HPDBSCAN: highly parallel DBSCAN

Conference Paper · November 2015

DOI: 10.1145/2834892.2834894

CITATIONS
27

READS
1,698

3 authors:

Markus Götz
Karlsruhe Institute of Technology
20 PUBLICATIONS **84** CITATIONS
SEE PROFILE

Christian Bodenstein
Forschungszentrum Jülich
12 PUBLICATIONS **61** CITATIONS
SEE PROFILE

Morris Riedel
Forschungszentrum Jülich
112 PUBLICATIONS **797** CITATIONS
SEE PROFILE

HPDBSCAN – Highly Parallel DBSCAN

Markus Götz
m.goetz@fz-juelich.de

Christian Bodenstein
c.bodenstein@fz-juelich.de

Morris Riedel
m.riedel@fz-juelich.de

Jülich Supercomputing Center
Leo-Brandt-Straße
52428 Jülich, Germany

University of Iceland
Sæmundargötu 2
101, Reykjavik, Iceland

ABSTRACT

Clustering algorithms in the field of data-mining are used to aggregate similar objects into common groups. One of the best-known of these algorithms is called *DBSCAN*. Its distinct design enables the search for an apriori unknown number of arbitrarily shaped clusters, and at the same time allows to filter out noise. Due to its sequential formulation, the parallelization of *DBSCAN* renders a challenge. In this paper we present a new parallel approach which we call *HPDBSCAN*. It employs three major techniques in order to break the sequentiality, empower workload-balancing as well as speed up neighborhood searches in distributed parallel processing environments *i*) a computation split heuristic for domain decomposition, *ii*) a data index preprocessing step and *iii*) a rule-based cluster merging scheme.

As a proof-of-concept we implemented *HPDBSCAN* as an OpenMP/MPI hybrid application. Using real-world data sets, such as a point cloud from the old town of Bremen, Germany, we demonstrate that our implementation is able to achieve a significant speed-up and scale-up in common HPC setups. Moreover, we compare our approach with previous attempts to parallelize *DBSCAN* showing an order of magnitude improvement in terms of computation time and memory consumption.

Categories and Subject Descriptors

A.2.9 [**General and reference**]: Cross-computing tools and techniques—*Performance*; F.5.8 [**Theory of computation**]: Design and analysis of algorithms—*Parallel algorithms*; H.3.8 [**Information systems**]: Information systems applications—*Data mining*; I.2.1 [**Computing methodologies**]: Parallel computing methodologies—*Parallel algorithms*

General Terms

Algorithms, Performance

SC2015 2015 Austin, Texas USA

Keywords

High performance computing, scalable clustering, parallel DBSCAN, HPDBSCAN, OpenMP/MPI hybrid

1. INTRODUCTION

Cluster analysis is a data-mining technique that divides a set of objects into disjoint subgroups, each containing similar items. The resulting partition is called a *clustering*. A clustering algorithm discovers these groups in the data by maximizing a similarity measure within one group of items—or cluster—and by minimizing it between individual clusters. In contrast to supervised learning approaches, such as classification or regression, clustering is an unsupervised learning method. This means that, it tries to find the mentioned structures without any apriori knowledge about the actual ground-truth. Typical fields of application for cluster analysis include sequence analysis in bio-informatics, tissue analysis in neuro-biology, or satellite image segmentation.

Clustering algorithms can be divided into four classes: partitioning-based, hierarchy-based, density-based and grid-based [17]. In this paper we will discuss aspects of the two latter classes. Specifically, we are going to talk about the density-based clustering algorithm *DBSCAN—density-based spatial clustering of applications with noise* [9]—and how to efficiently parallelize it using computationally efficient techniques of grid-based clustering algorithms. Its principal idea is to find dense areas, the cluster cores, and to expand these recursively in order to form clusters. The algorithm's formulation has an inherent sequential control flow dependency at the point of the recursive expansion, making it challenging to parallelize.

In our approach we break the interdependency by adopting core ideas of grid-based clustering algorithms. We overlay the input data with a regular hypergrid, which we use to perform the actual *DBSCAN* clustering. The overlayed grid has two main advantages. Firstly, we can use the grid as a spatial index structure to reduce the search space for neighborhood queries; and secondly, we can separate the entire clustering space along the cell borders and distribute it among all compute nodes. Due to the fact that the cells are constructed in a regular fashion, we can redistribute the data points, by facilitating halo areas, so that there is no intermediate communication required during the parallel computation step.

In spatially skewed datasets regular cell space splits would lead to an imbalanced computational workload, since most of the points would reside in dense cell subspaces assigned to a small number of compute nodes. To mitigate this we propose

a cost heuristic that allows us to identify data-dependent split points on the fly. Finally, the local computation results are merged through a rule-based cluster merging scheme, with linear complexity.

The remainder of this paper is organized as follows. The next section surveys related work. Section 3 describes details of the original *DBSCAN* algorithm. Subsequently, Section 4 discusses our parallelized version of *DBSCAN* by the name of *HPDBSCAN* and shows its algorithmic equivalence. Section 5 presents details of our hybrid OpenMP/MPI implementation. Evaluations are shown in Section 6, where we also present our benchmark method, datasets and the layout of the test environment. We conclude the paper in Section 7 and give an overview of possible future work.

2. RELATED WORK

There are a number of previous research studies dealing with the parallelization of *DBSCAN*. To the best of our knowledge, the first attempt was made by Xiaowei Xu in collaboration with Kriegel et al. [27]. In their approach, single neighborhood search queries are parallelized facilitating a distributed version of the R-Tree—*DBSCAN*'s initial spatial index data structure. They adopt a master-slave model, where the index is built on the master node and the whole data set is split among the slaves according to the bounding rectangles of the index. Subsequently, they merge the local cluster results by reclustering the bordering regions of the splits. Zhou et al. [28] and Arlia et al. [3] present similar approaches, where they accelerate the neighborhood queries by replicating the entire index on each of the slave nodes, assuming the index fits entirely into the main memory. Brecheisen et al. [5] have published a parallel version of *DBSCAN* that approximates the cluster using another clustering algorithm called *OPTICS*. Each of the cluster candidates found in this manner is sent to a slave node in order to filter out the actual from the guessed cluster points. The local results are then merged by the master into one coherent view. This approach, however, fails to scale for big databases, since the pre-filtering has to be done on the master, in main memory. Chen et al. [7] propose another distributed *DBSCAN* algorithm, called *P-DBSCAN* that is based on a priority R-Tree. Unfortunately, the paper does not state how the data is distributed or how the clusters are formed. An in-depth speed and scale-up evaluation is also not performed. A paper by Fu et al. [13] demonstrates the first Map-Reduce implementation of *DBSCAN*. The core idea of this approach is the same as the first parallelization attempt of Xu, that is, to parallelize singular neighborhood queries—this time in form of Map-Tasks. He et al. [19] present another implementation of a parallel *DBSCAN* based on the Map-Reduce paradigm. They are the first to introduce the notion of a cell-based preprocessing step in order to perform a fully distributed clustering without the need to replicate the entire dataset or to communicate inbetween. Finally, Patwary et al. [25] have published research work that shows a parallel *DBSCAN* that scales up to hundreds of cores. Their main contribution is a quick merging algorithm based on a disjoint-set data structure. However, they either need to fit the entire dataset into main memory or need a manual preprocessing step that splits the data within a distributed computing environment.

