VERKFRÆÐISTOFNUN
HÁSKÓLA ÍSLANDS

# Survey and Performance Evaluation of DBSCAN Spatial Clustering Implementations for Big Data and High-Performance Computing Paradigms

Helmut Neukirchen
helmut@hi.is

November 8, 2016

Report nr. VHI-01-2016, Reykjavík 2016

Helmut Neukirchen. Survey and Performance of DBSCAN Implementations for Big Data and HPC Paradigms,
Engineering Research Institute, University of Iceland, Technical report VHI-01-2016, November 2016

Engineering Research Institute, University of Iceland, Hjarðarhagi 2-6, IS-107 Reykjavík, Iceland

## Abstract

Big data is often mined using clustering algorithms. *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) is a popular spatial clustering algorithm. However, it is computationally expensive and thus for clustering big data, parallel processing is required. The two prevalent paradigms for parallel processing are *High-Performance Computing* (HPC) based on *Message Passing Interface* (MPI) or *Open Multi-Processing* (OpenMP) and the newer big data frameworks such as Apache Spark or Hadoop. This report surveys for these two different paradigms publicly available implementations that aim at parallelizing DBSCAN and compares their performance. As a result, it is found that the big data implementations are not yet mature and in particular for skewed data, the implementation's decomposition of the input data into parallel tasks has a huge influence on the performance in terms of running time.

# Contents

# 1 Introduction

Spatial information contained in big data can be turned into value by detecting spatial clusters. For example, areas of interest or popular routes can be determined by this means from geo-tagged data occurring in social media networks. This has many applications ranging from commercial settings such as advertising to administrative settings such as traffic planning or disaster response [14].

A popular spatial clustering algorithm is *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) [18]. Unfortunately, DBSCAN is is computationally expensive and thus for clustering big data, parallel processing is required. The two main general purpose paradigms for parallel processing are on the one hand *High-Performance Computing* (HPC) based on *Message Passing Interface* (MPI) or *Open Multi-Processing* (OpenMP) and on the other hand the newer big data frameworks such as the MapReduce-based Apache Hadoop or the *Resilient Distributed Dataset* (RDD)-based Apache Spark. However, the DBSCAN algorithm has been defined as a serial, non-parallel algorithm. Therefor, several variants of DBSCAN have been suggested to parallelize its execution. This paper compares the performance of different publicly available implementations that aim at parallelizing DBSCAN for the HPC and big data paradigms.

The outline of this paper is as follows: subsequent to this introduction, we provide foundations on DBSCAN, HPC and big data. Afterwards, in Section 3, we describe as related work other comparison of algorithms running on HPC and big data platforms. In Section 4, we survey existing DBSCAN implementations. Those implementations that were publicly available are evaluated with respect to their running time in Section 5. Based on the experience of trying to evaluate all the implementations described in scientific publications, some side nodes on good academic practise are made in Section 6, before we conclude with a summary and an outlook in Section 7.

# 2 Foundations

## 2.1 DBSCAN

The clustering algorithm *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) [18] has the nice property that the number of the clusters need not to be known in advance, but are rather automatically determined, that it is almost independent from the shape of the clusters, and that it can deal with and filter out noise. Basically, the underlying idea is that for each data point, the neighbourhood within a given *eps* radius has to contain at least a *minpts* points to form a cluster, otherwise it is considered as noise.

In the simplest implementation, finding all points which are in the *eps* neighbourhood of the currently considered point, requires to check all $n-1$ points of the input data, doing this for each of the $n$ input points leads to a complexity of $O(n^2)$. Using spatially sorted data structures for the neighbourhood search, such as R-trees [23], R*-trees [8], or kd-trees [9], reduces the overall complexity to $O(n \log n)$. The standard algorithms to populate such spatially sorted data structures cannot run in parallel and require in particular to have the entire input data available in non-distributed memory.

Even if the problem of having a distributed, parallel-processing variant of populating and using a spatially sorted data structure is solved (in order to bring the overall complexity down to $O(n \log n)$), there are further obstacles in parallelizing DBSCAN so that it scales optimally.

But at least, the actual clustering can be easily parallelized: typically, all points that belong to the same partition (=a rectangle in case of 2 dimensional data, a cube in case of 3D data, or a hypercube for n≥3 dimensions) can be processed by one thread independently from other threads that process the remaining partitions of points. Only at the border of each rectangle/cube/hypercube, points from direct neighbour rectangles/cubes/hypercubes need to be considered up to a distance of *eps*. For this, the standard approach of ghost or halo regions can be applied, meaning that these points from a neighbour partition need to be accessible as well by the current thread (in case of distributed memory, this requires to copy them into the memory of the respective thread). In a final step, those clusters determined locally in each partition which form a bigger cluster spanning multiple partitions need to be merged.

To achieve a maximum speed-up, not only an efficient spatially sorted data structure, communication overhead (e.g. for halo regions or finally merging locally obtained clusters), but also the size of the partitions is crucial: the input domain needs to be decomposed so that each thread or processor core get an equal share of the work. The simple approach of dividing the input domain into spatially equally sized chunks (for example as many chunks as processor cores are available) yields imbalanced workloads for the different cores if the input data is skewed: some partitions may then be almost empty, others very crowded. For heavily skewed data, the spatial size of each partition needs to be rather adjusted, for example in a way that each partition contains an equal number of points. If ghost/halo regions are used, then also the number of points in these regions need to be considered.

As shown, parallelizing DBSCAN in a scalable way beyond a trivial number of nodes or problems size is a challenge. For example, PDBSCAN [38] is a parallelized DBSCAN, however it involves a central master node to aggregate intermediate results which can be a bottleneck with respect to scalability.

