TB-DBSCAN: Task Based - DBSCAN
An implementation using the PyCOMPSs framework

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 without having to guess the number of clusters.

TB-DBSCAN (Task Based - DBSCAN) is the first implementation of the DBSCAN clustering algorithm within the [1] framework for distributed computing. The main scope of this paper is to implement a task based version of the algorithm in Python with the PyCOMPSs syntax, test it in high-stress, highly distributed architectures and scale to hundreds of cores.

Which is the main contribution of this paper? Wait until results.

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 emphasises 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

Clustering algorithms aim to infer the hidden pattern underlying a non-classified dataset. The DBSCAN is a density based clustering algorithm introduced in Subsection 3.1

This section provides the necessary information to firstly understand the implementation proposed and secondly contextualise 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-neighbourhood** 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* neighbours within its ε-neighbourhood.
- A point p is said to be ***reachable*** if it lies within a ε-neighbourhood 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) $\forall p, q \in C, \exists p_1, \dots, p_r$ 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

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

In Algorithm 1 we present a schematic pseudocode of the original algorithm. For each new point being processed that has not been processed yet, we detect wether it is a core point or not. If it is, we stack all his neighbours and process them if they have not been processed yet. If one of these neighbours is a core point, all his neighbours are stacked on top of the previous. If the original point is not a core point, it is labeled as noise.

3.1.1 Other Clustering Algorithms

The urge to process larger and larger amounts of data without knowing their structure nor the underlying pattern puts clustering techniques in the limelight of machine learning algorithms. General purpose classifiactions difference between: hierarchical, centroid-based, distribution based and density based clustering.

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. 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.

Algorithm 1 Schematic Implementation of the DBSCAN algorithm.

```
function DBSCAN($data$, $epsilon$, $min\_points$)
    $next\_ID \leftarrow FIRST\_ID$
    for all $point$ in $data$ do
        if $point$ is $core\_point$ then
            label($point$)
            $point\_stack$.add(neigh($point$))
            for all $n\_point$ in $point\_stack$ do
                if $n\_point$ is $core\_point$ then
                    $point\_stack$.add(neigh($n\_point$))
                end if
                label($n\_point$)
            end for
        end if
    end for
end function
```

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 [**?**] 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. For remote objects to be available at the master thread a synchronisation is ran using the `compss_wait_on` built-in method. `COMPSs` is complemented by a set of tools for facilitating the monitoring and the post-mortem performance analysis (see Section).

4 Implementation Proposal

In this section, we describe our distributed, task-based DBSCAN algorithm (TB-DBSCAN), which is completely equivalent to the original implementation [?]. The code is public and can be found in the project's GitHub Repository[1]. TB-DBSCAN takes three input arguments: `data_file`, `min_points` and ε, and outputs a data file with the clustered data.

Algorithm 2 Main method of the TB-DBSCAN algorithm.

```
1:  function DBSCAN($data\_file$, $epsilon$, $min\_points$)
2:      for all $square$ in $data\_file$ do
3:          $frag\_data \leftarrow$ load_data($data\_file$)
4:          $local\_clusters \leftarrow$ partial_scan($frag\_data$)
5:          $cluster\_map \leftarrow$ sync_clusters($local\_clusters$)
6:      end for
7:      $frag\_data \leftarrow$ compss_wait_on($frag\_data$)
8:      $maps \leftarrow$ 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
```

Algorithm 2 drafts a general overview of TB-DBSCAN. Initially, TB-DBSCAN organises the input data in an n-dimensional space (D) where n is the number of features of each point. D is chunked in different regions (R_i) stored in independent data files. For each region, TB-DBSCAN retrieves core points and looks for local clusters (line 4). Then, TB-DBSCAN looks for possible cluster relations within a square and it's neighbours. Lastly it combines all the data gathered, updates the local labels and adds non-core points to the clusters (lines 7, 8 and 9).

In the following subsections we describe the different task methods in detail.

4.1 Task 1: Data Loading

The `load_data` task method loads a chunk of a region(R_i) to memory and initialises the points as noise (not belonging to any cluster).

We assume that each feature of each point in the dataset is normalised to the $[0,1]$ interval and that it is divided in different files, each one corresponding to a region $R_i \subset D$. For instance, if the points are two-dimensional, a possible file distribution would consist of the following files: `data_0_0.txt`, `data_0_1.txt`, ..., `data_0_9.txt`, `data_1_0.txt`, ..., `data_9_9.txt`. A point p belongs to a certain file R_i if the first decimal position of each feature matches the two numbers in the file name. I.e $p \in$ `data_i_j.txt` $\Leftrightarrow p = [0.i\ldots, 0.j\ldots]$. This spatial file distribution enables us to efficently query the neighbours of a point without needing a complex data structure.