3. THE *DBSCAN* ALGORITHM

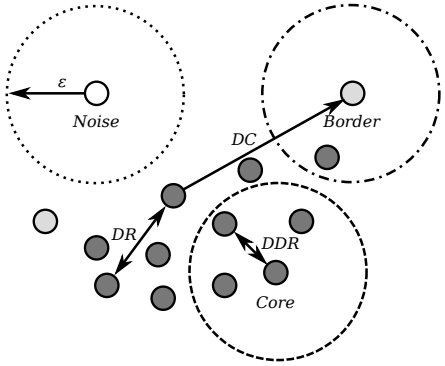

Figure 1: *DBSCAN* **clustering with** $minPoints = 4$

DBSCAN is a density-based clustering algorithm that was published 1996 by Ester et al. [9]. Its principal idea is to find dense areas and to expand these recursively in order to find clusters. A dense region is thereby formed by a point that has within a given search radius ε at least $minPoints$ neighboring points. This dense area is also called the *core* of a cluster. For each of the found neighbor points the density criteria is reapplied and the cluster is consequently expanded. All points that do not form a cluster core and that are not "absorbed" through expansion are regarded as *noise*.

A formal definition of the algorithm is as follows. Let X be the entire dataset of points to be clustered and $p, q \in X$ two arbitrary points of this set. Then the following definitions describe *DBSCAN* with respect to its parameters ε and $minPoints$. Figure 1 illustrates these notions.

Definition 1. Epsilon neighborhood (N_ε)—The epsilon neighborhood N_ε of p denotes all points q of the dataset X, which have a distance $dist(p, q)$ that is less or equal to ε, or formally: $N_\varepsilon(p) = \{q | dist(p, q) < \varepsilon\}$. In practice, the euclidean distance is often used for $dist$ making the epsilon-neighborhood of p equal to the geometrically surrounding hypersphere with radius ε.

Definition 2. Core point—p is considered a core point if the epsilon-neighborhood of p contains at least $minPoints$ number of points including itself: $Core(p) = |N_\varepsilon(p)| \geq minPoints$.

Definition 3. Directly density-reachable (DDR)—A point q is directly density-reachable from a point p, if p lies within q's epsilon-neighborhood and p is a core point, i.e., $DDR(p, q) = q \in N_\varepsilon(p) \wedge Core(p)$.

Definition 4. Density-reachable (DR)—A pair of points $p_0 = p$ and $p_n = q$ are called density reachable, if there exists a chain of directly density-reachable points—$\{p_i | 0 \leq i \wedge i < n \wedge DDR(p_i, p_{i+1})\}$—linking them with one another.

Definition 5. Border point—Border points are special cluster points that are usually located at the outer edges of a cluster. They do not fulfill the core point criteria but are still included in it due to direct density-reachability. Formally, this can be expressed as $Border(p) = |N_\varepsilon(p)| < minPoints \wedge \exists q : DDR(q, p)$.

Definition 6. Density-connected (DC)—Two points p and q are called density connected, if there is a third point r, such that r can density-reach p and q: $DC(p,q) = \exists r \in X : DR(r,p) \wedge DR(r,q)$. Note that density-connectivity is a weaker condition than density-reachability. Two border points can be density-connected, even though they are not density-reachable by definition due to not fulfilling the core point criteria.

Definition 7. Cluster—A cluster is a subset of the whole dataset, where each of the points is density-connected to all the other and that contains at least one dense region, or in other words a core point. This can be denoted as $\emptyset \subset C \subseteq X$ with $\forall p, q \in C : DC(p,q)$ and $\exists p \in C : Core(p)$.

Definition 8. Noise—Noise are special points that do not belong to any epsilon-neighborhood, such that $Noise(p) = \neg\exists q : DDR(q,p)$.

Listing 1 sketches pseudo code for a classic *DBSCAN* implementation. Some of the type and function definitions are left out, as their meaning can easily be inferred.

DBSCAN's main properties that distinguish it from more traditional clustering algorithms, such as *k-means* [17] for instance, are: *i*) it can detect arbitrarily shaped clusters that can even protrude into, or surround one another; *ii*) the cluster count does not have to be known apriori, and, *iii*) it has a notion of noise inside the data.

Finding actual values for ε and *minPoints* is dependent on the clustering problem and its application domain. Ester et al. [9] propose a simple algorithm for estimating ε. The core idea is to determine the "thinnest" cluster area through either visualization or a sorted 4-dist graph, and then choose ε to be equal to that width.

```
1  def DBSCAN(X, eps, minPoints):
2    clusters = list()
3    for p in X:
4      if visited(p):
5        continue
6      markAsVisited(p)
7      Np = query(p, X, eps)
8      if length(Np) < minPoints:
9        markAsNoise(p)
10     else:
11       C = Cluster()
12       add(clusters, C)
13       expand(p, Np, X, C, eps, minPoints)
14   return clusters
15
16 def expand(p, Np, X, C, eps, minPoints):
17   add(p, C)
18   for o in Np:
19     if notVisited(o):
20       markAsVisited(o)
21       No = query(o, X, eps)
22       if length(No) >= minPoints:
23         Np = join(Np, No)
24     if hasNoCluster(o):
25       add(o, C)
```

Listing 1: Classic *DBSCAN* pseudocode

4. *HPDBSCAN*

In this section we present Highly Parallel *DBSCAN*, or in short *HPDBSCAN*. Our approach to parallelize *DBSCAN* consists of four major stages. In the first step the entire dataset is loaded in equal-sized chunks by all processors in parallel. Then, the data is preprocessed. This entails assigning each of the d-dimensional points in the dataset to a virtual, unique spatial cell corresponding to their location within the data space, with respect to the given distance function. This allows us to sort the data points according to their proximity, and to redistribute them to distinct computation units of the parallel computing system. In order to balance the computational load for each of the processing units, we estimate the load using a simple cost heuristic accommodating the grid overlay.

After this division phase, we perform the clustering of the redistributed points in the second step locally on each of the processing units, i.e., we assign a temporary cluster label to each of the data points.

Subsequently, these have to be merged into one global result view in step three. Whenever the temporary label assigned by a processing unit disagrees with the ones in the halo areas of the neighboring processors, we generate cluster relabeling rules.