## 2.2 HPC

*High-Performance Computing* (HPC) is tailored to typically CPU-bound computationally expensive jobs. Hence, special and rather expensive hardware, e.g. compute nodes containing fast CPUs including many cores and large amounts of RAM, very fast interconnects (e.g. InfiniBand) for communication between nodes, and centralized *Storage-Area Network* (SAN) with high bandwith due to a huge *Redundant Array of Independent Disks* (RAID) and fast attachment of them to the nodes.

To make use of the many cores per CPU, typically shared-memory multi-threading based on *Open Multi-Processing* (OpenMP) [15] is applied. To make use of the many nodes connected via the interconnects, an implementation of the *Message Passing Interface* (MPI) [31] is used which supports low-level 1:1 communication and also group communication. The underlying programming model is very low-level, the domain decomposition of the input data, all parallel processing, synchronisation, and communication has to be explicitly programmed using OpenMP and MPI. Typically rather low-level, but fast programming languages such as C, C++ and Fortran are used in the HPC domain. In addition to message passing, MPI supports parallel I/O to read different file sections from the SAN in parallel into the different nodes. To this aim, parallel file systems such as Lustre [32] or the *General Parallel File System* (GPFS) [27] make efficient use of the underlying RAID-like SAN to provide a high aggregated storage bandwidth. Typically, binary file formats such as netCDF or *Hierarchical Data Format* (HDF) [19] are used for storing input and output data in a structured way. They come with access libraries that are tailored to MPI parallel I/O.

While the low-level approach allows fast running implementations, their implementation takes considerable time. Furthermore, no fault tolerance is included: a single process failure on one of the many cores will cause the whole HPC job to fail which then needs to be restarted from the beginning. However, due to the server-grade hardware, hardware failures are considered to occur seldom (but still, they occur in practise).

## 2.3 Big Data

The big data paradigm is tailored to process huge amounts of data, however the actual computations to be performed on this data are often not that computationally intensive. Hence, cheap commodity hardware is sufficient for most applications. Being rather I/O-bound than CPU-bound, the focus is on *High-Throughput Computing* (HTC). To achieve high-throughput, locality of data storage is exploited by using distributed file systems storing locally on each node a part of the data. The big data approach aims at doing computations on those nodes where the data is locally available. By this means, slow network communication can be minimised. (Which is crucial, because rather slow Ethernet is used in comparison to the fast interconnects in HPC.)

An example distributed file system is the *Hadoop Distributed File System* (HDFS), introduced with one of the first open-source big data frameworks, Apache Hadoop [6] which is based on the MapReduce paradigm [17]. As it is intended for huge amounts of data, the typical block size is 64 MB or 128 MB. Hadoop has the disadvantage that only the MapReduce paradigm is supported which restricts the possible class of parallel algorithms and in particular may lead to unnecessarily storing intermediate data on disk instead of allowing to keep it in fast RAM. This weakness is overcome by Apache Spark [7] which is based on *Resilient Distributed Dataset*s (RDDs) [39] which are able to store a whole data set in RAM distributed in partitions over the nodes of a cluster. A new RDD can be obtained by applying in parallel on all partitions transformations to an input RDD. To achieve fault tolerance (Spark executor node failures are more severe as in Hadoop, because data is stored in RAM only instead of persistent disk storage), an RDD can be reconstructed by re-playing transformations on those RDDs (or respectively those partitions of them) that survived a failure. The initial RDD is obtained by reads from HDFS. While RDDs are kept in RAM, required data may not be available in the local RDD partition of a node. In this case, it is necessary to re-distribute data between nodes. Such shuffle operations are expensive, because slow network transfers are needed for them. Typically, textual file formats (such as *Comma-Separated Values* (CSV)) are used that can be easily split to form the partitions on the different nodes.

High-level, but (in comparison to C/C++ or Fortran) slower languages such as Java or the even more high-level Scala or Python are used in big data frameworks. Spark has over Python the advantage that it is compiled into Java bytecode and is thus natively executed by the *Java Virtual Machine* (JVM) used by Hadoop and Spark in contrast to Python implementations that may suffer from JVM/Python frictions. While the code to be executed is written as a serial code, the big data frameworks take behind the scenes care that each node applies in parallel the same code to the different partitions of the data. Even though Spark has many advantages compared to other distributed computation frameworks, the programmers still need to spend considerable time on implementing algorithms to make their parallelization as efficient as possible.

Because commodity hardware is used which is more error prone than HPC server-grade hardware, big data approaches need to anticipate failures as the norm and have thus built-in fault tolerance, such as redundant data storage or restarting failed jobs.

## 2.4 Convergence of HPC and Big Data

Convergence of HPC and big data approaches is taking place: typically either in form of *High-Performance Data Analysis* (HPDA), meaning that HPC is used in domains that used to be the realm of big data platforms [35], or big data platforms are used in the realm of HPC [20], or big data platforms are deployed at run-time on HPC clusters, however, sacrificing data locality aware scheduling [28]. In this paper, we investigate how mature this convergence is by comparing DBSCAN clustering implementations available for HPC and for big data platforms.

# 3 Related Work

HPC and big data data implementations for the same algorithms have been studied before. Jha et al. [28] compare these two parallel processing paradigms in general and introduce "Big Data Ogres" which refer to different computing problem areas with clustering being one of them. In particular, they evaluate and compare the performance of *k*-means clustering [30] implementations for MPI-based HPC, for MapReduce-based big data platforms such as Hadoop and HARP (which introduces MPI-like operations into Hadoop), and for the RDD-based Spark big data platform. In their evaluation, the considered HPC/MPI *k*-means implementation outperforms the other more big data-related implementation with the implementation based on HARP being second fastest and the implementation for Spark ranking third.

