DBSCAN for PyCOMPS
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

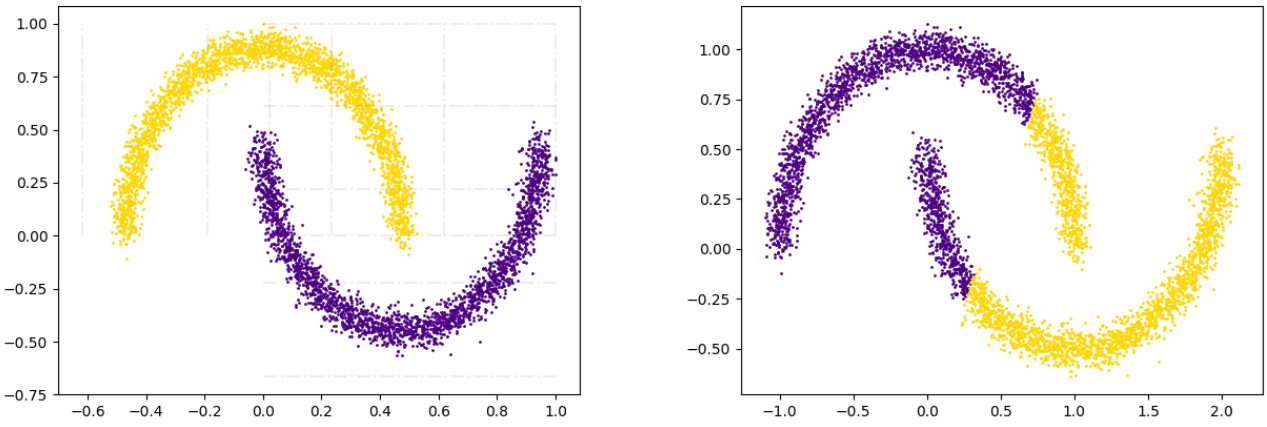
The algorithm implemented is a reinterpretation of the one exposed in [7], it is still though completely equivalent. The implementation can be found in this GitHub repository. The main algorithm would be structured as in Algorithm 1 (note that all the methods stated are later expanded and explained). 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]. It outputs the same dataset labeled with the cluster id they belong to.

Algorithm 1 Main method for the DBSCAN algorithm.

```
1:  function DBSCAN(data_file, epsilon, min_points)
2:      for all square in frag_data do
3:          frag_data ← load_data(data_file)
4:          local_clusters ← partial_scan(frag_data)
5:          cluster_map ← sync_clusters(local_clusters)
6:      end for
7:      frag_data ← compss_wait_on(frag_data)
8:      maps ← map_clusters(frag_data, cluster_map)
9:      for all square in frag_data do
10:         expand_clusters(square, maps)
11:     end for
12:     return 1
13: end function
```

Note that all the methods executed within the loop statements include the `@task` decorator and therefore can potentially run in parallel. Following along, this general scheme is to be divided and each different step analysed.

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 Loading

It is important to bear in mind the punctualisation from the section above: data is assumed to be normalised (each feature of each point belonging to the interval $[0, 1]$) and distributed in a file system grid-shaped. This is, if for instance the points are two-dimensional, a possible distribution would consist in the following files: `data_0_0.txt`, `data_0_1.txt`,..., `data_0_9.txt`, `data_1_0.txt`,..., `data_9_9.txt`. Where a point belongs to a file if each feature's first decimal position matches the file name. Were the files to be too big, two decimal places could be used as an identifier. Note that having multiples of a 100 files eases an hypothetical pre-processement if all the data was given in a single, gigantic, file.

The aim of this method is to read the data belongig to a cube in our space grid, a file in the file system, and load it to memory. When doing so, each point is given a cluster id. It will be really common throughout the whole implementation to follow a recursive approach within a single task to tackle possible load imbalacments. It is such a usual practice that it is worth introducing a generic explanation.

A recursive approach to tackle load balancing

The starting problem is a task that has as an input data file that can be arbitrarily big. This will cause an imbalancement in the tasks length and a decrease in the overall performance and speedup of the algorithm. It is easy to know the length of the data file and, with a simple recursive orquestrator, run tasks over chunks of the data file with a limited number of points (`THRESHOLD`). The generic structure would be that of Algorithm 2. Note that `orquestrate_task` is not a `@task` and is therefore executed by the master thread whilst `task` is indeed and will be executed by a worker.

Back to the data initialisation, for each data file, an orquestration following the structure stated in 4.1.1 is performed upon the length of each data file with a threshold value of approximately 1 % of the number of points in the dataset. For each chunk then the data is loaded to a Python object and the cluster id's initially set to noise level.