In the fourth step, the rules are broadcasted and applied locally. Figure 2 shows a schematic overview of the process using the fundamental modeling concepts (FMC) notation [21]. The next sections scrutinizes each of the substeps theoretically.

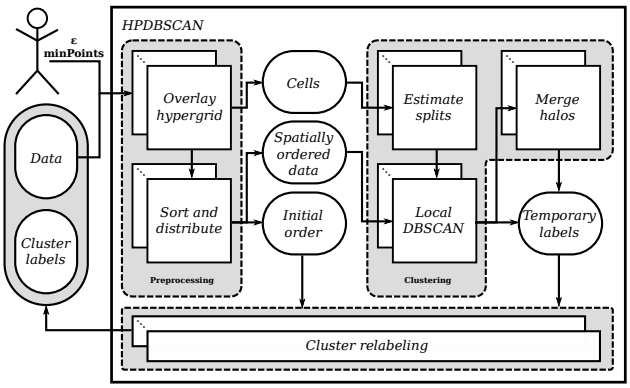

Figure 2: Schematic overview of *HPDBSCAN*

4.1 Grid-based data preprocessing and index

The original *DBSCAN* paper proposes the use of R-trees [4] in order to reduce the neighborhood search complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(log(n))$. The construction of the basic R-tree cannot be performed in parallel as it requires the entire dataset to be known. Therefore, other researchers [4, 27] propose to either just replicate the entire dataset, and perform linear neighborhood scans in parallel for each data item, or to use distributed versions of the R- or k-d-trees. However, He et al. [19] point out that these approaches do not scale in terms of memory consumption or communication cost with respect to large datasets and number of parallel processors.

Therefore, we have selected a far more scalable approach for *HPDBSCAN* that is based on grid-based clustering algorithms like, e.g., STING [17], and common spatial problem in HPC, like for example HACC in particle physics [16]. Its core idea is that the d-dimensional bounding box of the entire dataset, with respect to *dist*, is overlayed by a regular, non-overlapping hypergrid structure, which is then decomposed into subspaces by splitting the grid along the grid cell

boundaries. Each of the resulting subspaces is then exclusively assigned to a parallel processor that is responsible for computing the local clustering. In order to be able to do so in a scalable fashion, all the data points within a particular subspace should be in the local memory of the respective parallel processor, so that communication overhead is avoided. However, in most cases the data points will be distributed in arbitrary order within the dataset. Therefore, the data has to be indexed first and then redistributed to the parallel processor responsible for clustering the respective subspace.

In *HPDBSCAN* the indexing is performed by employing a hashmap with offset and pointers into the data memory. For this, all parallel processors read an arbitrary, non-overlapping, equally-sized chunk of the complete dataset first. Then each data item of a chunk is uniquely associated with the cell of the overlayed grid that it spatially occupies, and vice versa—every grid cell contains all the data items that its bounding box covers. This in turn enables us to order all of the local data items with respect to their grid cell so that they are consecutively placed in memory. Finally, an indexing hashmap can be constructed with the grid cells being the key, and the tuple of pointer into the memory and number of items in this cell the value. An indexing approach like this has an additional memory overhead of $\mathcal{O}(log(n))$ similar to other approaches like R- or k-d-trees. Figure 3 shows the indexing approach examplified by the dataset, introduced in Section 3, for the data chunk of a processing unit called `processor 1`.

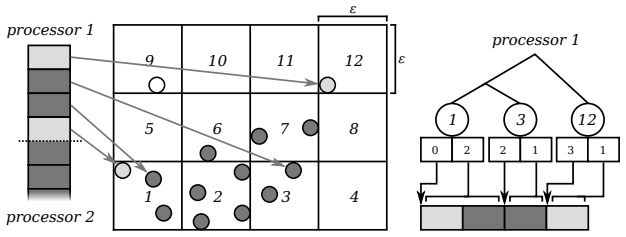

Figure 3: Sorted data chunks locally indexed by each processor using hashmaps pointing into the memory

Using that index the data redistribution can be performed in a straightforward fashion. The local data points of a parallel processor that do not lie within its assigned subspace are simply transferred to the respective parallel processor "owning" them. Afterwards all parallel processors have to rebuild their local data indices in order to encompass the received data. An efficient way of doing this is to send the section of the data index structure along with the data to the recipients. Due to the fact that the received and the local data are pre-sorted, the sent data index section and its memory pointers can be used to quickly merge them using, e.g., the merge-step of mergesort. The downside of the data redistribution approach is that it requires an additional memory overhead of $\mathcal{O}(\frac{n}{p})$, per parallel processor, with p being the number of parallel processors, to be able to restore the initial data arrangement after the clustering. However, since the additional overhead has linear complexity, it is maintainable even for large scale problems.

Using the described index structure, cell-neighborhood queries execute in amortized computation time of $\mathcal{O}(1)$. The cell-neighborhood N_{cell} thereby consists of all cells that are directly bordering the searched cell, its diagonals, as well as the cell itself with respect to all dimensions. For the cell labeled `6` in Figure 3 the cell-neighborhood is the set $\{1, 2, 3, 5, 6, 7, 9, 10, 11\}$. A formal definition follows.

Definition 9. Cell neighborhood—The cell neighborhood $N_{Cell}(c)$ of a given cell c denotes all cells d from the space of all available grid cells C that have a Chebychev distance $dist_{Chebychev}$ [6] of zero or one to c, i.e., $N_{Cell}(c) = \{d | d \in C \wedge dist_{Chebychev}(c, d) \leq 1\}$.

The actual epsilon neighborhood is then constructed from all points within the direct cell-neighborhood, filtered using the distance function *dist*. Šidlauskas et al. [26] show that a spatial grid index like this, is superior to R-trees and k-d-trees on index creation and queries, in terms of computation time, under the assumption that the cell granularity is optimal with respect to future neighborhood searches. Due to the fact that *DBSCAN*'s search radius is constant, the cells can trivially be determined to be hypercubes with the side length of ε. From a technical perspective it has the additional advantage that each of the d parts of the entire cell-neighborhood vector are consecutive in memory. This in turn enables data pre-fetching and the reuse of cell neighborhoods, thus reducing the number of cache misses.

In order to be able to answer all range queries within its assigned subspace, a parallel processor needs an additional one-cell-thick-layer of redundant data items, surrounding the grid splits that allow them to compute the cell neighborhood even at the edges of said splits. In parallel codes this is commonly referred to as *halos* or *ghost cells*. An efficient way of providing these halo cells is to transfer them along with the actual data during the data redistribution phase. This way the parallel processor will also index them along with the other data. Halo cells do not change the actual split boundaries in which a parallel processor operates and can be removed after the local clustering.