The influence of data storage locality as exploited by Spark and other big data platforms compared to centralized HPC SAN storage has been investigated by Wang et al. [37]. They use data intensive Grep search and compute intensive logistic regression as case study and come to the conclusion that even with a fast 47 GB/s bandwith centralized Lustre SAN, data locality matters for Grep and thus accesses to local SSDs are faster. However, for the logistic regression, locality of I/O does not matter.

# 4 Survey of Parallel DBSCAN Implementations

The following subsections contain a survey of the considered DBSCAN implementations. To be able to compare their run-time performance on the same hardware and using the same input, only open-source implementations have been considered. Where documented by the respective authors, details concerning complexity and the partitioning scheme used for domain decomposition, are described.

For comparison, we also used also ELKI 0.7.1 [36], an *Environment for DeveLoping KDD-Applications supported by Index-Structures*. ELKI is an optimised serial open-source DBSCAN implementation in Java which employs R*-trees to achieve $O(n \log n)$ performance. By default, ELKI reads space- or comma-separated values and it supports arbitrary dimensions of the input data.

## 4.1 HPC DBSCAN Implementations

A couple of parallel DBSCAN implementations for HPC platforms exist (as, for example, listed by Patwaryat et al. [34] or Götz et al. [22]). However, to our knowledge, only for two of them, the source is available: PDSDBSCAN and HPDBSCAN. In the following, we therefore restrict to these two. Both support arbitrary input data dimensions.

PDSDBSCAN by Patwary et al. [34] (C++ source code available on request from the PDSDBSCAN first author [33]) makes use of parallelization either based on shared memory using OpenMP or based on distributed memory using MPI. For their OpenMP variant, the the input data needs to fits into the available RAM; for the MPI variant, a pre-processing step is required to partition the input data onto the distributed memory. Details of this pre-processing step are not documented as the authors do not consider this step as part of their algorithm and thus it is neither parallelized nor taken into account when they measure their running times. The implementation reads the input data via the netCDF I/O library.

HPDBSCAN by Götz et al. [22] (C++ source code available from Bitbucket repository [21]) makes use of parallelization based on shared memory and/or distributed memory: besides a pure OpenMP and pure MPI mode, also a hybrid mode is supported. This is practically relevant, because an HPC cluster is a combination of both memory types (each node has RAM shared by multiple cores of the same node, but RAM is not shared between the many distributed nodes) and thus, a hybrid mode is most promising to achieve high performance. For the domain decomposition and to obtain a spatially sorted data structure with $O(\log n)$ access complexity, the arbitrary ordered input data is first indexed in a parallel way and then re-distributed so that each parallel processor has points in the local memory which belong to the same spatial partition. For load-balancing, these partitions are sized using a cost function that considers the number of comparisons by multiplying the number of points in a partition with the number of points in the cell neighbourhood (=number of pairs for which the distance function needs to be calculated). The command line version of the implementation reads the input data via the HDF I/O library.

## 4.2 Spark DBSCAN Implementations

Even though our search for Apache Spark big data implementations of DBSCAN[1] was restricted to JVM-based Java or Scala[2] candidates, we found several parallel open-source[3] implementations of DBSCAN: Spark DBSCAN, RDD-DBSCAN, Spark_DBSCAN, and DBSCAN On Spark. They are described in the following.

Spark DBSCAN by Litouka (source code via GitHub repository [29]) is declared as experimental and being not well optimised. For the domain decomposition to partition for parallel processing, the data set is considered initially as a large box full of points. This box is then along its longest dimension split into two parts containing approximately the same number of points. Each of these boxes is then split again. This process is repeated until the number of points in a box becomes less than or equal to a threshold, or a maximum number of levels is reached, or the shortest side of a box becomes smaller than $2\,eps$ [29]. Each such a box becomes a record of an RDD which can be processed in parallel, thus yielding an overall time complexity of $O(m^2)$ with $m$ being the number of points per box [2].

RDD-DBSCAN by Cordova and Moh [13] (source code via GitHub repository [12] which also points to minor forks). The authors state that it is loosely based on MR-DBSCAN [25]. Just as the above Spark DBSCAN by Litouka, the data space is split into boxes that contain roughly the same amount of data points until the number of points in a box becomes less than a threshold or the shortest side of a box becomes smaller than $2\,eps$. R-trees are used to achieve an overall $O(n \log n)$ complexity [13].

Spark_DBSCAN by GitHub user "aizook" (source code via GitHub repository [1]) is a very simple implementation (just 98 lines of Scala code) and was not considered any further, because of its complexity being $O(n^2)$ [2].

DBSCAN On Spark by Raad (source code via GitHub repository [4]) uses for domain decomposition a fixed grid independent from how many points are contained in each cell of the resulting "fishnet" grid. Furthermore, to reduce the complexity, no Euclidian distance function is used (= a circle with $2\,eps$ diameter), but the fishnet cells are rather used as a square box (with $2\,eps$ edge length) to decide concerning neighbourhood (see function `findNeighbors` in [4]). So, while it is called "DBSCAN On Spark" it implements only an approximation of the DBSCAN algorithm and does in fact return significantly differing (=worse) clustering results.

### 4.2.1 Common features and limitations of the Spark Implementations

All the considered implementations of DBSCAN for big data platforms assume the data to be in CSV (or space-separated) format.

All the Apache Spark DBSCAN implementations (except for the closed-source DBSCAN by Han et al. [24]) work only on 2D data: On the one hand, the partitioning schemes used

[1]Remarkably, the machine learning library MLlib which is a part of Apache Spark does not contain DBSCAN implementations.