Since the size of these files can be arbitrarily big, one task only load one chunk of the file into memory at a time. For this, we employ a *recursive scheme* to spawn a variable number of tasks, guaranteeing that each task does not process more than a fixed number of data points. This *recursive scheme* can be seen in Algorithm 3.

The `orquestrate_task` function takes three input parameters, a path to a data file (DF) whose length is known,

1. https://github.com/csegarragonz/DBSCAN-pyCOMPSs

Algorithm 3 Limit the number of points processed by a particular task.

```
1:  function ORQUESTRATE_TASK($data\_file, q = 1, r = 0$)
2:      $THRESHOLD = TH\_VALUE$
3:      if len($data\_file$)$/q \geq TH\_VALUE$ then
4:          $out =$ orquestrate_task($data\_file, 2*q, 2*r+0$)
5:          $out =$ orquestrate_task($data\_file, 2*q, 2*r+1$)
6:      else
7:          $out.append$(task($data\_file, q, r$))
8:      end if
9:      return $out$
10: end function
```

a quotient q initially set to 1 and a remainder r initially set to 0. If q divides the data file in chunks larger than a threshold value, two more `orquestrate_task` methods are invoked (lines 3-5). Otherwise a task is spawned which will process all the points in DF whose index $(p_1, p_2, \ldots)$ is congruent with r modulo q (line 7). All the outputs are lastly gathered to reduce the different results obtained from the different tasks applied to the different chunks from DF.

4.2 Task 2: Partial Scan

The `partial_scan` task method identifies and clusters the core points (i.e points with more than `min_points` neighbours) in a region.

Given a space region R_i let us define the neighbour regions of R_i, $NR(R\_i)$ as the set of regions reachable from R_i with ε distance. This is,

$$NR(R_i) = \{R_j \subset D : d(R_i, R_j)\}$$

where d is the infinity norm distance. We choose this metric to take advantage of our file distribution (see figure 1).

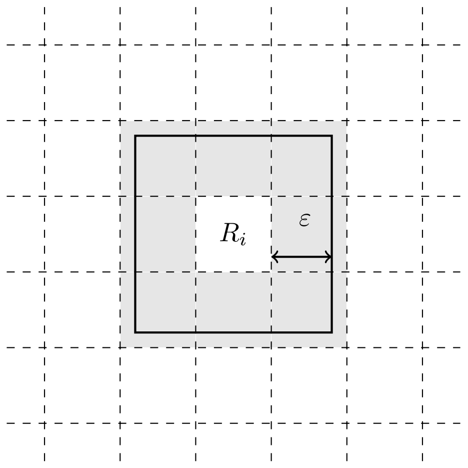

Fig. 1. R_i surrounded by $NR(R_i)$ coloured in gray and further regions left white.

Applying the recursive scheme to R_i each task looks for core points in a chunk of R_i considering all the points in $NR(R_i)$. We have chosen to fraction R_i rather than $NR(R_i)$ to guarantee the sufficiency of the results with no additional effort. I.e, dividing $NR(R_i)$ could lead to a task finding $\frac{min\_points}{2}$ neighbours of a given point and another one finding other $\frac{min\_points}{2}$ of the same point, missclassifying it as non-core when it really is.

The partial scan method labels each point p according to the following three rules:

(i) If $|N_\varepsilon(p)| \geq$ `min_points` we label it as a core point.
(ii) If $|N_\varepsilon(p)| <$ `min_points` but $\exists q \in N_\varepsilon(p)$ such that $|N_\varepsilon(q)| \geq$ `min_points` then TB-DBSCAN labels p in a way that remembers to copy q's id when the clustering is finished.
(iii) The point is not a core point neither has one in it's ε-neighbourhood. TB-DBSCAN keeps the original label but keeps track of p's neighbours.

We consider cases (ii) and (iii) to reduce the computational cost of labeling reachable points.

After labeling the points, we cluster only the core points. We realised that clustering all the points, not only core points, could lead to the following situation. A task clusters a set of points containing a reachable point q. This reachable point is at the same time in the ε-neighbourhood of another point in another cluster processed by another task. The standard DBSCAN implementation [7] would arbitrarilly assign q to one of the two clusters. Our syncing phase would merge them leading to an incorrect answer. To cluster the points, TB-DBSCAN recalculates the ε-neighbourhood of each core point and assigns the point to the first cluster within the ε-neighbourhood or to a new cluster if there is no cluster within it's $varepsilon$-neighbourhood. We do this recalculation because it is more efficient to re-compute these distances than it is to pre-calculate and store them.