4.2 Cost heuristic

In the previous section we introduced the notion of subdividing the data space in order to efficiently parallelize *HPDBSCAN* and its spatial indexing especially also in distributed computing environments. However, we have not introduced a way to determine the boundaries of these splits. One of the most naïve approaches is to subdivide the space in equally-sized chunks in all dimensions, so that the resulting number of chunks equals to the number of available cores. While the latter part of the assumption is sensible as it minimizes the communication overhead, the former is not. Consider a spatially skewed dataset like shown in figure 4. The sketched dotted boundary, chunking the data space equally for two parallel processors, results in a highly unbalanced computational load, where one core needs to cluster almost all the data points and the other idles most of the time. Due to the fact that computing the *dist* function, while filtering the cell neighborhood, is for many distance functions the most processing intensive part of *DBSCAN*, this distribution pattern is particularly undesirable. It should also be clear that this is not only an issue of the presented example, but other spatially skewed datasets and larger processing core counts as well.

Therefore, we employ a cost heuristic to determine a more balanced data space subdivision. For this, we exploit the computation complexity properties of the cell neighborhood

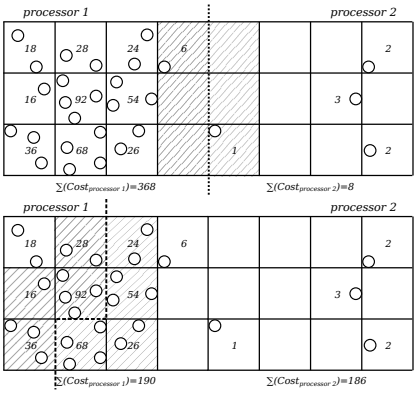

Figure 4: Impact of naive and heuristic-based hypergrid decompositions on compute load balancing. Halo cells are marked with a hatched pattern.

query. For each data item we have to perform n computations of the distance function *dist*, where n is the number of data items in the cell neighborhood. Since we have to do that for all m data items within a cell, the total number of comparisons for one cell is $n * m$. The sum of all comparisons, i.e. the cost scores, for all cells gives us the total "difficulty" of the whole clustering problem, at least in terms of the *dist* function evaluations. Then, we can assign to each parallel processor a consecutive chunk of cells, the cost of which is about a p-th part of the total score with p being the number of available parallel processing cores. The formal definitions are as follows.

Definition 10. Cell cost—The cell cost $Cost_{Cell}(c)$ of a cell c is the product of the number of items in it multiplied with the number of data points in the cell neighborhood – $Cost_{Cell}(c) = |c| * |N_{Cell}(c)|$.

Definition 11. Total cost—The total cost $Cost_{Total}$ is equal to the sum – $\sum_{c \in Cells} Cost_{Cell}(c)$ – of all individual cells.

Since the data items are already pre-sorted due to the spatial preprocessing step, the hypergrid subdivision can be performed by iteratively accumulating cell cost scores until the per-core threshold is reached or exceeded. Moreover, the the cell itself can be subdivided to gain more fine-grained control. For this, the cost score of the cell is not added entirely but in n-steps for each data item in the cell, where n is the number of items in the cell neighborhood. Figure 4 shows an example of a dataset, its overlayed hypergrid, the annotated cell cost values and the resulting subdivision. These subdivision can easily be computed in parallel by computing the cell score locally, reducing them to a global histogram and finally determining the boundaries according to the explained accumulative algorithm.

4.3 Local *DBSCAN*

Having redistributed the chunks among the compute nodes in a balanced fashion, the local *DBSCAN* execution follows. To break the need for sequential computation, implied by the recursive cluster expansion, this stage is converted to a parallelizable version with a single loop iterating over all data points. This enables the further, fine-grained parallelization of the algorithms using shared-memory parallelization approaches such as threads for example. The performance-focused algorithm redesign is twofold at this stage. Besides the parallelization of the iterations, the amount of computation per iteration is also minimized. Due to the cell-wise sorting and indexing of data points within the local data chunk, all points occupying one cell are stored consecutively in memory. This ensures that each cell-neighborhood must be computed at most once per thread, as each of them can be cached until all queries from within the same cell are visited. Listing 2 presents the pseudocode of the converted, iterative local *DBSCAN*.

```
1  def localDBSCAN(X, eps, minPts):
2    rules = Rules()
3    @parallel
4    for p in X:
5      Cp, Np = query(p, X, eps)
6      if length(Np) >= minPts:
7        markAsCore(p)
8        add(Cp, p)
9        for q in Np:
10         Cq = getCluster(q)
11         if isCore(q):
12           markAsSame(rules, Cp, Cq)
13         add(Cp, q)
14     elif notVisited(p):
15       markAsNoise(p)
16   return rules
```

Listing 2: Local *DBSCAN* pseudocode

For each of the points the epsilon neighborhood query is performed independently, i.e., not as an recursive expansion. When a query for a point p returns more than $minPoints$ data points, from which none is yet labeled, p is marked as a core of a cluster. The newly created cluster is then labeled using p's data point index, which is globally unique for the entire sorted dataset. If the epsilon neighborhood, numbering at least $minPoints$, contains a point q that is already assigned to a cluster, the point p is added to that cluster and inherits the cluster label from q. In case of multiple cluster labels present in the neighborhood, the core p inherits any one of the cluster labels and notes information indicating that each of the encountered subclusters actually are one, as they are inherently density connected. That information is vital to formulate merger rules for the subsequent merging of local cluster fragments and unification of cluster labels in the global scope (see section 4.4).

In all of the above cases, the remainder of non-core points in the epsilon neighborhood, which may also include halo area points, is added to the cluster of p. If p has less than $minPoints$ data points in its neighborhood, it is marked as visited and labeled as noise. The below proof shows that replacing the iterative cluster relabeling is equivalent to the original recursive expansion.

Theorem 1. Given points $p \in C_p$ and $q \in C_q$: $(Core(p) \vee Core(q)) \wedge DDR(p,q) \implies \exists C : C_p \bigcup C_q \subseteq C \wedge p, q \in C$

Proof. If neither p nor q is core, or they are mutually not DDR, the assumption is false and the implication trivially true. If p, q or both are cores, and they are DDR then by definition, they are also DR and therefore DC, with the linking point r being either p or q. Given the density connection DC between p and q, they belong to the same cluster C. By extension, any point belonging to C_p or C_q also belongs to C. □

The result of local *DBSCAN* is a list of subclusters along with the points and cores they contain, a list of noise points, and a set of rules describing which cluster labels are equivalent. This information is necessary and sufficient for the next step of merging the overlapping clusters with contradictory labels within the nodes' halos.