[2]Note that also purely serial Scala implementations of DBSCAN are available, for example GSBSCAN from the Nak machine learning library [3]. However, these obviously make not use of Apache Spark parallel processing. But still, they can be used from within Apache Spark code to call these implementations in parallel, however each does then cluster disjoint, unrelated data points [11].

[3]There is another promising DBSCAN implementation for Spark by Han et al. [24]. However, it is not available as open-source. A kd-tree is used to obtain $O(n \log n)$ complexity. Concerning the partitioning, the authors state "We did not partition data points based on the neighborhood relationship in our work and that might cause workload to be unbalanced. So, in the future, we will consider partitioning the input data points before they are assigned to executors." [24] While the first author sent the binary code on request, the execution on our Spark cluster failed with `java.lang.ArrayIndexOutOfBoundsException`. Due to the lack of source code, it was not possible to fix this issue in order to evaluate that implementation.

for decomposition are based on rectangles instead of higher-dimensional hyper-cubes. On the other hand, for calculating the distance between points, most implementations use a hard-coded 2D-only implementation of calculating the Euclidian distance[4].

## 4.3 MapReduce DBSCAN Implementations

For further comparison, it would have been interesting to evaluate MapReduce-based DBSCAN implementations for the Apache Hadoop platform and candidates found to be worthwhile (because they claim to be able to deal with skewed data) were MR-DBSCAN [26, 25] by He et al. and DBSCAN-MR [16] by Dai and Lin. However, none of these implementations were available as open-source and e-mail requests to the respective first authors to provide their implementations either as source code or as binary were not answered. Hence, it is impossible to validate the performance claims made by these authors.

[4]Note that spheric distances of longitude/latitude points should in general not be calculated using Euclidian distance in the plane. However, as long as these points are sufficiently close together, clustering based on the simpler and faster to calculate Euclidian distance is considered as appropriate.

# 5 Evaluation of Parallel DBSCAN Implementations

Often, comparison between HPC and big data implementations are difficult as the implementations run typically on different cluster hardware (HPC hardware versus commodity hardware) or cannot exploit underlying assumptions (such as missing local data storage when deploying big data frameworks at run-time on HPC clusters using a SAN).

## 5.1 Hardware and Software Configuration

In this paper, the same hardware is used for HPC and Spark runs: the cluster JUDGE at Jülich Supercomputing Centre. JUDGE was formely used for HPC and has been turned into a big data cluster. It consists of IBM System x iDataPlex dx360 M3 compute nodes each comprising two Intel Xeon X5650 (Westmere) 6-core processors running at 2.66 GHz. For the big data evaluation, we were able to use 39 executor nodes, each having 12 cores or 24 virtual cores with hyper-threading enabled (=936 virtual cores) and 42 GB of usable RAM per node and local hard disk.

In the HPC configuration, a network-attached GPFS storage system, called JUelich STorage (JUST) cluster was used to access data (measured peak performance of 160 GB/s). The big data configuration relies on local storage provided on each node by a Western Digital WD2502ABYS-23B7A hard disk (with peak performance of 222.9 MB/s per disk, corresponding to 8.7 GB/s total bandwidth if all 39 nodes read their local disk in parallel). 200 GB on each disk were dedicated to HDFS using a replication factor of 2 and 128 MB HDFS block size.

The software setup in the HPC configuration was SUSE Linux SLES 11 with kernel version 2.6.32.59-0.7. The MPI distribution was MPICH2 in version 1.2.1p1. For accessing HDF5 files, the HDF group's reference implementation version 1.8.14 was used. The compiler was gcc 4.9.2 using optimisation level O3.

The big data software configuration was deployed using the Cloudera CDH 5.8.0 distribution providing Apache Spark version 1.6.0 and Apache Hadoop (including HDFS and YARN which was used as resource manager) version 2.6.0 running on a 64-Bit Java 1.7.0_67 VM. The operating system was CentOS Linux release 7.2.1511.

## 5.2 Input Data

Instead of using artificial data, a real data set containing skewed data was used for evaluating the DBSCAN implementations: geo-tagged tweets from a rectangle around the United Kingdom and Ireland (including a corner of France) in the first week of June 2014. The data was obtained by Junjun Yin from the National Center for Supercomputing Application (NCSA) using the Twitter streaming API. This data set contains 3 704 351 longitude/latitude points and is available at the scientific data storage and sharing platform B2SHARE [10]. There, the data is contained in file `twitterSmall.h5.h5`. A bigger Twitter data set `twitter.h5.h5` from the same B2SHARE location covers whole of June 2014 containing of 16 602 137 data points, some of them are bogus artefacts though – still we used it to check whether implementations are able to cope with bigger data sets; a 3D point cloud for the city of Bremen is also provided there, however it was not usable for benchmarking the surveyed DBSCAN implementations which typically support only 2D data.

Table 5.1: Size of the Used Twitter Data Sets

| | Data points | HDF5 size | CSV size | SSV with Ids |
|---|---|---|---|---|
| Twitter Small | 3 704 351 | 57 MB | 67 MB | 88 MB |
| Twitter Big | 16 602 137 | 254 MB | 289 MB | 390 MB |

Table 5.2: Used Open Source Repository Versions

| Implementation | Repository | Version date |
|---|---|---|
| HPDBSCAN | `bitbucket.org/markus.goetz/hpdbscan` | 2015-09-10 |
| Spark DBSCAN | `github.com/alitouka/spark_dbscan` | 22 Feb 2015 |
| RDD DBSCAN | `github.com/irvingc/dbscan-on-spark` | 14 Jun 2016 |
| DBSCAN on Spark | `github.com/mraad/dbscan-spark` | 30 Jan 2016 |

The original file of the small Twitter data set is in HDF5 format and 57 MB in size. To be readable by ELKI and the Spark DBSCAN implementations, it has been converted using the `h5dump` tool (available from the HDF group) into a 67 MB CSV version and into an 88 MB space-separated version (SSV) that contains in the first column an increasing integer number as point identifier (expected by some of the evaluated DBSCAN implementations). The size of these two data sets is summarised in Table 5.1.

