DBSCAN for PyCOMPSi
A distributed approach

Carlos Segarra
carlos.segarra@bsc.es

Abstract—The DBSCAN algorithm is one of the most popular clustering techniques used nowadays. However there is a lack of distributed implementations. Additionally, parallel implementations fail when they try to scale to hundreds of cores. The scope of this report is to provide a novel implementation of the algorithm that behaves well in large distributed architectures and with big datasets. Using the PyCOMPSs framework and testing on the Mare Nostrum 4 supercomputer, encouraging results have been obtained reaching a 2467 speedup when run with 4096 cores. Everything whilst keeping the code clean and transparent to the user.

Index Terms—DBSCAN, Clustering, Machine Learning, Distributed Computing, COMPSs, PyCOMPSs

1 Introduction

Big data and data mining is on the daily agenda of nowadays' engineers and having tools to process and analyze this data quickly is of vital importance. Precisely, machine learning, and more precisely, unsupervised learning and clustering are useful for detecting trends and patterns between data without needing the user to previously classify it. The DBSCAN belongs to this family of methods and is useful for detecting non-convex (arbitrarily-shaped) clusters as explained in Section 3.

This document covers the first implementation of the DBSCAN clustering algorithm within the COMPSs [1] framework for distributed computing. The main scope of this paper is to accomplish a reasonable speedup when scaling to thousands of cores, being able to process datasets of hundreds of thousands or millions of points in a reasonable amount of time.

Our main contribution is a simple algorithm, completely sequential at first glance, that a standard programmer without much knowledge on concurrency could maintain and develop, programmed in Python (which enhances even more the readability) that scales to thousands of nodes and performs well on big clusters.

The structure of the paper is the following. Section 2 covers related work, other attempts to implementing a parallel version of the DBSCAN in other programming frameworks as well as other clustering algorithms. It briefly covers different benchmarks related to performance and emphasizes the novelty of our proposal. Following along, Section 3 introduces the state of the art of clustering algorithms, describes the Density-Based Spatial Clustering Algorithm the programming model chosen for the implementation and covers the environment where tests have been performed. Section 4 describes the algorithm developed to fulfill the scope of the paper, the dependency graph drawn by it and covers possible equivalence issues faced when reinterpreting a sequential algorithm as a parallel one. Section 5 summarizes all the tests performed with synthetic and real data, presents the results and gives a critical review to the algorithm using post-mortem analysis tools and tracing. Lastly Section 6 gathers the conclusions obtained and proposes further developments that did not fit our initial scope.

2 Related Work

Quickly after the algorithm was first presented (1996) the necessity to feed the clustering algorithm with big datasets arose. First approaches of DBSCAN parallelization only dealt with the region query issue which is indeed where more calculus power is invested. For instance, [2] presents a skeleton-structured program where slaves are responsible for answering the regions query performed by the master.

Latter developments started including smart data partitioning. [3] proposes what has become a standard procedure for distributed implementations of the DBSCAN. Initially data is efficiently divided, secondly a series of partial clusterings are performed and final results are merged by the master. Most implementations differ on both the partition and the merging. [3] divides data using a low-precision hierarchical clustering algorithm and merges the data using a graph connectivity technique similar to the one used in this paper. Graph connectivity techniques are also used in [5]. Yaobin HE et. al [4] propose a novel partitioning method based on computational costs predictions and merge results using a 4 step MapReduce strategy.

Latest developments focus on flexibility and scalability to high feature datasets. A. Lulli [6] presents a Spark-based algorithm that does not require euclidean metric and that works with arbitrary data.

The novelty of our proposal lies that in all the previously mentioned papers, the code might be quite difficult to understand and specially to mantain for a user. Taking advantage of the COMPSs framework for distributed programming, the final code produced is highly-readable.

3 Background

This section provides the necessary information to firstly understand the implementation proposed and secondly to contextualize the tests performed.

3.1 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 [7] 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 given a 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 fulfil 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) such that $p \in N_\varepsilon(p_1)$, $p_i \in N_\varepsilon(p_{i+1})$ and $p_r \in N_\varepsilon(q)$ where only, maybe p and/or q are reachable

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

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

3.1.1 Other Clustering Algorithms

Recently a lot of research has been dedicated to improving clustering algorithms, resulting in the current classification in four different categories: hierarchical clustering, centroid-based clustering, distribution-based clustering and density based clustering.

