Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

Article

Heterogeneous Distributed Big Data Clustering on Sparse Grids

David Pfander [1], Gregor Daiß [2] and Dirk Pflüger [3]

1 University of Stuttgart; David.Pfander@ipvs.uni-stuttgart.de
2 University of Stuttgart; Gregor.Daiss@ipvs.uni-stuttgart.de
3 University of Stuttgart; Dirk.Pflueger@ipvs.uni-stuttgart.de

Version February 1, 2019 submitted to Preprints

Abstract: Clustering is an important task in data mining that has become more challenging due to the ever-increasing size of available datasets. To cope with these big data scenarios, a high-performance clustering approach is required. Sparse grid clustering is a density-based clustering method that uses a sparse grid density estimation as its central building block. The underlying density estimation approach enables the detection of clusters with non-convex shapes and without a predetermined number of clusters. In this work, we introduce a new distributed and performance-portable variant of the sparse grid clustering algorithm that is suited for big data settings. Our compute kernels were implemented in OpenCL to enable portability across a wide range of architectures. For distributed environments, we added a manager-worker scheme that was implemented using MPI. In experiments on two supercomputers, Piz Daint and Hazel Hen, with up to 100 million data points in a 10-dimensional dataset, we show the performance and scalability of our approach. The dataset with 100 million data points was clustered in 1198 s using 128 nodes of Piz Daint. This translates to an overall performance of 352 TFLOPS. On the node-level, we provide results for two GPUs, Nvidia's Tesla P100 and the AMD FirePro W8100, and one processor-based platform that uses Intel Xeon E5-2680v3 processors. In these experiments, we achieved between 43% and 66% of the peak performance across all compute kernels and devices, demonstrating the performance portability of our approach.

Keywords: clustering; machine learning; distributed computing; performance portability; GPGPU; OpenCL; peak performance

---

1. Introduction

In data mining, cluster analysis partitions a dataset according to a given measure of similarity. The partitions obtained as a result of the clustering process are called clusters. The clustering of big datasets poses additional challenges as not all clustering algorithms scale well in the size of the dataset. Furthermore, mapping clustering approaches to modern hardware platforms such as graphics processing units (GPUs) requires new parallel approaches. And for the use on supercomputers or in the cloud, algorithms need to be designed for distributed computing.

There is a wide range of algorithms that perform clustering. The classic k-means algorithm iteratively improves an initial guess of cluster centers [1]. Efficient variants of the k-means algorithm have been proposed, *e.g.* by using domain partitioning through k-d-trees [2] or by a more careful selection of the initial cluster centers [3]. As a major disadvantage, k-means requires the number of clusters to be known in advance, which is not always possible. Moreover, in contrast to many alternatives k-means cannot detect clusters with non-convex shape.

DBSCAN probably is the most widely-used density-based clustering approach [4]. In its basic form, it constructs a cluster based on the number of data points in an ϵ-sphere around each data point. If spheres overlap and have enough data points in them, the data points are part of the same cluster. For m data points, the complexity of DBSCAN was stated as $\mathcal{O}(m \log m)$ in the original paper [4]. However, more recent work shows that the actual complexity has a lower bound of $\Omega(m^{4/3})$ [5,6].

© 2019 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

DENCLUE is another example for clustering based on density estimation. It uses a kernel density estimation algorithm [7]. Spectral clustering methods cluster datasets by solving a mincut problem on a weighted neighborhood graph [8]. There are many more approaches to clustering, *e.g.* using neural networks [9], described in the literature [1,10].

Some clustering algorithms support GPUs for higher performance. Takizawa and Kobayashi presented a distributed and GPU-accelerated k-means implementation in 2006, before modern GPGPU frameworks like CUDA and OpenCL were available [11]. Since then, further GPU-enabled k-means algorithms have been developed [12–15]. Fewer published results are available for density-based GPU-accelerated clustering. CUDA-DClust is a GPU-accelerated variant of DBSCAN that uses an indexing approach to reduce distance calculations [16]. Andrade *et al.* developed a GPU-accelerated variant of DBSCAN called G-DBSCAN employing an algorithm with quadratic complexity in the dataset size [17].

While many clustering algorithms have been proposed, not many have been shown to work in big data scenarios. k-means++ is a map-reduce variant of the k-mean algorithm that has been used to cluster a 4.8 million data points dataset on a Hadoop cluster with 1968 nodes [18]. MR-DBSCAN is a DBSCAN variant that could cluster a 2d dataset with up to 1.9 billion data points and is implemented with a map-reduce approach as well [19]. The published results of MR-DBSCAN demonstrate excellent performance. However, it uses a grid discretization that makes assumptions on the distribution of the dataset throughout the domain. Furthermore, it is unclear how the algorithm will scale to higher dimensions, as the grid discretization is fully affected by the curse of dimensionality [20]. RP-DBSCAN implements a similar approach compared to MR-DBSCAN, but uses a more advanced partitioning scheme [5]. RP-DBSCAN was able cluster a 13d dataset with 4.4 billion data points.