For all runs, *eps* = 0.01 and *minpts* = 40 were used as parameters of DBSCAN. All further command line parameters can be found at `http://uni.hi.is/helmut/2016/08/17/dbscan-evaluation`.

## 5.3 DBSCAN Implementation Versions

The dates of the used DBSCAN implementation source code versions and their repository is provided in Table 5.2.

Note that by default, Spark makes each HDFS block of the input file an RDD partition. With the above file sizes of the small Twitter data set being lower than the used HDFS block size of 128 MB, this means that the initial RDD would contain just a single partition located on the node storing the corresponding HDFS block. In this case, no parallelism would be used to process the initial RDD. Therefore, if the Spark DBSCAN implementations did not anyway allow to specify the number of partitions to be used, the implementations were changed so that it is possible to specify the number of partitions to be used for the initial file read. This means, that non-local reads will occur, however as the overall processing time is much bigger than the time needed by a non-local read, the overhead of these non-local reads is negligible.

## 5.4 Measurements

Comparing the C++ PDSDBSCAN and HPDBSCAN implementations to the JVM-based DBSCAN implementations for Spark is somewhat comparing apples and oranges. Hence, we used as a further comparison a Java implementation, the pure serial ELKI with `-db.index "tree.metrical.covertree. SimplifiedCoverTree$Factory"` spatial indexing option running just on one of the cluster nodes. The times were measured using the POSIX command `time`.

Table 5.3: Deviation of Run-times

| Run | 1. | 2. | 3. | 4. |
|---|---|---|---|---|
| Time (s) | 653 | 546 | 553 | 560 |

As usual in Hadoop and Spark, the Spark DBSCAN implementations create the output in parallel resulting in one file per parallel acRDD output partition. If a single output file is intended, it can be merged afterwards, however this time is not included in the measured run-time. The reported times were taken from the "Elapsed" line of the application's entry in the Spark web user interface for the completed run.

For the number of experiments that we did, we could not afford to re-run all of them multiple times to obtain averages or medians. However, for one scenario (RDD-DBSCAN, 233 executors, each using 4 cores with 912 initial partitions running on the small Twitter data set), we repeated execution four times. The observed run-times are shown in Table 5.3. For these 9–10 minute jobs, deviations of up to almost 2 minutes occurred. The fact that the first run was slower than the subsequent runs might be attributed to caching of the input file. In all of our experiments, we had exclusive use of the assigned cores.

### 5.4.1 Preparatory Measurements

In addition to the DBSCAN parameters *eps* and *minpts*, the parallel Spark implementations are influenced by a couple of parallelism parameters which were determined first.

Spark uses the concepts of *executors* with a certain numbers of threads/cores per executor process. Some sample measurements using a number of threads per executor ranging from 3 to 22 have been performed and the results ranging from 626 seconds to 775 seconds are within the above deviations, hence the influence of threads per executor is not considered significant. (Most of the following measurements have been made with 8 threads per executor – details can be found at `http://uni.hi.is/helmut/2016/08/17/dbscan-evaluation`.)

Parallelism in Spark is influenced by the number of partitions into which an RDD is divided. Therefore, measurements with varying initial partition sizes have been made (in subsequent RDD transformations, the number of partitions may however change depending on the DBSCAN implementations). Measurement for RDD-DBSCAN running on the small Twitter data set on the 932 core cluster (not all cores were assigned to executors to leave cores available for cluster management) have been made for a number of initial number of input partitions ranging from 28 to 912. The observed run-times were between 622 seconds and 736 seconds which are all within the above deviation range. Hence, these experiments do not give observable evidence of an optimal number of input partitions. However, in the remainder, it is assumed that making use of the available cores already from the beginning is optimal and hence 912 was used as the initial number of input partitions.

After the input data has been read, the DBSCAN implementations aim at distributing the read data points based on spatial locality in order to parallelize the subsequent clustering step: as described in section 4.2, most Spark DBSCAN implementations aim at recursively decomposing the input domain into spatial rectangles that contain approximately an equal number of data points and they stop doing so as soon as a rectangle contains only a certain number of points; however, a rectangle becomes never smaller than 2 *eps* edge length. Assuming that the subsequent clustering steps are also based on 912 partitions, the 3 704 351 points of the small Twitter data set divided by 912 partitions yield 4061 points per partition as optimal rectangle size. However, due to the fact a rectangle becomes never smaller than 2 *eps* edge length, some rectangles of that size still contain more points (e.g. in the dense-

Table 5.4: Influence of number of points used in domain decomposition

| No. of points threshold | 4 061 | 9 000 | 20 000 | 25 000 | 50 000 |
|---|---|---|---|---|---|
| Times (s) | 1 157 | 823 | 867 | 675 | 846 |

populated London area, some of these 2 *eps* rectangle contain up to 25 000 data points) and thus, the domain decomposition algorithm terminates with some rectangles containing more than that number of points.

Experiments have been made with a couple of different number of points used as threshold for the domain decomposition. According to the results are shown in Table 5.4, a threshold of a minimum of 25 000 data points per rectangle promises fastest execution. As this number is also the lowest number that avoids the domain decomposition to terminate splitting rectangles because of reaching the 2 *eps* edge length limit, a natural explanation would be that all rectangles contain an approximately equal number of points thus leading to best load balancing. However, this contradicts in fact later findings discussed in Section 5.5.

