with different number of workers has been performed to test the algorithm scalability. The input parameters have been modified with different data sizes but never inside the same batch of executions. Lastly all the executions have been submitted in MN4 (1.3) with Scratch file system. The results are summarized in Table 1

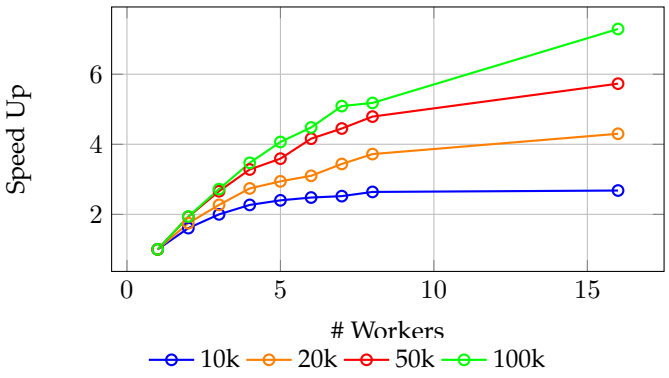


Fig. 6. Speed Up as a function of the number of workers for different datasets.

In figure 6 the speed up vs number of workers is plotted. To understand why it increases as the number of points

TABLE 1
Performance measured on MN4 without tracing and `fragSize`=8 and 16 CPUs per node.

SCRATCH Workers	NP=16 10k	numComp=10 Speed Up	NP = 16 20k	numComp=10 Speed Up	NP = 32 50k	numComp=10 Speed Up	NP = 32 100k	numComp=10 Speed Up
1	36.1205	1	125.034	1	820.14	1	3160.46	1
2	22.47	1.63	71.66	1.74	423.36	1.93	1623.92	1.94
3	18.05	2.02	55.043	2.27	308.014	2.62	1165.101	2.71
4	15.916	2.25	45.609	2.74	249.59	3.28	910.3	3.47
5	15.01	2.40	42.497	2.94	227.93	3.59	777.195	4.06
6	14.55	2.48	40.23	3.16	196.69	4.16	703.9	4.48
7	14.3	2.52	36.364	3.43	184	4.45	620.531	5.09
8	13.644	2.64	33.611	3.72	171	4.79	609.211	5.18
16	13.45	2.68	29.025	4.30	143	5.73	433.126	7.29
Seq	4690.8		—		—		—	

increases check section 3.2. In addition to that, a runtime vs number of workers plot can be found at 7.

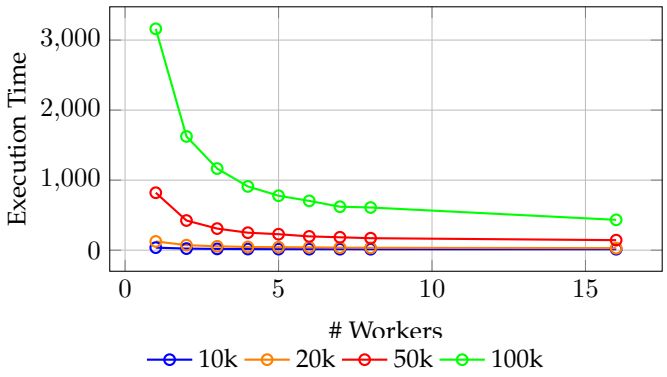


Fig. 7. Execution time as a function of the number of workers for different datasets.

3.2 Critical Analysis

Dependence on the scale parameters:

From the explanation in Section 1.1 it is easy to see that the paradigm has shifted from a two parameter algorithm to a four one. In the one hand this enables, when run with knowledge, the algorithm to adapt better to mutable resources and data loads. In the other hand this also increases the variability of the performance and may be a drawback at some points.

Emphasizing on this two extra parameters, one of them has not got a clear impact on the runtime metrics. The number of comparisons executed by a worker, `numComp` will determine a certain type of task's length but won't increase their variability. However, `numParts` chunks the data again having a direct effect on the number of clusters initially found and consequently on the number of tasks performed at the syncing stage. In addition to that, it reduces the average task duration but at the same time makes tasks less homogeneous. This process causes bigger executions (more points and assigned nodes) to behave less homogeneously. In spite of that, this scale parameter is necessary to ensure the algorithm scalability, otherwise this would depend entirely on the data distribution.

To dig in this sections where the algorithm is not behaving as expected, or at least not as desired, a performance analysis using tracing (Paraver) is performed. In

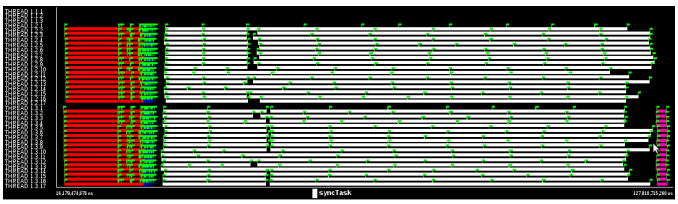


Fig. 8. Tracing with 10k points and `numParts` $= 16$.

Figure 8 there is an example of a small execution with `numParts` $= 16$ and 10000 points. Consequently with the above lines, a homogeneous trace is obtained. In Figure 9 the opposite behavior is exposed. For this second execution the number of points was around 20000 and `numParts` $= 32$. It is easy to see that tasks duration becomes much more volatile (green flags indicate a task' beginning and ending) and they are not so easily scheduled.

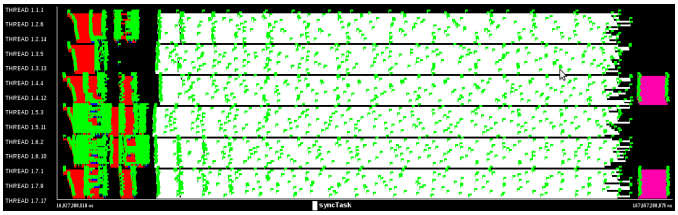


Fig. 9. Tracing with 20k points and `numParts` $= 32$.

On memory, CPUs per node and scalability:

Another issue faced when testing the algorithm is the memory vs cpu per node paradigm. Initially each node at 1.3 has 2GB of memory , however if the user requests for less workers and exclusivity each node will have more memory since the whole batch is divided between the running cores. When the dataset tends to be bigger and bigger, the memory requested is bigger as well. Hence the performance at some points might be better with less `cpus_per_node` in spite of the time lost with inter-node communication.

4 Conclusion

Following along the reasoning from 3.2, the algorithm is really sensitive to changes in the input parameters. Thus this parameters should somehow depend on the resources available, however that would make the algorithm not

structure-unaware. This might be one of the reasons why scalability and speed up is not exactly as desired since all the tests for the same dataset are performed under the same input parameters (to be able to compare them). Therefore the theoretical speedup might decrease in comparison to if we were to run the algorithm manually test by test.

4.1 Further Development and Improvement Proposals

The main development I would recommend implementing would be a smart guessing at execution time of the scale parameters. Empirically it has been proven that if the user is able to choose the right parameters, the algorithm is going to have a great performance. As a consequence, being able to guess them at execution time would guarantee a stability that for this moment I can not ensure.

Secondly, the DBSCAN method could be expanded to a generalized DBSCAN to detect arbitrary cluster densities removing the dependencies to the input parameters, `minPoints` and ε.

Lastly and in the opposite direction of the second proposal, some research into parallel optimization models so to guess the input parameters (rather than using a thumb rule) could be done.

REFERENCES

[1] Ester, Martin Kriegel, Hans-Peter Sander, Jorg Xu, Xiaowei (1996). *A density-based algorithm for discovering clusters in large spatial databases with noise.* Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)