4.4 Rule-based cluster merging

The relabeling rules created by distinctive nodes are insufficient for merging cluster fragments from separate dataset chunks. The label-mapping rules across different nodes are created based on the labels of halo points. Upon the completion of the local *DBSCAN*, each halo zone is passed to the node that owns the actual neighboring data chunk. Subsequently, the comparison of local and halo point labels follows, resulting analogously in a set of relabeling rules for neighboring chunks, which may create transitive cluster label chains. These rules are then serialized and broadcast to all other nodes. Only then is the minimization of all local and inter-chunk label-mapping rules possible, and all transitive labels can be removed. Thus each compute node is equipped with a list of direct mappings from each existing subcluster label to a single global cluster label.

Each compute node then proceeds to relabel the owned clusters using the merger rules. At that stage each data point, now having assigned a cluster label, is sent back to the compute node that originally loaded it from the dataset. Recreation of the order of all data points is enabled by the initial ordering information created during the data redistribution phase. The distributed *HPDBSCAN* execution is complete and the result is a list of cluster ids or noise markers per data item.

5. IMPLEMENTATION

In this section we present our prototypical realization of *HPDBSCAN* and specifics of distinct technical details. The C++ source code can be obtained freely from our source code repository [23]. It depends on the parallel programing APIs Open Multiprocessing (OpenMP) [8] in version 4.0+ and Message Passing Interface (MPI) [15] in version 1.1+. Additionally, the command-line version requires the I/O library Hierarchical Data Format 5 (HDF5) [18] in order to pass the data and store computational results.

5.1 Data distribution and gathering

As explained in section 4.1, the data items of the datasets are redistributed in the preprocessing step, in order to achieve data locality. Implementing this behavior in shared-memory architectures is trivially not required, due to the fact that all processors can access the same memory. For distributed environments, however, this step is needed and can be quite challenging to realize—especially in a scalable fashion.

Since *HPDBSCAN* sorts the data points during the indexing phase and lays them out consecutively in memory, we are able to exploit collective communication operations of the MPI. We first send the local histograms of data points from each compute node to the one that owns the respective bounds during the local *DBSCAN* execution. This can be implemented either by an `MPI_Reduce` or, alternatively, by an `MPI_Alltoall` and a subsequent summation of the array. After that, each of the compute nodes allocates local memory, and the actual data points are exchanged using an `MPI_Alltoallv` call. Using the received histograms, the compute nodes are also able to memorize the initial ordering of the data points, in a flat array, for example.

Vice versa, the gather step can be implemented analogously. Instead of sorting the local data items by their assigned grid cell, they are now re-organized by their initial global position in the dataset. After that, they can be exchanged again, using the MPI collectives, and stored. Note that in this step the computed cluster labels are transferred along with the data points in order to avoid multiple communication.

5.2 Lock-free cluster labeling

To ensure that the cluster labels are unique within a chunk as well as globally, each cluster label c is determined by the lowest index of a core point inside a cluster—$c = \min_{p \in C | Core(p)} index(p)$. The $index(p)$ function returns the position of a data point p, within the globally sorted dataset, redistributed to the compute nodes. Additionally to ensuring global uniqueness, this mechanism also maximizes the size of consistently labeled cluster fragments within the same compute node, as each consecutive iteration over the points increments the current point's index. Whenever a core is found in the epsilon neighborhood, the current point inherits its cluster label, even if it is a core itself.

A data race may occur, when the current epsilon neighborhoods of multiple parallel threads overlap. In that case each thread may attempt to assign a label to a point within their neighborhood intersection. The naïve approach of locking the data structures storing the cluster label and core information is not scalable.

The better alternative of using atomic operations, here atomic *min*, requires encoding the values to operate on, with a single native data type. For this, we use signed long integer type values, and compress all flags and labels described by *DBSCAN*'s original definition, i.e., "visited", "core" and "noise" flags, and a "cluster label", to that data type. As the iterations are performed for each data point exactly once, the "visited" flag, is made redundant and abandoned. The cluster label value is stored using the absolute value of the lowest core point index it contains. The sign bit is used to encode the "core" flag, such that each core of cluster c is marked by value -c, and each non-core point—by value c. As cluster labels are created using point indexes, their value never exceeds $|X|$. The noise label can then be encoded using any value from outside the range $[-|X|, |X|]$. For this, we have selected the upper bound of the value range—the maximal positive signed long integer. As long as $range(signed\_long\_int) >= |X| + 1$, signed long integers are sufficient to encode the cluster labels as well as the core and noise flags. In that way, minimizing the cluster label is possible via simple atomic min implementation to set the cluster label and core flag at once. Some processor architectures, e.g., Intel x86, do not provide an atomic *min* instruction. Instead, a spinlock realization using basic atomic read and compare-and-swap instruction, as shown in Listing 3, is used.

```
1 def atomicMin(address, val):
2   prev = atomicRead(address)
3   while(prev > val):
4     swapped = CAS(address, prev, val)
5     if swapped: break
6     prev = atomicRead(address)
```

Listing 3: Spinlock atomic *min*

5.3 Parallelization of the local *DBSCAN* loop

The iterative conversion of *DBSCAN* allows us to divide the computation of the loop iterations among all threads of a compute node. Because the density of data points within a chunk can be highly skewed, a naive chunking approach is suboptimal (see Section 4.2), and can lead to a highly unbalanced work load. To mitigate this, a work stealing approach is advisable. Our *HPDBSCAN* implements threading using OpenMP's `parallel for` pragma. The closest representative of work stealing in OpenMP is the `schedule(dynamic)` clause, added to the `parallel for` pragma. Optimal performance is achieved, when the dynamically pulled workload is small enough—so that the workload imbalances are split and fairly divided, and at the same time large enough—so that not too many atomic *min* operations (whether supported by hardware or not) are performed simultaneously on the same memory location. This number is highly dependent on environment details, such as the clustered problem and the execution hardware. Through empirical tests, however, we determined a reasonable dynamic chunk size of 40.

6. EXPERIMENTAL EVALUATION

In this section we will describe the methodology and findings of the experiments conducted to evaluate the parallel *DBSCAN* approach described above. The main focus of the investigation is the performance evaluation of the implementation with respect to computation time, memory consumption and the parallel programming metrics: speed- and scale-up [12].

6.1 Hardware setup

To verify the computation time and speed up of our implementation, we have performed tests on the Juelich Dedicated Graphic Environment (JUDGE) [10]. It consists of 206 IBM System x iDataPlex dx360 M3 compute nodes, where each node has 12 compute cores combined through two Intel Xeon X5650 (Westmere) hex-core processors clocked at 2.66 GHz. A compute node has 96 GB of DDR-2 main memory. JUDGE is connected to a parallel, network-attached GPFS-storage system, called Juelich Storage Cluster (JUST) [11]. Even though the system has a total core count of 2.472, we were only able to acquire a maximum of 64 nodes (768 cores) for our benchmark, as JUDGE is used as a production cluster for other scientific applications. Our hardware allocation, though, was solely dedicated for us, which ensured that no other computations interfered with our tests. The plugged-in Westmere processors allow to use 24 virtual processors, when hyperthreading is enabled. For the test runs, however, we disabled this feature as it can falsify or destabilize measurement correctness, as Leng et al. [22] have shown. In a multithreading scenario we facilitate for this reason a maximum of 12 threads per node.