### 5.4.2 Run-time Measurements on Small Data Set

After these algorithm parameters have been determined based on these preparatory preparatory experiments, a comparison of the run-times of the different implementations was made when clustering the small Twitter data set.

Table 5.5 shows results using a lower number of cores in parallel. The C++ implementation HPDBSCAN (running in MPI only mode) performs best in all cases and scales well: even with just one core, only 114 seconds are needed to cluster the small Twitter data set. Second in terms of run-time is C++ PDSDBSCAN (MPI variant), however, the scalability beyond 8 cores is already limited.

Even the Java ELKI which is optimised for a serial execution is much slower than the C++ implementations. The implementations for Spark are even more slower (for all of them, 912 initial input partitions were used). For RDD-DBSCAN, 25 000 data points were used as domain decomposition threshold. (Spark DBSCAN was not measured using a low number of cores, because already with a high number of cores it was very slow.) When running on many cores, the Spark implementations beat ELKI but they are still by one (DBSCAN on Spark) or two (RDD-DBSCAN) magnitudes slower than HPDBSCAN and do not scale as well. While DBSCAN on Spark is faster than RDD-DBSCAN, it does only implement a simple approximation of DBSCAN and thus delivers different clusters than DBSCAN and should therefore not be considered as DBSCAN.

Table 5.6 shows results using a higher number of cores. (No measurements of any of the two DBSCAN HPC implementations on the small Twitter data set have been made, as we can already see from Table 5.5 that using a higher number of cores does not give any gains on this small data set – measurements with many cores for HPDBSCAN running on the bigger Twitter data set are presented later). For Spark DBSCAN, an initial experiment has been made using 928 cores (and 25 000 data points as domain decomposition threshold just as for RDD-DBSCAN), but as it was rather slow, so no further experiments have been made for this implementation. For RDD-DBSCAN, no real speed-up can be observed when scaling the number of cores (run-times are more or less independent from the number of used cores and constant when taking into account the measurement deviations to be expected). The same applies to the DBSCAN on Spark implementation.

As pointed out in Section 4.3, it would have been interesting to compare the running

Table 5.5: Run-time (in Seconds) vs. Number of Cores

| Number of cores | 1 | 2 | 4 | 8 | 16 | 32 |
|---|---|---|---|---|---|---|
| HPDBSCAN MPI | 114 | 59 | 30 | 16 | 8 | 6 |
| PDSDBSCAN MPI | 288 | 162 | 106 | 90 | 85 | 88 |
| ELKI | 997 | – | – | – | – | – |
| RDD-DBSCAN | 7 311 | 3 521 | 1 994 | 1 219 | 889 | 832 |
| DBSCAN on Spark(*) | 1 105 | 574 | 330 | 174 | 150 | 147 |

(*) Does only implement an approximation of the DBSCAN algorithm.

Table 5.6: Run-time (in Seconds) vs. Number of Cores

| Number of cores | 58 | 116 | 232 | 464 | 928 |
|---|---|---|---|---|---|
| Spark DBSCAN | – | – | – | – | 2 406 |
| RDD-DBSCAN | 622 | 707 | 663 | 624 | 675 |
| DBSCAN on Spark(*) | 169 | 167 | 173 | 183 | 208 |

(*) Does only implement an approximation of the DBSCAN algorithm.

time of MapReduce-based implementations using the same data set and hardware. Han et al. [24] who tried as well without success to get the implementations of MR-DBSCAN and DBSCAN-MR, developed for comparison reasons their own MapReduce-based implementation and observed a 9 to 16 times slower performance of their MapReduce-based DBSCAN implementation in comparison to their implementation for Spark.

### 5.4.3 Run-time Measurements on Big Data Set

While the previous measurements were made using the smaller Twitter data set, also the bigger one containing 16 602 137 points was used in experiments. While HPDBSCAN can easily handle it, the Spark implementations have problems with this 289 MB CSV file.

When running any of the Spark DBSCAN implementations while making use of all available cores of our cluster, we experienced out-of-memory exceptions[1]. Even though each node in our cluster has 42 GB RAM, this memory is shared by 24 virtual cores of that node. Hence, the number of cores used on each node had to be restricted using the `-executor-cores` parameter, thus reducing the parallelism, but leaving each thread more RAM (which was adjusted using the `-executor-memory` parameter).

The results for the big Twitter data set are provided in Table 5.7. HPDBSCAN (in the hybrid version using OpenMP within each node and MPI between nodes) scales well. Spark DBSCAN failed throwing the exception `java.lang.Exception: Box for point Point at (51.382, -2.3846); id = 6618196; box = 706; cluster = -2; neighbors = 0 was not found`. DBSCAN on Spark did not crash, but returned completely wrong clusters (it anyway does not cluster according to the original DBSCAN). RDD-DBSCAN took almost one and a half hour. While DBSCAN on Spark finishes faster, it delivered completely wrong clustering results. Due to the hopelessly long run-times of the DBSCAN implementations for Spark already with the maximum number of cores, we did not perform measurements with a lower number of cores.

[1]Despite these exceptions, we did only encounter once during all measurements a re-submissions of a failed Spark tasks – in this case, we did re-run the job to obtain a comparable measurement.

Table 5.7: Run-time (in Seconds) on Big Twitter Data Set

| Number of cores | 1 | 384 | 768 | 928 |
|---|---|---|---|---|
| HPBDBSCAN hybrid | 2 079 | 10 | 8 | – |
| ELKI | 15 362 | – | – | – |
| Spark DBSCAN | – | – | – | Exception |
| RDD-DBSCAN | – | – | – | 5 335 |
| DBSCAN on Spark(*) | – | – | – | 1 491 |

(*) Does only implement an approximation the DBSCAN algorithm.