Algorithm 2 Limit the number of points processed by the same task.

function ORQUESTRATE_TASK($data\_file, q = 1, r = 0$)
  $THRESHOLD = TH\_VALUE$
  if len($data\_file$)$/q \geq TH\_VALUE$ **then**
    $out =$ orquestrate_task($data\_file, 2 * q, 0$)
    $out =$ orquestrate_task($data\_file, 2 * q, 1$)
  else
    $out =$ task($data\_file, q, r$)
  end if
  return out
end function

4.1.2 Partial Scan

The aim of this method is to find core points, i.e points with over min_points neighbors and make clusters of them. For each cube in our initial grid, known the grid size and ε it is possible to quickly determin the number of neighbour cubes necessary to correctly identify all the core points within our goal refgion. In Figure 2, it is easy to see the criterion followed to quickly obtain the neighbos of a square. Limiting the amount of neighbor squares, the number of points where a certain point within our cube might have neighbors. This first neighbor square retrieval might be seen as a lazy function since speed was prefered over accuracy in this case.

Once it becomes clear where possible neighbor points might be looked for, for each cube in the grid, the point-limiting approach is followed. As seen in Algorithm 2, the user must choose a `data-file` to divide. Inutively, and due to size constrains, one might assume that the files to be divided are the set of neighbors. In practice however, this approach proves to be mistaken. The objective of this method is to find all the core points within a square, i.e over `min_points` neighbors. Dividing the neighbors

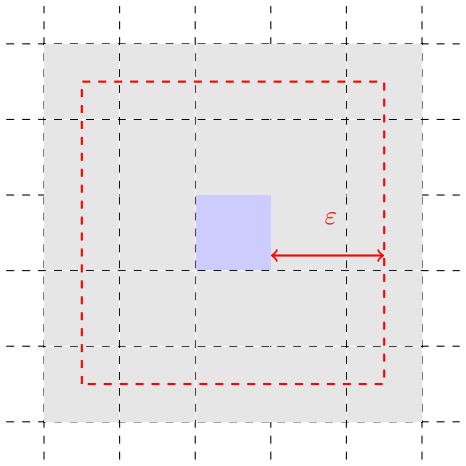


Fig. 2. Given the square in processment (blue) and ε, the squares shaded in gray will be considered neighbors.

would make impossible to decide wether a point is core or not at execution time. Given that the square itself is also considered a neighbor of itself, it is clear that there is no problem whatsoever in dividing the square to process instead of the neighbors. This explanation is exposed in Figure 3. Each worker looks for core points within their colored square and look for possible neighbors within all the gray and colored area.

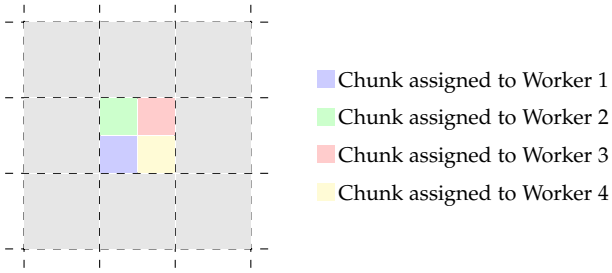


Fig. 3. Data partition for each task performing a `partial_scan`.

Each worker then looks for core points within his area. Once it is done with that, it tries and cluster them. I.e, it performs core-points clusters which do not exactly fit in the cluster definition given before. The output of each task is a chunk of the square dataset labeled. The labeling process is relevant and worth explaining in detail.

For each point processed, if the point is a core point then its label is updated to -1, if it is not a core point but has an already found core point in its neighborhood, a numerical pointer to the before mentioned core point is introduced so, in the final processing stages, labeling these reachable points will have no computing cost at all. Lastly, if it has at least more than one neighbor, the algorithm keeps track of them so that, if any gets later labeled with a cluster, it will be instant to label the first.

Once this initial core point search and labeling is finished, the worker clusters only the core points, labeling each one from zero onwards. For each square, a reduce task is implemented to combine the work performed by each worker. Since later a syncronisation task will take care of clusters that should be merged, all the clusters found by the different clusters are considered to be different and labels are updated accordingly. Note that they might need to be merged but moving this computation from the reduce task to the synchronisation one improves scalability. Algorithm 3 drafts this process.

Algorithm 3 Looks for all the core points in a certain square and performs an initial clustering.

```
1:  @task()
2:  function PARTIAL_SCAN(sub_square, neigh_data)
3:                                              ▷ Core point retrieval
4:      for all point in sub_square do
5:          if neighs(point, neigh_data) ≥ min_points then
6:              point is core_point
7:          else
8:              if core_p in neighs(point, neigh_data) then
9:                  link(point, core_point)
10:             else
11:                 map(point, neighs(point, neigh_data)
12:             end if
13:         end if
14:     end for
15:                                             ▷ Core point clustering
16:     clust_count = 0
17:     for all point in core_points do
18:         if close(point, cluster) then
19:             point_label = cluster_id
20:         else
21:             point_label = clust_count
22:             clust_count+ = 1
23:         end if
24:     end for
25:     return sub_square
26: end function
```

4.1.3 Cluster Synchronization

The partial scanning results in each square having an arbitrary number of clusters of core points contained in them. Note that not the whole cluster must necessary be contained within the square itself but all the points contained in it are, for sure, core points. The following task again, for each square, computes to which clusters contained in their neighbors each cluster from iself can be merged with. Note as well that the neighbors are those from Figure 2. As a consequence it outputs a sort of adjacency matrix where, for each cluster contained in the square, there is a list with all the squares it can be merged with.