In comparison to its main competitor the **k-means** algorithm [8], 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. To obtain the Figure the dataset has been generated using the *Sklearn* package for datasets (`make_moons`), 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.

3.1.2 S.o.A for sequential and 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 [7] 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. Otherwise, the DBSCAN for PyCOMPSs reformulates the algorithm trying to adapt it to distributed architectures making sure it is still equivalent to the original implementation.

3.2 The framework: COMPSs

The COMPSs framework [1] 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` [9] is the model chosen to develop the application. A master orchestrates a series of workes with a certain number of threads that are responsible of running tasks. In order to mark a function as a task for the scheduler to take it into account, a small decorator must be added. `COMPSs` is complemented by a set of tools for facilitating the monitoring and the post-mortem performance analysis (see Section **??**).

4 Implementation Proposal

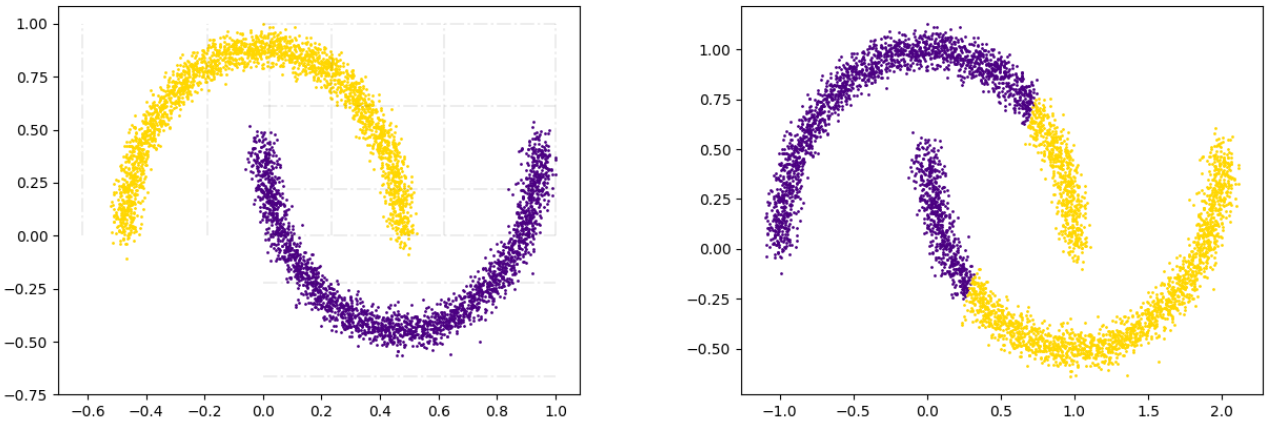
The algorithm implemented is a reinterpretation of the one exposed in [7], it is still though completely equivalent. The implementation can be found in this GitHub repository. The main algorithm would be structured as in Algorithm 1 (note that all the methods stated are later expanded and explained). It takes as an input the two required parameters: ε and `min_points` as well as the path to a distirbuted dataset stored in some sort of database[1]

Algorithm 1 Main method for the DBSCAN algorithm.

```
1:  function DBSCAN(data_file, epsilon, min_points, frag_size)
2:      norm_data ← normalize_data(data_file)
3:      frag_data ← partition_space(norm_data, frag_size)
4:      for all square in frag_data do
5:          core_points ← partial_scan(square, epsilon,
                min_points, frag_data)
            clusters[square] ← merge_cluster(core_points,
                epsilon)
6:      end for
7:      clusters = compss_wait_on(clusters)
8:      clusters_links = sync_clusters(clusters)
9:      clusters = update(clusters, clusters_links)
10:     returnexpand_clusters(clusters)
11: end function
```

where the `update` method finds the connected components of a graph given its adjacency matrix (using an MF-Set) and merges the corresponding clusters.

1. For the speciphic implementation given, the structure assumed is that of the GPFS in 5.1 and points are assumed to be divided in a grid. Otherwise a linear classificlation should be incoporated in the pre-processing.



(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.

4.1 Step-by-Step analysis

4.1.1 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.