## 5.5 Discussion of Results

Big data approaches aim at avoiding "expensive" network transfer of initial input data by moving computation where the data is available on local storage. In contrast, in HPC systems, the initial input data is not available locally, but via an external SAN storage system. As DBSCAN is not that I/O bound, but rather CPU bound, the I/O speed and file formats do not matter as much as the used programming languages and clever implementations in particular with respect to the domain decomposition for parallel execution.

HPDBSCAN outperforms all other considered implementations. Even the optimised serial ELKI is slower than a serial run of HPDBSCAN. This can attributed to C++ code being faster than Java and to the fact that HPDBSCAN uses the fast binary HDF5 file format, whereas all other implementations have to read parse and create/ write a textual input and output file. Having a closer look at the scalability reveals furthermore, that HPDBSCAN scales very well even into many cores which is not the case for the DBSCAN implementations available for Spark.

While DBSCAN on Spark is faster than RDD-DBSCAN, it delivers completely wrong clusters, hence it has to be considered useless.

For the given skewed data set, scalability of RDD-DBSCAN is only given for a low number of cores (the run-time difference between 16 and 32 cores is within to be expected measurement deviations), but not beyond[2]. A closer analysis of the run-time behaviour reveals that in the middle of the running implementation, only one long running task of RDD-DBSCAN is being executed by Spark: while one core is busy executing this task, all other cores are idle and the subsequent RDD transformations can not yet be started as they rely on the long running task. Amdahl's law [5] explains why RDD-DBSCAN does not scale beyond 16 or 32 cores: adding more cores just means adding more idle cores; while one core executes the long running task, the remaining 15 or 31 cores are enough to handle the workload of the other parallel tasks.

In fact, the serial ELKI using just one core is faster than RDD-DBSCAN using up to 8 cores and even beyond, RDD-DBSCAN is not that much faster and not really justifying using a high number of cores. The long running task in the middle of the RDD-DBSCAN run can be attributed to skewed data. As already explained in Section 5.4.1, the domain decomposition of RDD-DBSCAN does not yield smaller rectangles than 2 *eps* which limits the number of resulting spatial partitions and degree of parallelism and leads to imbalanced workloads. In contrast, HPBDBSCAN can use for its domain decomposition smaller rectangles (or hypercubes in the general case).

While the HPC implementations are much faster than the big data implementations, it is in favour of the latter that HPC implementations requires much more lines of code than the more abstract Scala implementations for Spark.

[2]Remarkably, the authors of RDD-DBSCAN [13] performed scalability studies only up to 10 cores.

# 6 Some Side Notes on Good Academic Practise

The experiences made when performing the described evaluation give also some insights into good and bad academic practise:

- With the advent of fake or bogus journals, not only honest authors may be trapped by these journals, they open also doors for plagiarism published in such journals[1]. As these journals are not concerned about their reputation, it is impossible to fight this kind of plagiarism as these journals will not retreat articles.
- Authors of scientific papers denying access to their implementations prevent reproducing their results. Thus, validating and reviewing their claims concerning the performance of their algorithms and implementations is impossible.
- Demonstrating scalability for just a few nodes (as done by some of the authors of the Spark implementations) is not sufficient to prove scalability on a bigger scale – an insight that is well accepted in the HPC scientific community but is still not common in the Big Data scientific community.

[1]A plagiarised version of Cordova et al. [13] using the title "Efficient Clustering on Big Data Map Reduce Using DBScan" has been accepted and published in the *International Journal of Innovative Research in Science, Engineering and Technology*.

# 7 Summary and Outlook

We surveyed existing parallel implementations of the spatial clustering algorithm DBSCAN for *High-Performance Computing* (HPC) platforms and big data platforms, namely Apache Hadoop and Apache Spark. For those implementations that were available as open-source, we evaluated and compared their performance in terms of run-time. The result is devastating: none of the implementations for Apache Spark is anywhere near to the HPC implementations. In particular on bigger (but still rather small) data sets, most of them fail completely and do not even deliver correct results.

As typical HPC hardware is much more expensive than commodity hardware used in most big data applications, one might be tempted to say that it is obvious that the HPC DBSCAN implementations are faster than all the evaluated Spark DBSCAN implementations. However, in this case study, the same hardware was used (in this case, HPC hardware, but using instead commodity hardware would not change the result). An analysis reveals that typical big data considerations such as locality of data are not relevant in this case study, but rather proper parallelization such as decomposition into parallel tasks matters. The Spark implementations of DBSCAN suffer from a suitable decomposition of the input data. Hence, skewed input data leads to tasks with extremely imbalanced running times. However, as the example geo-tagged Twitter data set shows, skew in spatial data is not unusual in practise.

However, it has to be said that in general, the big data platforms such as Spark offer resilience (such as re-starting crashed sub-tasks) and a higher level of abstraction (reducing time spent implementing an algorithm) in comparison to the HPC approach.

As future work, it is worthwhile to transfer the parallelization concepts of the HPDBSCAN [22] to a Spark implementation, in particular the domain decomposition below rectangles/hypercubes smaller than 2 *eps*. This would give the end user faster DBSCAN clustering on big data platforms. And having more or less the same parallelization ideas implemented on both platforms would also allow to assess influence of C/C++ versus Java/Scala and of MPI versus the RDD approach of Spark. Also, the scientific binary HDF5 data file format can currently not be processed by Hadoop or Spark in a way that data storage locality is exploited. As soon as the big data implementations of algorithms such as DBSCAN become less CPU bound and instead more I/O bound, data locality matters. A simple approach to be able to exploit the harder to predict locality of binary formats is to create some sort of "street map" in an initial and easily to parallelize run and use later-on the data locality information contained in this street map [20].