It is important to notice that for a cluster to be considered possible to merge with, and since all the points considered are core points, only a point from each one must be at distance under ε. Therefore it is possible to now apply the recursive approach not over the square data but rather over the neighbors data as presented in Figure 4.

To gather and synchronise all the results from the different workers it is as simple as sequentally cover all the diferent adjacency matrix found and add a cluster to the merging list if it is not there. The whole algorithm is drafted in Algorithm 4.

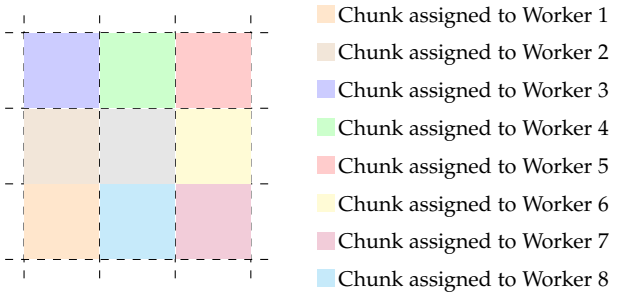


Fig. 4. Data partition for each task performing a `sync_clusters`.

Algorithm 4 Builds an adjacency matrix from a square w.r.t its neighbors.

```
1:  @task()
2:  function SYNC_CLUSTERS(square, sub_neigh_data)
3:      core_points ← get_core_points(square)
4:      for all point in core_points do
5:          neigh_cores ← get_core_points(sub_neigh)
6:          if close(point, n_point in neigh_cores) then
7:              adj_matrix[point_id].add(n_point_id)
8:          end if
9:      end for
10:     return adj_matrix
11: end function
```

4.2 Cluster Mapping

This is the only non-task method, i.e it must run sequentially, and it can be thought as a reduce method for each of the adjacency matrix provided by each square in the grid. This problem can be easily solved chosing the right data structure. A disjoint or union-find set immediately yields the result since it is the easier way to, given connections between nodes in a graph, find the connected components.

4.2.1 Cluster Expansion

The goal of this last task is to assign a label to all the reachable points. According to Algorithm 3 the non-core points can either be directly pointing to another core point in which case labeling them is as simple as copying the label of the point they are looking at or can keep track of all the points they were close to. This second case needs to be considered because, since the algorithm works sequentially point by point, it could happen that a point has a core point as a neighbor but by the time it is being porcessed the core point has not yet been processed and as a consequence he does not acknowledge it. In this case, hovering over all the points it was mapped to, and linking to the first one labeled with a cluster is sufficient. Remapping all the labels with the links found in the preceeding method is the last thing to do before printing the results to the out file. This process is drafted in Algorithm **??**.

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.

Algorithm 5 Label all the reachanle points within a cluster.

```
1:  @task()
2:  function EXPAND_CLUSTERS(square_data, mappings)
3:                                  ▷ Reachable points labeling
4:      non_core_points ← get_non_core(square_data)
5:      for all point in non_core do
6:          if point_id points to core_point then
7:              point_id ← core_point_id
8:          else if point_id maps to core_point then
9:              point_id ← core_point_id
10:         end if
11:     end for
12:         ▷ Remap the cluster labels according to the sync
13:     square_data_id ← remap(square_data_id, mappings)
14:     save_to_file(square_data)
15: end function
```

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

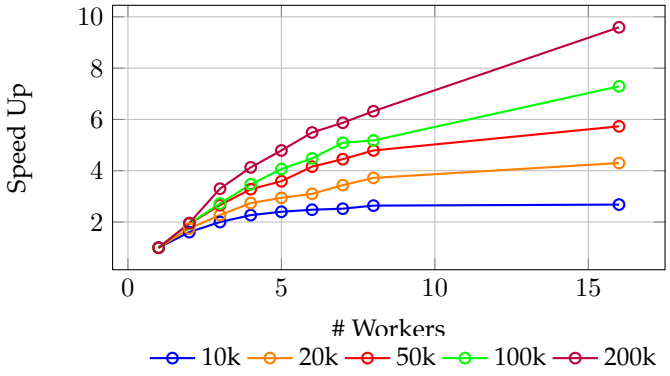


Fig. 5. Speed Up as a function of the number of workers for different datasets.