Algorithm 2 Given a dataset, divide each dimension by its maximum value

```
function NORMALIZE_DATA(data_file)
    data ← load(data_file)
    for all dimensions of data do
        Divide by th maximum along its axis
    end for
    return save(data)
end function
```

4.1.2 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, these values are stored in `fragVec`. Iteratively, an additional point is added in the middle of the interval containing the higher amount of points, this is done `fragSize` times along each dimension. By performing this partition, excessive unbalancing of points per square can be avoided. Once this step is finished, the dataset is stored in a dictionary where the key is a unique identifier of each spatial region and the value is a list containing all the points in that region.

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

4.1.3 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. The query structure is the one presented in Figure 2. All the points from the squares reachable with ε distance from the square where the point is are considered as possible neighbors. If a point has over `min_points` neighbors then it is considered a core point.

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 point neighborhood contains enough people). In the example of Figure 3, `num_parts` 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.

Algorithm 3 Return a space partition basing on point density

```
function PARTITION_SPACE(dataset, frag_size, epsilon)
    Set dimension, fragVec
    for all dimensions of data do
        for i ← 1, frag_size do
            for all p in dataset do
                Find most dense interval.
            end for
            Divide that interval in two.
        end for
    end for
    return save(data)
end function
```

4.1.4 Cluster Synchronization

As a result of 4.1.3, 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

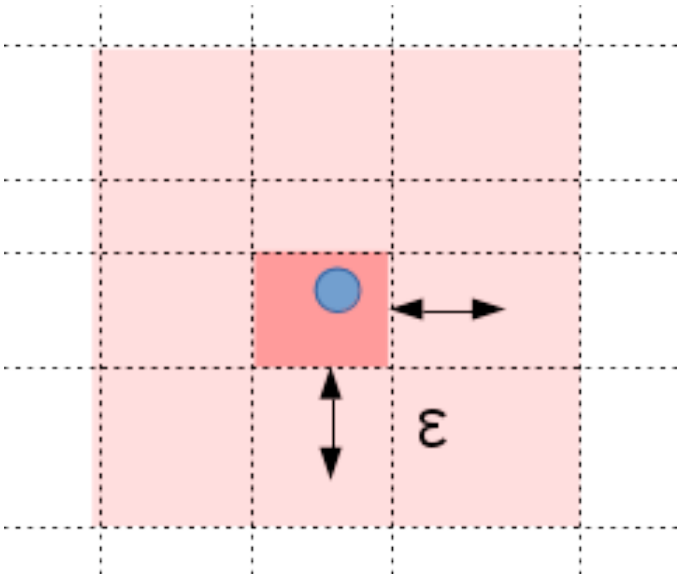


Fig. 2. In darker red the square where the point belongs, in lighter red the squares where the algorithm will look for possible neighbors.

Algorithm 4 Looks for all the core points in a certain square

```
1: @task()
2: function PARTIAL_SCAN(square, epsilon, min_points)
3:         frag_data)
4:     point_set_real ← get_data(square)
5:     point_set ← get_data(get_neighbors(square))
6:     for all point in point_set_real do
7:         for all pos_neigh in point_set do
8:             if distance(point, pos_neigh) ≤ epsilon then
9:                 point.neighbors += 1
10:            end if
11:        end for
12:        if point.neighbors ≥ min_points then
13:            core_points.add(point)
14:        end if
15:    end for
16:    returns core_points
17: end function
```

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.

4.1.5 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 add them non-deterministically. Finally results are exported to a text file

Algorithm 5 Returns an adjacency matrix

```
1: function SYNC_CLUSTERS(clusters)
2:     for all clust1 in clusters do
3:         for all clust2 in clusters do
4:             @task()
5:             if distance(clust1, clust2) ≤ epsilon then
6:                 adjMatrix[clust1.num, clust2.num] ← 1
7:             end if
8:         end for
9:     end for
10:    return adjMatrix
11: end function
```

and, if selected, clusters are plotted and the plot saved to file.