6.2 Software setup

The operation system running on JUDGE is a SUSE Linux SLES 11 with the kernel version 2.6.32.59-0.7. All applications in the test have been compiled with gcc 4.9.2 using the optimization level O3. The MPI distribution on JUDGE is MPICH2 in version 1.2.1p1. For the compilation of *HPDBSCAN*, a working HDF5 development library including headers and C++ bindings is required. For our benchmarks we used the HDF group's reference implementation, version 1.8.14, pre-installed on JUDGE. Later in this section we present a comparison of *HPDBSCAN* with *PDSDBSCAN-D* created by Patwari et al [25]. The latter needs the parallel netCDF I/O library. We have obtained and compiled pnetCDF from the project's web page at Northwestern University with version 1.5.0 [24].

Dataset	Points	Dims.	Size (MB)	ε	$minPts$
Tweets [t]	16,602,137	2	253.34	0.01	40
Twitter small [ts]	3,704,351	2	56.52	0.01	40
Bremen [b]	81,398,810	3	1863.68	100	10000
Bremen small [bs]	2,543,712	3	48.51	100	312

Table 1: *HPDBSCAN* benchmark datasets properties

6.3 Datasets

Despite *DBSCAN*'s popularity its parallelization attempts were mainly evaluated using synthetic datasets. To their advantage, they can provide an arbitrarily large number of data points and dimensionality. The downside, however, is that they are not representative for actual real world applications. They might have inherent regular patterns from, e.g., pseudo random number generators that will silently bias the implementation's performance. For this reason, we decided to resort to actual real-world data and its potential skew. An overview of the chosen examples is depicted in Table 1. We acknowledge that an evaluation of higher-dimensional datasets is of great interest for some clustering application, such as, for instance, genomics in bio-informatics, but could not be obtained at the time of writing.

6.3.1 Geo-tagged collection of tweets

This set was collected and made available to us by Junjun Yin form the National Center for Supercomputing Application (NCSA). The dataset was obtained using the free twitter streaming API and contains exactly one percent of all geo-tagged tweets from the United Kingdom in June 2014. It was initially created to investigate the possibility of mining people's trajectories and to identify hotspots and points of interest (clusters of people) through monitoring tweet density. The full collection spans roughly 16.6 million tweets. A smaller subset of this was generated by filtering the entire set for the first week of June only. Both datasets are available at the scientific storage and sharing platform B2SHARE [14].

6.3.2 Point cloud of Bremen's old town

This data was collected and made available by Dorit Borrmann and Andreas Nüchter from the Institute of Computer Science at the Jacobs University Bremen, Germany. It is a 3D-point cloud of the old town of Bremen. A point cloud is a set of points and its representing coordinate system that often model the surface of objects. This particular point cloud of Bremen was recorded using a laser scanner system mounted onto an autonomous robotic vehicle. It has stopped at 11 different locations, performing each time a full 360° scan of the surrounding area. Given the GPS triangulated position and perspective of the camera, the subpoint clouds were combined to one monolith. The raw data is available from Borrmann and Nüchter's webpage [20]. An already combined version in HDF5 format, created by us, can be obtained from B2SHARE [14]. *DBSCAN* can be applied here in order to clean the dataset from noise or outliers, such as falsy scans or unwanted reflections of moving

objects. Moreover, *DBSCAN* can also be used to find distinct objects, represented as clusters, in the point cloud like houses, roads or people. The whole point cloud contains roughly 81.3 million data points. A smaller variant was generated by randomly sampling $\frac{1}{32}$ of the points that is also available on B2SHARE [14].

6.4 Speed up evaluation of *HPDBSCAN*

We benchmark our *HPDBSCAN* application's speed up using both, the full Twitter (t) and the full Bremen (b) dataset. Our principal methodological approach is thereby as follows. Each benchmark is ran five times, measuring the application's walltime at the beginning and end of the `main()` function of the process with the MPI rank 0 and the OpenMP thread number 0. After these five runs we double the number of nodes and cores, starting from one node and 12 cores, up to the maximum of 768 cores. In addition to that we have run a base measurement with exactly one core on one node. For each "five-pack" benchmark run we report the minimum, maximum, mean μ, standard deviation σ and coefficient of variation (CV), defined as $\nu = \frac{\mu}{\sigma}$ [1]. The speed up coefficient is calculated in comparison to the single core run, based on the mean values of the measurements for each processor count configuration. Both datasets are processed using the OpenMP/MPI hybrid features of our application. That means that we spawn an MPI process for each node available and parallelize locally on the nodes using OpenMP. For the Bremen point cloud we have additionally parallelized the computation with MPI alone, i.e., we use one MPI process per core, enabling direct comparison of the hybrid and fully distributed versions.

Nodes			1	1	2	...	32	64
Cores			1	12	24		384	768
OpenMP+MPI hybrid b								
time (s)	**Mean**	μ	**79372.29**	**8037.71**	**4271.64**		**327.07**	**172.53**
	StDev	σ	17.6011	71.2829	16.2092		2.5971	1.3801
	CV	υ	0.00022	0.00886	0.00379		0.0079	0.0079
	Min		79342.08	7937.48	4253.45		322.71	170.77
	Max		79385.57	8129.85	4293.86		329.65	174.47
Speed-Up			**1**	**9.9**	**18.6**		**242.7**	**460.0**
MPI b								
time (s)	**Mean**	μ	**79372.29**	**8028.67**	**4403.96**		**515.21**	**354.99**
	SrDev	σ	17.6012	9.5769	7.1526		94.7806	42.0006
	CV	υ	0.00022	0.00119	0.00162		0.18396	0.11832
	Min		79342.08	8019.10	4395.78		471.10	302.27
	Max		79385.57	8040.83	4415.45		684.74	420.01
Speed-Up			**1**	**9.9**	**18.0**		**154.1**	**232.7**
OpenMP+MPI hybrid t								
time (s)	**Mean**	μ	**2079.26**	**212.77**	**115.66**		**10.04**	**7.88**
	StDev	σ	1.06455	0.56826	0.35893		0.42128	1.03302
	CV	υ	0.00051	0.00267	0.00310		0.04194	0.13106
	Min		2078.16	212.05	115.34		9.76	7.14
	Max		2080.47	213.43	116.17		10.78	9.70
Speed-Up			**1**	**9.8**	**18.0**		**207.0**	**263.8**