Lastly, TB-DBSCAN reduces the results from the different tasks spawned by the orchestrator without paying attention to which clusters should be merfed or not.

The `partial_scan` method can be seen in Algorithm 4. Lines 4-14 cover the labeling process addressing cases (i), (ii) and (iii) in the different conditional clauses in lines 5, 8 and 10 respectively. The auxiliary method `link` stores that *point* has *core_p* in his ε-neighbourhood. This way when TB-DBSCAN assigns a cluster to *core_p*, *point* will be assigned to it as well. Similarly, `map` stores *point*'s neighbours. Lines 16-25 perform the local clustering where *cluster* in line 18 refers to the already labeled clusters and `close` computes wether *point* is close to another core point which is already assigned to a cluster. In this case it returns the cluster id.

Algorithm 4 Looks for core points and clusters them.

```
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($point, neigh\_data$) $\geq min\_points$ then
6:              $point$ is $core\_point$
7:          else
8:              if $core\_p$ in neighs($point, neigh\_data$) then
9:                  link($point, core\_p$)
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
```

4.3 Task 3: Cluster Synchronization

The `sync_clusters` method identifies which clusters can be merged together and stores this information in an adjacency matrix. An adjacency matrix is a square, boolean matrix. It has as many rows (and consequently columns) as nodes in a graph. The (i, j)-th entry of the matrix is 1 if and only if node i is connected to node j. In our situation, nodes are clusters. Two clusters C_1 and C_2 are connected if and only if $d(C_1, C_2) \leq \varepsilon$.

TB-DBSCAN considers clusters from a region R_i and looks for possible connected clsters in $NR(R_i)$. In fact, it *applies the recursive scheme to* $NR(R_i)$ so each spawned task looks for connected clusters in a chunk of $NR(R_i)$. In this case we can chunk the neighbour regions. To prove it, let C_1 be a cluster belonging to R_i and C_2 a cluster belonging to a chunk of $NR(R_i)$

$$d(C_1, C_2) \leq \varepsilon \Leftrightarrow \exists p \in C_1 \land q \in C_2 : d(p_1, p_2) \leq \varepsilon$$

Worst-case scenario, Task 1 finds a point p and a point q (note that Task 1 may not be treating all the points belonging to C_2) such that C_1 and C_2 are connected and stores it in it's adjacency matrix, Task 2 realises that C_1 and C_2 are connected as well (for a pair $\overline{p}$, $\overline{q}$ where $\overline{p}$ *can* be different than p and $\overline{q}$ *must* be different than q) and stores it in it's adjacency matrix. The reduce task succesfully combines the information. Each task processes all the points p in R_i and each point q in $NR(R_i)$ is processed by at least one task, thus all possible pairs are covered.

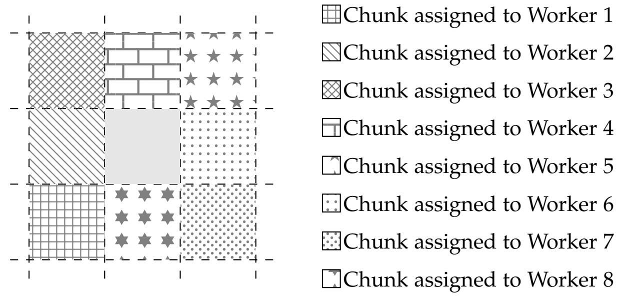
Chunk assigned to Worker 1
Chunk assigned to Worker 2
Chunk assigned to Worker 3
Chunk assigned to Worker 4
Chunk assigned to Worker 5
Chunk assigned to Worker 6
Chunk assigned to Worker 7
Chunk assigned to Worker 8

Fig. 2. Data partition for each task performing a `sync_clusters`.

For each core point in R_i (line 4 in Algorithm 5) we look for neighbor core points (line 6) and add their cluster id to the corresponding region in our adjacency mattrix (line 7).

Algorithm 5 Builds an adjacency matrix from a region w.r.t its neighbours.

```
@task()
function SYNC_CLUSTERS($region, sub\_neigh\_data$)
    $core\_points \leftarrow$ get_core_points($region$)
    for all $point$ in $core\_points$ do
        $neigh\_cores \leftarrow$ get_core_points($sub\_neigh$)
        if close($point, n\_point$ in $neigh\_cores$) then
            $adj\_matrix[point\_id].add(n\_point\_id)$
        end if
    end for
    return $adj\_matrix$
end function
```

4.4 Cluster Mapping

In the previous method we introduced our analogy from clusters to graph, let us in this section define it properly.