Algorithm 6 Expands all clusters from the list

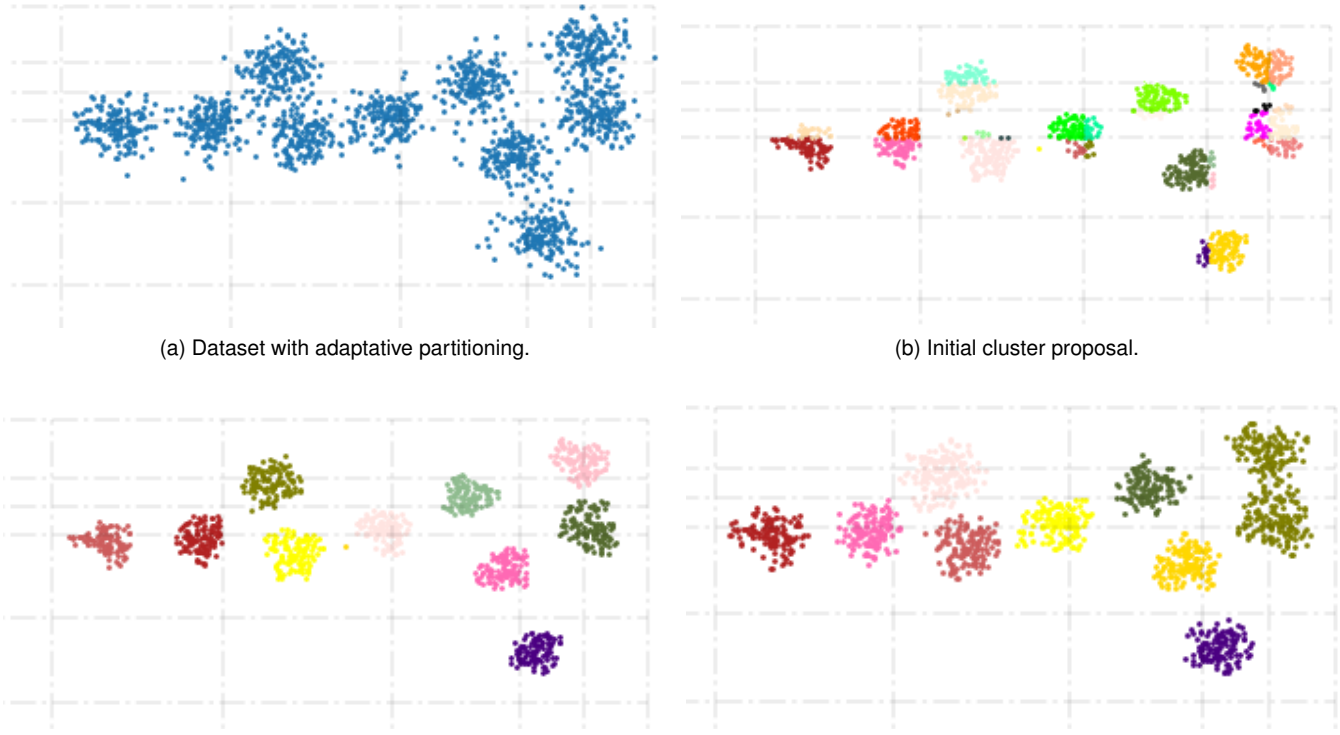
```
1: function EXPAND_CLUSTERS(clusters)
2:     for all clust in clusters do
3:         tmp_points ← get_neighbors(clust.square)
4:         poss_neigh ← tmp_points not in clust.points
5:         for all point in poss_neigh do
6:             if distance(point, clust ≤ epsilon) then
7:                 clust.add(point)
8:             end if
9:         end for
10:    end for
11: end function
```

The whole process is summarized in Figure 3.

4.2 Dependency Graph

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

In Figure 4 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 does 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 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 6.1. The second bottleneck comes after the initial cluster proposal is synced. Once we know for each cluster which clusters it can be merged too, it is mandatory to wait for all the information processed since otherwise a lot of time could be lost in moving points from a cluster



(a) Dataset with adaptative partitioning.

(b) Initial cluster proposal.

(c) Clusters after synchronization.

(d) Final output.

Fig. 3. Plotting of the current result at each step of the algorithm.