Table 2: Measured and calculated values of the *HPDBSCAN* speed-up evaluation

The results in Table 2 and Figure 5 show that we are able to gain substantial speed up for both data sets. It peaks for Bremen at 460.0 using 768 cores, and in the Twitter analysis case at slightly more than half of this value at 263.8. For the MPI-only clustering of the Bremen dataset the speed up value falls short of the hybrid implementation, being only roughly half of it with 232.7 using 768 cores. There are two noteworthy facts that can be observed in the measurement data. The first and obvious one is that the hybrid implementation outperforms the fully distributed MPI runs by a factor of two. The access to a shared cell index and the reduced number of nodes to communicate it to, significantly reduces communication overhead and enables faster processing time. Secondly, one can observe a steady decrease in the efficiency of additional cores used for the clustering. This seems to be especially true for the tweet collections compares to the Bremen dataset. This observation can be explained best through Amdahl's law [2]. In the benchmark we use a constant problem size, disallowing infinite speed up performance gains. Instead, we approach the asymptote of the single threaded program parts. Due to the fact that tweet collection is smaller in size, we approach this boundary earlier than with the Bremen data for instance. Additional network communication overhead with larger processor counts, `atomicMin()` clashes as well as load imbalances are good examples of simultaneously growing serial program parts. Moreover, the growing CV value is a good indicator for the increasing influence of external factors onto the measurements, like varying operating system scheduling.

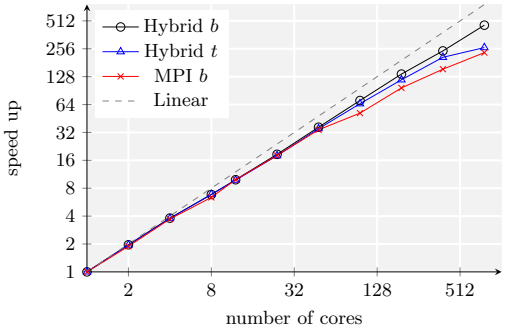

Figure 5: Speed up curves of the *HPDBSCAN* application analyzing the Bremen and Twitter datasets

6.5 Scale up evaluation of *HPDBSCAN*

In this section we investigate *HPDBSCAN*'s scalability properties. Our principal measuring methodology remains unchanged. Instead of the speed up coefficient, we report the efficiency value $e_p = \frac{t_1}{t_p}$ for each benchmark run, which is the fraction of the execution time with a single core and the execution time with p processing cores. Perfect scalability is achieved, when the efficiency equals one or is almost one. Yet, it requires doubling the dataset size, whenever we double the processor count.

As a base for this we use the small Bremen dataset bs. In particular, for each run, we copy the entire data p times, where p is equal to the number of used processors. Then, each copy is shifted along the first axis of the dataset by the copy's index, times the axis range, and concatenated with the others. This way, we get multiple (p) Bremen old towns next to one another. We chose this approach to get a better grasp of the overhead of our implementation by presenting the same problem to each available MPI process. In contrast to that, a random sampling of the whole Bremen dataset, for instance, would have altered the problem difficulty.

The results of our test can be seen in Table 3 and Figure 6. In all of the three scenarios a near constant efficiency

Cores			1	2	4	8	16	32
OpenMP								
time (s)	**Mean**	μ	**99.5044**	**102.644**	**107.462**	**121.35**	**-**	**-**
	StDev	σ	0.0991	0.0439	0.1515	5.6788	-	-
	CV	v	0.00099	0.00042	0.00141	0.04679	-	-
	Min		99.41	102.58	107.2	117.81	-	-
	Max		99.66	102.69	107.59	131.45	-	-
MPI								
time (s)	**Mean**	μ	**99.50**	**101.86**	**103.74**	**105.48**	**107.12**	**109.19**
	StDev	σ	0.0991	0.0884	0.1131	0.1881	0.5399	0.6653
	CV	v	0.00099	0.00086	0.00109	0.00178	0.00504	0.00609
	Min		99.41	101.79	103.65	105.3	106.56	108.52
	Max		99.66	102	103.91	105.76	107.77	110.15

Cores			12	24	48	96	192	384
OpenMP/MPI hybrid								
time (s)	**Mean**	μ	**10.46**	**13.3**	**14.02**	**14.786**	**16.31**	**19.76**
	StDev	σ	0.0974	0.1425	0.2420	0.2548	0.58307	3.15081
	CV	v	0.00931	0.01071	0.24207	0.25481	0.03578	0.15943
	Min		10.38	13.12	13.76	14.57	15.67	17.909
	Max		10.63	13.45	14.28	15.16	17.12	25.26

Table 3: Measured and calculated values of the *HPDBSCAN* scale-up evaluation of the Bremen data

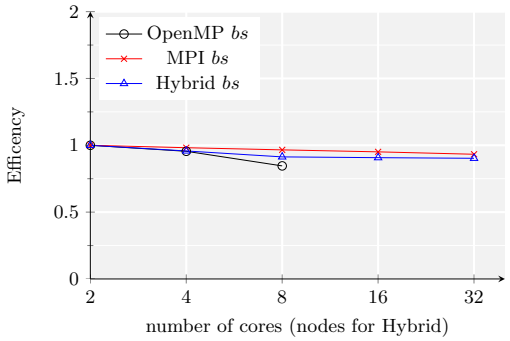

Figure 6: Scale up curves of *HPDBSCAN* analyzing the Bremen and Twitter datasets

value can be achieved, indicating good scalability. While the MPI-only and OpenMP/MPI hybrid benchmark runs only have a slightly increasing execution time curve, we can observe a clear peek for the OpenMP benchmark with four and more cores. Through a separate test, we can attribute this increase to more contentions in the spinlock of our `atomicMin()` implementation, introduced in Section 5.

6.6 Comparison of *HPDBSCAN* and *PDSDBSCAN*

As discussed in related work there are a number of other parallel versions of *DBSCAN*. Most of them report different value permutations for the computation time, memory consumption, speed up and scalability of their implementations. Almost all of them do not to provide either their used benchmark datasets or the source code of their implementations. This in turn, disallows us to verify their results or to perform a direct comparison of our approaches. To the best of our knowledge the only exception are Patwary et al. [25]. Their datasets can also not be recreated, but they made the C++ implementation of their parallel disjoint-set data structure *DBSCAN—PDSDBSCAN*—open-source. This allows us to compare our approach with theirs at least using our benchmark datasets. Patwary et al. offer two versions of *PDSDBSCAN*. One targets shared-memory architectures only and is based on OpenMP. The other can also be used in distributed environments and is implemented using MPI. In order to distinguish between these two, the suffixes -S or -D are added respectively.

In order to compare both parallel *DBSCAN* approaches, we have performed another speed up benchmark according to the introduced methodology on the small Twitter dataset. Due to their technical similarities our two "contestants" are *HPDBSCAN* using MPI processes only and *PDSDBSCAN-D*. Thereby, we scale the process count from one to a maximum of 32, each being executed on a separate compute node. Even though we have executed five runs for each level of used processors, we report here only the mean value for execution time, memory consumption and speed up, because of space considerations.