Let $\mathcal{G}$ be an undirected graph, $\mathcal{N}$ the set of nodes and $\mathcal{E}$ the set of edges. Let $\{M_1, \ldots, M_n\}$, where $M_i \in \{0,1\}^{|\mathcal{N}|\times|\mathcal{N}|}$ $\forall i \in \{1,\ldots,n\}$, be a set of incomplete adjacency matrices of G fulfiling 1.

$$\forall (u,v) \in \mathcal{E}, \exists j \in \{1,\ldots,n\} : M_j(u,v) = 1 \qquad (1)$$

By incomplete we mean that, except for $n = 1$, generally $M_i \neq AM$ $\forall i \in \{1,\ldots,n\}$ where AM is $\mathcal{G}$'s adjacency matrix.

Let us now state the following problem: *given the set $\{M_1,\ldots,M_n\}$ find the connected components of $\mathcal{G}$. I.e find $\{S_1,\ldots,S_m\}$, $S_i \subset \mathcal{N}$ fulfilling 2.*

$$\begin{aligned}&\forall i \in \{1,\ldots,m\}, \forall u, v \in \mathcal{N} \cap S_i \; \exists \{e_1, \ldots e_s\}\\ &e_j \in \mathcal{E} \text{ , a path connecting } u \text{ and } v \text{, and this path}\\ &\text{does not exist } \forall v \in (\mathcal{N} \cap S_i)^C\end{aligned} \qquad (2)$$

This problem can be solved in linear time using a BFS or a DFS [11], given that we have AM. Our case resembles more to the one of dinamically keeping track of the connected components since we will sequentially process each M_i. The S.o.A for this problem uses a Disjoint-Set data structure to do so.

A Disjoint-Set (or Union-Find, Merge-Find set) is a data structure that, exactly, keeps track of the elements of a set of elements partitioned in non-overlapping subsets (connected components). Computationally-wise, adding elements can be done in near-constant time ($\mathcal{O}(\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function).

We claim now that this problem is equivalent to the one of obtaining the core point clusters given the set of adjacency matrices. Each one corresponding to a region R_i. The equivalence holds for the following analogies.

(i) $\mathcal{N}$ is the set of all the clusters found in each region R_i stacked together
(ii) n (the number of incomplete adjacency matrices) is exactly the number of regions R_i
(iii) Each matrix M_i contains all the connections found by all the tasks spawned in region R_i

The nodes in the connected components of the graph are exactly all the clusters that need to be merged and the connected components themselves are the clusters we are looking for. Note that transitivity holds *because* we only considered core points for the clusters.

4.4.1 Cluster Expansion

In method `expand_clusters` we label reachable points. Non-core points that are in the ε-neighbourhood of a core point.

Thanks to the pre processing performed in the `partial_scan` method, non-core points can fall into two different groups:

(i) Points that, when processed, were already close to a core point. In this case their label is a numerical pointer to that core point so we just have to copy it's label. (Line 7 in Algorithm 6)
(ii) Points that, when processed, were *not* close to a core point. In this case, we have record of the neighbours. We iter through them and, if some has a cluster id, we copy it. (Line 9 in Algorithm 6)

We lastly update all the labels according to the general scope clusters found during the cluster mapping (line 13).

Algorithm 6 Label all the reachanle points within a cluster.

```
1:  @task()
2:  function EXPAND_CLUSTERS($square\_data, mappings$)
3:                                         ▷ Reachable points labeling
4:      $non\_core\_points \leftarrow$ get_non_core($square\_data$)
5:      for all $point$in$non\_core$ do
6:          if $point\_id$ points to $core\_point$ then
7:              $point\_id \leftarrow core\_point\_id$
8:          else if $point\_id$ maps to $core\_point$ then
9:              $point\_id \leftarrow core\_point\_id$
10:         end if
11:     end for
12:             ▷ Remap the cluster labels according to the sync
13:     $square\_data\_id \leftarrow$ remap($square\_data\_id, mappings$)
14:     save_to_file($square\_data$)
15: end function
```

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 genere-tion of supercomputers located in the Barcelona Super-computing 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 clus-ters, `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 num-ber 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 execu-tions were ran in MN4 with Scratch file system. Results are summarized in Table 1

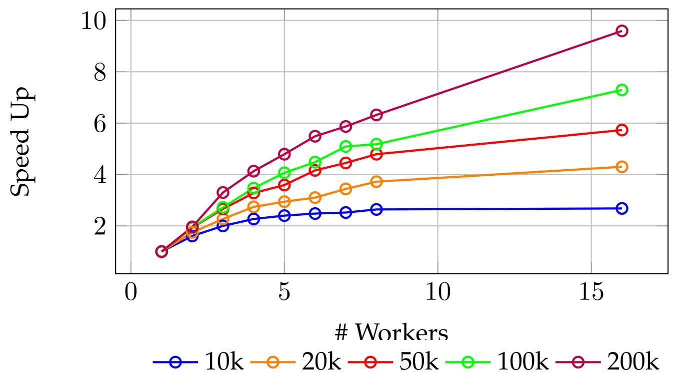

Fig. 3. Speed Up as a function of the number of workers for different datasets.