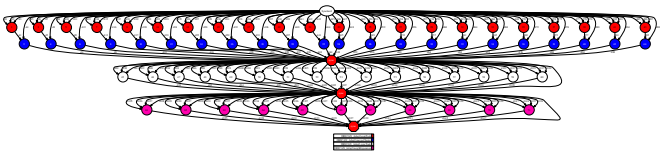


Fig. 4. Dependency graph for a small execution.

to another. For instance, if cluster 1 must be merged with cluster 2 and cluster 2 with clusters 1 and 3, not waiting for all the adjacency matrix to be computed would represent copying all the data twice with the corresponding loss in performance.

4.3 About equivalence

Basing on the definitions given in Section 3.1 and the implementation exposed in Section 4.1. It is pretty clear that the algorithm performs completely equivalently to the one stated in [7]. The neighbor retrieval is the same as in the reference implementation. When syncing two different situations may appear.

- **Clusters in different squares.** This is one of the most common situations. A natural cluster divided by the space partition performed at 4.1.2. 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 5. Bear in mind though that after 4.1.3 only clusters of core points are found. What if the linking point between two clusters is not a core point?

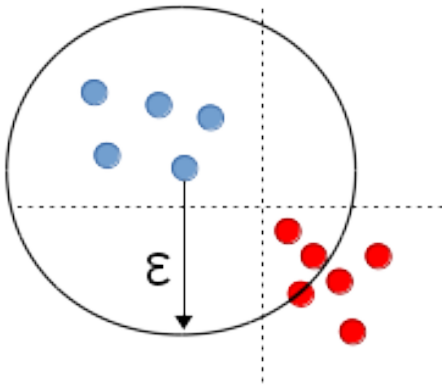


Fig. 5. A cluster divided by the chunking process.

- **Clusters that share 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 must be non-deterministically assigned to one of the two clusters. This does not represent a problem to the initial algorithm since its sequentiality prevents it from needing to take a decision. In 4.1.5 at the Cluster Expansion Stage the same point will be assigned to both clusters but only assigned to the first of them in the cluster list. This situation is illustrated in Figure 6.

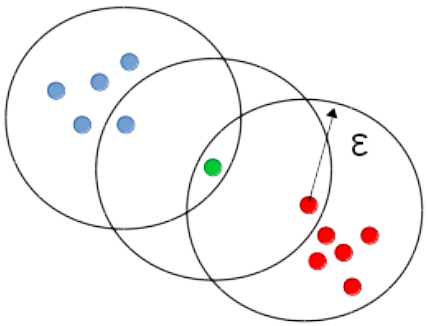


Fig. 6. Cluster linked by a non-core point.

5 Evaluation

This section covers the evaluation of the implementation presented. The first subsection 5.1 covers how was the algorithm implemented and tested. The second subsection 5.2 gathers all the data collected from the different experiments performed. The last subsection 5.3 provides a critical review on the results, what was expected and what has been achieved.

5.1 Infrastructure

As mentioned before, the algorithm has been implemented fully in Python within the PyCOMPSs framework. COMPSs version is the 2.1 and Python's is the 2.7.13 since COMPSs does not support Python 3.X. To run the program hence, the user must have COMPSs installed, the corresponding Python version and the numeric python library (numpy). The algorithm can be ran both locally and in a cluster, in the project's GitHub repository there is a shell script (`run.sh`) that contains methods to invoke the DBSCAN both as local host (through `runcompss` command) and to a queuing system (through `enqueue_compss` command).

The experiments were ran on a cluster located in Barcelona. The Mare Nostrum 4 (MN4) is the fourth generetion of supercomputers located in the Barcelona Supercomputing Center - Centro Nacional de Supercomputacion (BSC-CNS) its current Linpack Rmax Performance is 6.2272 Petaflops and it has been running since early July 2017. It is equipped with 48 racks with 3456 nodes. Each node has two Intel Xeon Platinum chips with 24 processors per unit. COMPSs through its queuing script for clusters, `enqueue_compss`, requires each node on exclusivity. Therefore from now on, whenever a node is mentioned it must be considered that no other program is going to be running there besides our execution. Additionally the queuing script incorporates a `cpus_per_node` flag that determines the number of processors running in each node.