Nodes	1	2	4	8	16	32
HPDBSCAN MPI						
time (s)	114.39	58.99	30.14	15.71	8.37	6.07
Speed-Up	1.00	1.94	3.80	7.28	13.67	18.85
Memory (MB)	251064	345276	433340	678248	1101000	2111000
PDSDBSCAN-D						
time (s)	288.35	162.47	105.94	89.87	85.37	88.42
Speed-Up	1	1.77	2.72	3.21	3.38	3.36
Memory (MB)	500512	725104	1370000	4954000	19724000	59685000

Table 4: Comparison of *HPDBSCAN* and *PDSDBSCAN − D* using the Twitter dataset

Table 4 and Figure 7 present the obtained results. *HPDBSCAN* shows a constant, near linear speed-up curve, whereas *PDSDBSCAN-D* starts similarly, but soon flattens, stabilizing at a speed-up of around 3.5. The curve for the memory consumption is inverse. *HPDBSCAN* shows a linear increase again, seemingly being dependent on the number of used processing cores, which can be explained by the larger number of replicated points in the halo areas. *PDSDBSCAN-D*, however, presents an exponential memory consumption. An investigation of the source code reveals that each MPI process always loads the entire datafile into main memory, effectively limiting its capabilities to scale with larger datasets. This is also the reason why we have used the small Twitter dataset *ts* for this experiment, as larger datasets have caused out-of-memory exceptions. As a consequence, we have not been able to reproduce the performance capabilities of *PDSDBSCAN-D*.

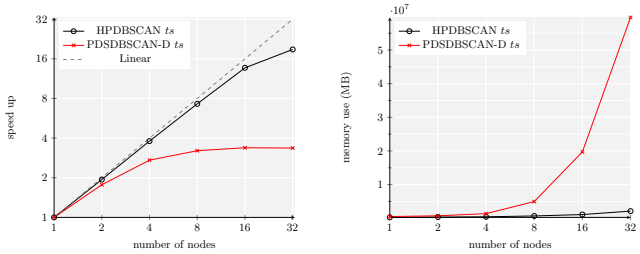

Figure 7: Speed up and memory usage of *HPDBSCAN* compared to *PDSDBSCAN−D*

7. CONCLUSION

In this paper, we have presented *HPDBSCAN*—a scalable version of the density-based clustering algorithm *DB-*

SCAN. We have overcome the algorithm's inherent sequential control flow dependencies through a divide-and-conquer approach, using techniques from cell-based clustering algorithms. Specifically, we employ a regular hypergrid as the spatial index in order to minimize the neighborhood-search spaces and to partition the entire cluster analysis into local subtasks, without requiring further communication. Using a rule-based merging scheme, we combine the found local cluster-labels into a global view. In addition to that, we also proposes a cost heuristic that allows to balance the computation workload, facilitated by the previously mentioned cells, divided among the compute nodes according to their computation complexity. We have implemented *HPDBSCAN* as an open-source OpenMP/MPI hybrid application in C++, which can be deployed in shared-memory as well as distributed-memory computing environments. Our experimental evaluation of the application has proven the algorithm's scalability in terms of memory consumption and computation time, outperforming *PDSDBSCAN*, the first parallel HPC implementation. The presented cell-based spatial index can easily be transferred to other clustering and neighborhood-search problems with constant search range. In future work we plan to demonstrate this on the basis of parallelizing other clustering algorithms, such as *OPTICS* and *SUBCLU*.

8. REFERENCES

[1] H. Abdi. Coefficient of variation. *Encyclopedia of research design*, pages 169–171, 2010.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the spring joint computer conference 1967*, pages 483–485. ACM, 1967.

[3] D. Arlia and M. Coppola. Experiments in parallel clustering with dbscan. In *Euro-Par 2001 Parallel Processing*, pages 326–331. Springer, 2001.

[4] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409, 1979.

[5] S. Brecheisen, H.-P. Kriegel, and M. Pfeifle. Parallel density-based clustering of complex objects. In *Advances in Knowledge Discovery and Data Mining*, pages 179–188. Springer, 2006.

[6] C. Cantrell. *Modern mathematical methods for physicists and engineers*. CUP, 2000.

[7] M. Chen, X. Gao, and H. Li. Parallel DBSCAN with Priority R-Tree. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference*, pages 508–511. IEEE, 2010.

[8] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[9] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[10] Forschungszentrum Jülich GmbH. Juelich Dedicated GPU Environment. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUDGE/JUDGE_node.html`.

[11] Forschungszentrum Jülich GmbH. Juelich Storage Cluster. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Datamanagement/OnlineStorage/JUST/JUST_node.html`.

[12] I. Foster. *Designing and building parallel programs*. Addison Wesley Publishing Company, 1995.

[13] Y. X. Fu, W. Z. Zhao, and H. F. Ma. Research on parallel dbscan algorithm design based on mapreduce. *Advanced Materials Research*, 301:1133–1138, 2011.

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

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[16] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel, et al. The universe at extreme scale: multi-petaflop sky simulation on the bg/q. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 4. IEEE Computer Society Press, 2012.

[17] J. Han, M. Kamber, and J. Pei. *Data Mining: concepts and techniques - 3rd ed*. Morgan Kaufmann, 2011.

[18] HDF Group. Hierachical Data Format 5. `http://www.hdfgroup.org/HDF5`.

[19] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. MR-DBSCAN: an efficient parallel density-based clustering algorithm using mapreduce. In *Parallel and Distributed Systems, 2011 IEEE 17th International Conference*, pages 473–480, 2011.

[20] Jacobs University Bremen. 3D Scan Repository. `http://kos.informatik.uni-osnabrueck.de/3Dscans/`.

[21] A. Knöpfel, B. Gröne, and P. Tabeling. *Fundamental modeling concepts*, volume 154. Wiley, UK, 2005.

[22] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 2002.

[23] Markus Götz. HPDBSCAN implementation. `https://bitbucket.org/markus.goetz/hpdbscan`.

[24] Northwestern University. Parallel netCDF. `http://cucis.ece.northwestern.edu/projects/PnetCDF/`.

[25] M. M. A. Patwary, D. Palsetia, A. Agrawal, et al. A new scalable parallel dbscan algorithm using the disjoint-set data structure. In *High Performance Computing, Networking, Storage and Analysis (SC), International Conference for*, pages 1–11. IEEE, 2012.

[26] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys. Trees or grids?: indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 236–245. ACM, 2009.

[27] X. XU, J. JAGER, and H.-P. KRIEGEL. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Mining and Knowledge Discovery*, 3:263–290, 1999.

[28] A. Zhou, S. Zhou, J. Cao, Y. Fan, and Y. Hu. Approaches for scaling dbscan algorithm to large spatial databases. *Journal of computer science and technology*, 15(6):509–526, 2000.