In figure 5 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 6.

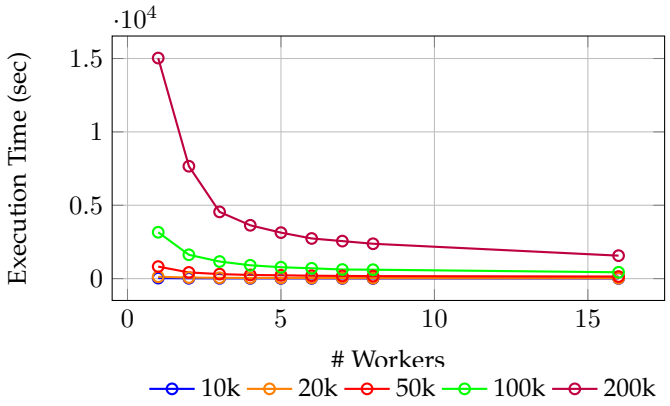


Fig. 6. Execution time as a function of the number of workers for different datasets.

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 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.

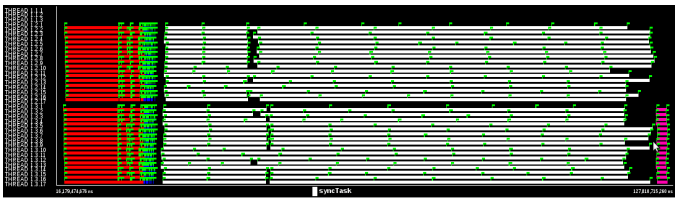


Fig. 7. Tracing with 10k points and `numParts` $= 16$.

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 7 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 8 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.

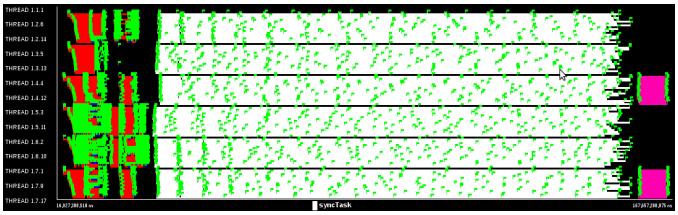


Fig. 8. Tracing with 20k points and `numParts` $= 32$.

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.

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		—		—		—		—	

5.3.3 *On time complexity*

Figures 9 and 10 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:

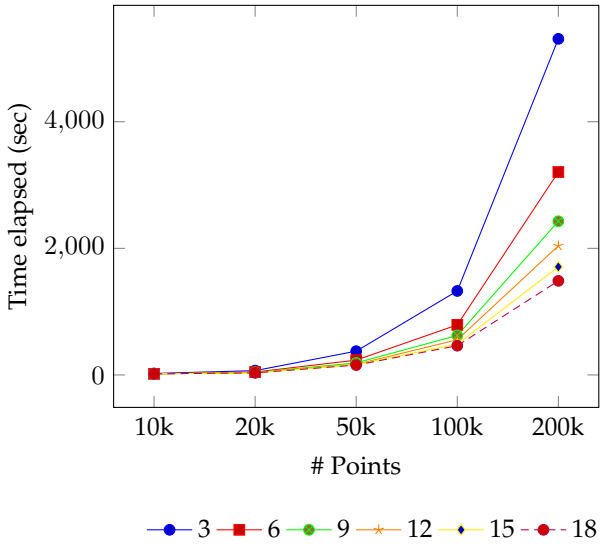


Fig. 9. Evolution of time elapsed with a fixed number of workers. 16 CPUs per node.

6 Conclusion

Following along the reasoning from 5.3, 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 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.

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

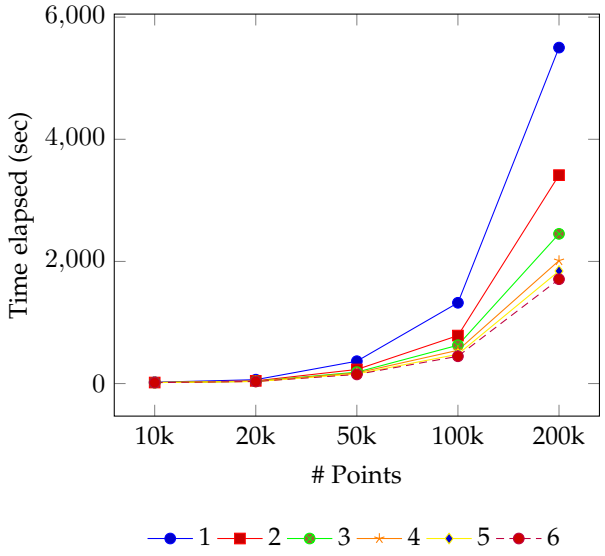


Fig. 10. Evolution of time elapsed with a fixed number of workers. 48 CPUs per node.

[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*.