5.2 Runtime Metrics

To evaluate the algorithm's performance for different dataset sizes (all three dimensional but with different number of points) the execution time has been measured. This duration is computed 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 was performed to test the algorithm's scalability. The input parameters vary between different data sizes but never inside the same batch. All the executions were ran in MN4 (**??**) with Scratch file system. Results are summarized in Table 1

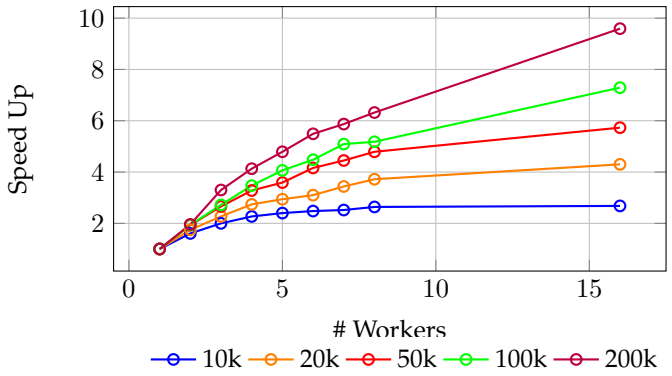


Fig. 7. Speed Up as a function of the number of workers for different datasets.

In figure 7 there is a speed up vs number of workers plot. To understand why it increases as the number of points increases check section 5.3.2. Additionally, a runtime vs number of workers plot can be found at 8.

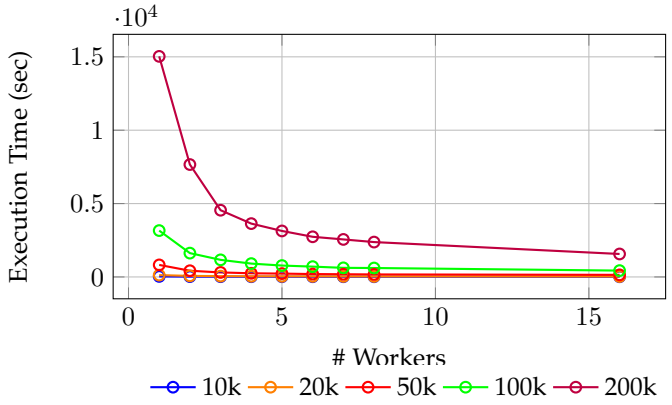


Fig. 8. Execution time as a function of the number of workers for different datasets.

5.3 Critical Analysis

5.3.1 *Dependence on the scale parameters:*

From the explanation in Section 3.1 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 by someone familiar to it, the algorithm to adapt better to mutable resources and data loads. In the other hand this also increases the variability of the performance, may be a drawback at some points and makes the implementation not 100% structure-unaware.

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

TABLE 1
Performance measured on MN4 without tracing and 16 CPUs per node.

SCRATCH	NP=16	10k	NP = 16	20k	NP = 32	50k	NP = 32	100k	NP=32	200k
Workers	Time (sec)	SUp	Time (sec)	SUp	Time (sec)	SUp	Time (sec)	SUp	Time (sec)	SUp
1	36.12	1.00	125.03	1.00	820.14	1.00	3160.46	1.00	15030.99	1.00
2	22.47	1.63	71.66	1.74	423.36	1.93	1623.92	1.94	7659.83	1.96
3	18.05	2.02	55.04	2.27	308.014	2.62	1165.10	2.71	4547.23	3.3
4	15.91	2.25	45.60	2.74	249.59	3.28	910.30	3.47	3634.97	4.13
5	15.01	2.40	42.49	2.94	227.93	3.59	777.19	4.06	3134.47	4.79
6	14.55	2.48	40.23	3.16	196.69	4.16	703.90	4.48	2736.65	5.49
7	14.3	2.52	36.36	3.43	184.00	4.45	620.53	5.09	2557.32	5.87
8	13.64	2.64	33.61	3.72	171.00	4.79	609.21	5.18	2376.56	6.32
16	13.45	2.68	29.02	4.30	143.00	5.73	433.12	7.29	1565.82	9.59
Seq	4690.80		—		—		—		—	

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 and at the same time makes tasks less homogeneous. This process causes bigger executions' (more points and assigned nodes) behavior more volatile. In spite of that, this scale parameter is necessary to ensure the algorithm's scalability, otherwise it would depend entirely on the data distribution.