In figure 3 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 4.

5.3 Critical Analysis

5.3.1 *Dependence on the scale parameters:*

From the explanation in Section 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

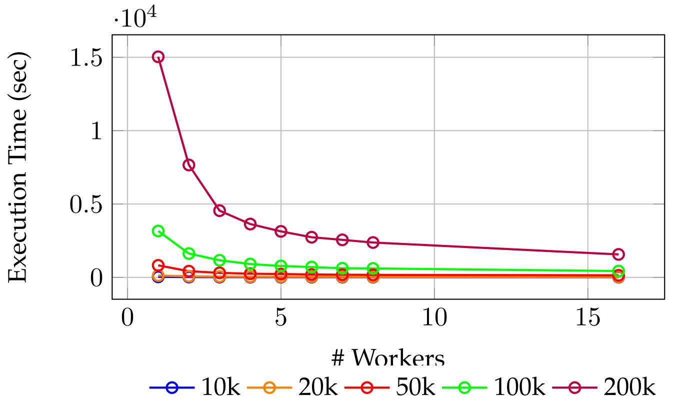

Fig. 4. Execution time as a function of the number of workers for different datasets.

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.

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 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 the opposite be-havior 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.

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

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		—		—		—		—	

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 5 and 6 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 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.

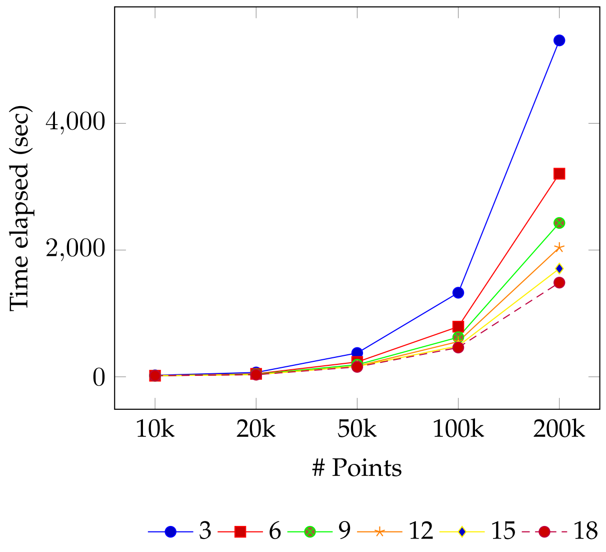

Fig. 5. Evolution of time elapsed with a fixed number of workers. 16 CPUs per node.

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, Pages 3236, (2015)

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

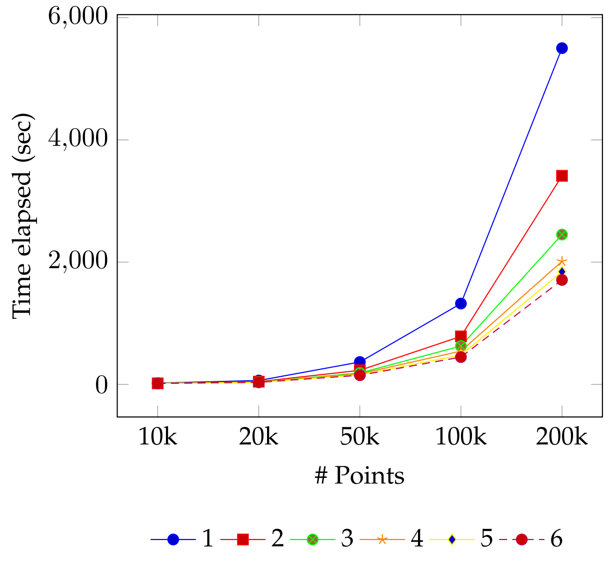

Fig. 6. Evolution of time elapsed with a fixed number of workers. 48 CPUs per node.

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

[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, p. 157-168 (2016)

[7] Ester, Martin Kriegel, Hans-Peter Sander, Jorg Xu, Xiaowei *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) (1996)

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

[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*.

[11] Hopcroft, J., Tarjan, R. *Algorithm 447: efficient algorithms for graph manipulation* Communications of the ACM, 16 (6): 372378 (1973)