## 8 Acknowledgment

The author likes to thank all those who provided implementations of their DBSCAN algorithms. Special thanks go to the division Federated Systems and Data at the Jülich Supercomputing Centre (JSC), in particular to the research group High Productivity Data Processing and its head Morris Riedel for my stay as visiting scientist which led to fruitful discussions, to providing computing time, and finally enabled writing this paper. The author gratefully acknowledges the computing time granted on the supercomputer JUDGE at Jülich Supercomputing Centre (JSC).

# Bibliography

[1] Spark_DBSCAN source code. GitHub repository `https://github.com/aizook/SparkAI`, 2014.

[2] Apache Spark distance between two points using squaredDistance. Stack Overlow discussion `http://stackoverflow.com/a/31202037`, 2015.

[3] ScalaNLP/Nak source code. GitHub repository `https://github.com/scalanlp/nak`, 2015.

[4] DBSCAN On Spark source code. GitHub repository `https://github.com/mraad/dbscan-spark`, 2016.

[5] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[6] Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org/`.

[7] Apache Software Foundation. Apache Spark. `http://spark.apache.org/`, 2016.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, volume 19. ACM, 1990.

[9] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[10] Christian Bodenstein. HPDBSCAN Benchmark test files. `http://hdl.handle.net/11304/6eacaa76-c275-11e4-ac7e-860aa0063d1f`, 2015.

[11] Natalino Busa. Clustering geolocated data using Spark and DBSCAN. O'Reilly web page `https://www.oreilly.com/ideas/clustering-geolocated-data-using-spark-and-dbscan`, 2016.

[12] Irving Cordova. RDD DBSCAN source code. GitHub repository `https://github.com/irvingc/dbscan-on-spark`, 2014.

[13] Irving Cordova and Teng-Sheng Moh. DBSCAN on Resilient Distributed Datasets. In *2015 International Conference on High Performance Computing & Simulation (HPCS)*, pages 531–540. IEEE, 2015.

[14] Justin Cranshaw, Raz Schwartz, Jason I Hong, and Norman Sadeh. The livehoods project: Utilizing social media to understand the dynamics of a city. In *Proceedings of the Sixth International AAAI Conference on Weblogs and Social Media*. AAAI Press, 2012.

[15] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[16] Bi-Ru Dai and I-Chang Lin. Efficient map/reduce-based DBSCAN algorithm with optimized data partition. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 59–66. IEEE, 2012.

[17] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, Berkeley, CA, USA, 2004. USENIX Association.

[18] M Ester, HP Kriegel, J Sander, and X Xu. Density-based spatial clustering of applications with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1996.

[19] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[20] Fabian Glaser, Helmut Neukirchen, Thomas Rings, and Jens Grabowski. Using MapReduce for High Energy Physics Data Analysis . In *2013 International Symposium on MapReduce and Big Data Infrastructure*. IEEE, 2013/2014.

[21] Markus Götz. HPDBSCAN source code. Bitbucket repository `https://bitbucket.org/markus.goetz/hpdbscan`, 2015.

[22] Markus Götz, Christian Bodenstein, and Morris Riedel. HPDBSCAN: highly parallel DBSCAN. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, held in conjunction with SC15: The International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015.

[23] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, volume 14. ACM, 1984.

[24] Dianwei Han, Ankit Agrawal, Wei-Keng Liao, and Alok Choudhary. A novel scalable DBSCAN algorithm with Spark. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1393–1402. IEEE, 2016.

[25] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. Mr-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, 2014.

[26] Yaobin He, Haoyu Tan, Wuman Luo, Huajian Mao, Di Ma, Shengzhong Feng, and Jianping Fan. MR-DBSCAN: an efficient parallel density-based clustering algorithm using MapReduce. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 473–480. IEEE, 2011.

[27] IBM. General Parallel File System Knowledge Center. `http://www.ibm.com/support/knowledgecenter/en/SSFKCN/`, 2016.

[28] Shantenu Jha, Judy Qiu, Andre Luckow, Pradeep Mantha, and Geoffrey C Fox. A tale of two data-intensive paradigms: Applications, abstractions, and architectures. In *2014 IEEE International Congress on Big Data*, pages 645–652. IEEE, 2014.

[29] Aliaksei Litouka. Spark DBSCAN source code. GitHub repository `https://github.com/alitouka/spark_dbscan`, 2014.

[30] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, Volume 1: Statistics*, pages 281–297. University of California Press, Berkeley, California, 1967.

[31] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0. `http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, September 2012.

[32] OpenSFS and EOFS. Lustre homepage. `http://lustre.org/`, 2016.

[33] Md Mostofa Ali Patwary. PDSDBSCAN source code. Web page `http://users.eecs.northwestern.edu/~mpatwary/Software.html`, 2015.

[34] Md Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2012.

[35] Pethuru Raj, Siddhartha Duggirala, Anupama Raman, and Dhivya Nagaraj. *High-Performance Big-Data Analytics*. Springer, 2015.

[36] Erich Schubert, Alexander Koos, Tobias Emrich, Andreas Züfle, Klaus Arthur Schmid, and Arthur Zimek. A framework for clustering uncertain data. *PVLDB*, 8(12):1976–1979, 2015.

[37] Yandong Wang, Robin Goldstone, Weikuan Yu, and Teng Wang. Characterization and Optimization of Memory-Resident MapReduce on HPC Systems. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 799–808. IEEE, 2014.

[38] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. A fast parallel clustering algorithm for large spatial databases. *Data Mining and Knowledge Discovery*, 3(3):263–290, 1999.

[39] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.