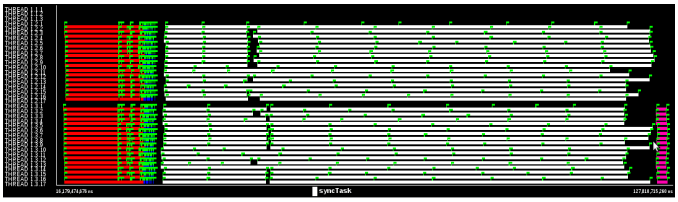


Fig. 9. Tracing with 10k points and `numParts` = 16.

To dig in this sections where the algorithm is not performing as expected, or at least not as desired, we undergo a performance analysis using tracing (Paraver). In Figure 9 there is an example of a small execution with `numParts` = 16 (relatively small value of the parameter) and 10000 points. Consequently with the above lines a homogeneous trace is obtained. In Figure 10 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 durations become much more unstable (green flags indicate a task's beginning and ending) and they are not so easily scheduled by the COMPSs runtime.

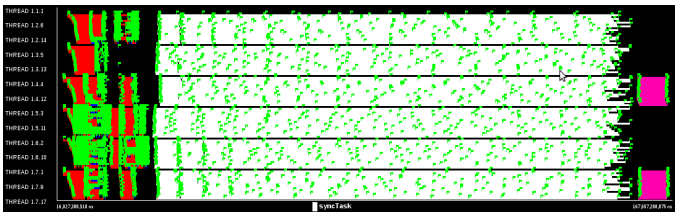


Fig. 10. Tracing with 20k points and `numParts` = 32.

5.3.2 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 has 2GB of RAM, however if the user requests less CPUs per node and exclusivity each worker will virtually have more random access memory since the whole batch is divided between the running threads. When the dataset tends to get bigger and bigger, the memory requested increases 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. To dig deeper into this situation, another set of tests is performed. Results are summarized in Table 2. The `cpus_per_node` values to be compared are 16 and 48. 48 is 16 times 3, therefore to perform a fair comparison the number of workers assigned to the 16 `cpus_per_node` test must be three times the ones assigned to the 48 `cpus_per_node` one. Surprisingly enough given the reasoning above, results do not show a clear dominance of 16 CPUs per node tests over 48 CPUs ones with small datasets. As the size of the input grows and as more calculations are required, the performance is the predicted and 16 `cpus_per_node` tests are quicker.

5.3.3 On time complexity

Figures 11 and 12 plot the evolution of the execution time as a function of the size of the input, i.e the number of points, for a fixed number of working nodes and two different `cpus_per_node` parameters, 16 and 48 respectively. 5.3.2 shows that there is no evident difference between the two options, as a consequence both Figures look really similar. The main conclusion one can deduce is that, the higher the number of workers (or CPUs in general) the least the application behaves quadratically as its naive sequential implementation and the better PyCOMPSs performs.

5.3.4 Testing with real data and comparing with S.o.A implementations

All the previous tests have been performed with synthetic data. The application must as well prove to be useful in real-life applications and when compared to other implementations. A real case use is the one presented in [10] where clustering is applied to an app trace to "outline the different trends of the CPU computation areas of parallel applications". Within the same paper, an implementation of the DBSCAN is proposed and thus we will be able to compare both performances. The trace used is one corresponding to:

6 Conclusion

Following along the reasoning from 5.3, the algorithm is really sensitive to changes in the input parameters. Thus

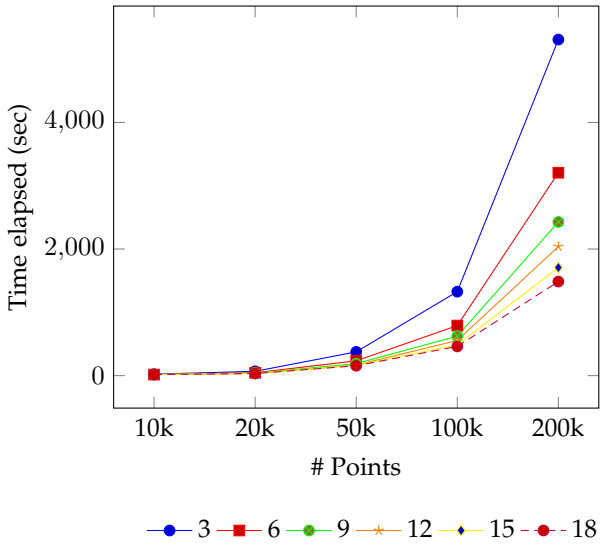


Fig. 11. Evolution of time elapsed with a fixed number of workers. 16 CPUs per node.

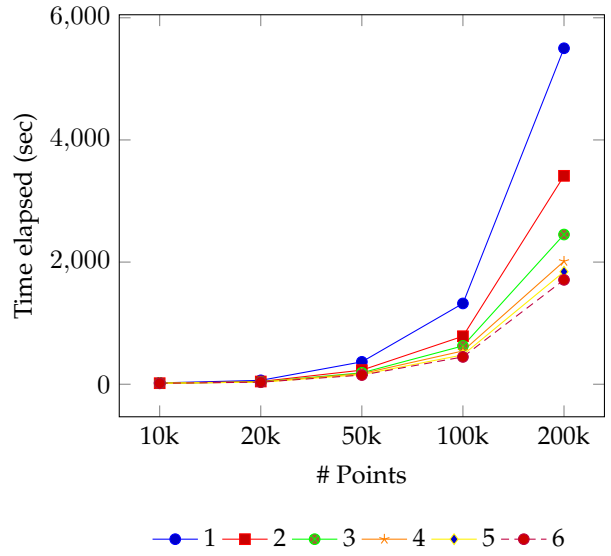


Fig. 12. Evolution of time elapsed with a fixed number of workers. 48 CPUs per node.

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 measured speedup might be worse in comparison to if we were to run the algorithm manually test by test.

6.1 Further Development and Improvement Proposals

The main development recommended would be performing a smart guess 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 can not be ensured.

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.

Acknowledgements

Acks

References

[1] Badia, R. M., J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent *COMP Superscalar, an interoperable programming framework* SoftwareX, Volumes 34, December 2015, Pages 3236,

[2] D. Arlia and M. Coppola, *Experiments in Parallel Clustering with DBSCAN* in Euro-Par 2001, Springer, LNCS, 2001, pp. 326-331.

[3] S. Brecheisen et al., *Parallel Density-Based Clustering of Complex Objects* Advances in Knowledge Discovery and Data Mining, pp. 179-188, 2006.

[4] Yaobin He et al. *MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data* Frontiers of Computer Science, vol 8, no. 1, pp 83-99, 2014.

[5] Md. Mostofa Ali Patwary et al. *A new scalable parallel DBSCAN algorithm using the disjoint-set data structure* Conference: High Performance Computing, Networking, Storage and Analysis (SC), 2012

[6] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, Laura Ricci *NG-DBSCAN: scalable density-based clustering for arbitrary data*. Proceedings of the VLDB Endowment Volume 10 Issue 3, November 2016, p. 157-168

[7] 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)

[8] MacQueen, J. B. (1967). *Some Methods for classification and Analysis of Multivariate Observations.* Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability.

[9] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, Jess Labarta, *PyCOMPSs: Parallel computational workflows in Python* IJHPCA 31(1): 66-82 (2017)

[10] J. Gonzalez, J. Gimenez, J. Labarta. *Automatic detection of parallel applications computation phases*.

TABLE 2
Execution time comparison with the same number of CPUs but different number of workers and different CPUs per node. Time measured in seconds.

Comparison CPUS per Node	**10k**		**20k**		**50k**		**100k**		**200k**	
#Workers - 16 (48)	**16**	**48**	**16**	**48**	**16**	**48**	**16**	**48**	**16**	**48**
3 (1)	23.44	21.81	68.22	63.6	375.47	366.49	1326.732	1322.73	5308.99	5501.241035
6 (2)	16.74	16.94	46.15	44.62	233.71	230.68	790.97	785.68	3205.49	3412.54
9 (3)	16.609	16.42	39.71	37.77	190.39	186.69	623.78	627.52	2427.87	2451.26
12 (4)	17.78	15.74	34.43	36.01	167.83	173.65	555.06	544.66	2039.61	2012.89
15 (5)	17.23	16.92	33.71	33.177	163.87	158.08	487.01	480.03	1707.57	1842.83
18 (6)	16.76	15.63	33.166	33.84	156.11	149.75	461.73	446.91	1486.20	1708.76