In this work, we introduce a new distributed and performance-portable variant of the sparse grid clustering algorithm. This approach builds upon prior work which presented the basic theory and compared our approach to other clustering strategies [21].

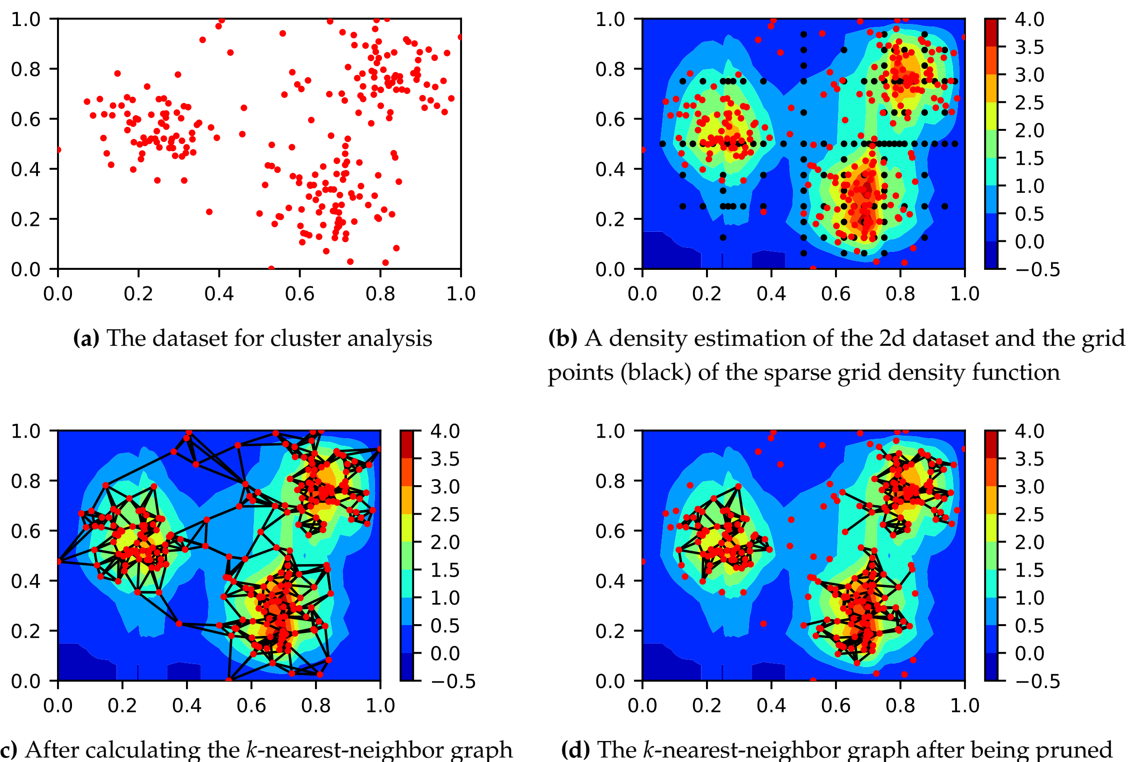
The unique building block of our approach is the sparse grid density estimation algorithm. Sparse grids are a method for spatial discretization that has been applied to higher-dimensional settings with up to 166 dimensions and moderate intrinsic dimensionality [22]. Therefore, in contrast to many spatial partitioning approaches, the underlying sparse grid density estimation mitigates the curse of dimensionality. The sparse grid clustering method does not rely on assumptions about the distribution of data, it can successfully suppress noise and it can detect clusters of non-convex shape. Compared to k-means, sparse grid clustering does not require the number of clusters as a parameter of the algorithm. In this paper, we use the Euclidean norm as the measure for the similarity of data points.

Methods based on sparse grids have been used for regression and classification tasks [22,23]. In prior work, we have shown the applicability of these methods in heterogeneous computing and high-performance computing settings [24–26]. However, to our knowledge this work presents the first high-performance results for sparse grid clustering.

Our algorithm was designed with a focus on high performance and performance portability. On the node-level we use OpenCL, as it offers basic portability across a wide range of hardware platforms. We not only support GPUs and processors of different vendors, our approach achieves a major fraction of the peak performance of all devices used. To map our method to clusters and supercomputers, we implemented a distributed manager-worker scheme. Due to the underlying method and the high-performance distributed approach, we show that sparse grid clustering is well-suited for large datasets. We provide results for 10d datasets with up to 100 million data points and 100 clusters.

The remainder of this paper is structured as follows. In Sec. 2, we give an overview of the sparse grid clustering algorithm. Then, in Sec. 3, we introduce the sparse grid density estimation as our core component. The other components of the algorithm are introduced in Sec. 4. Section 5 describes the parallel and distributed implementation and discusses features of the algorithms from a high-performance perspective. We show results for three node-level hardware platforms and two

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060



(a) The dataset for cluster analysis

(b) A density estimation of the 2d dataset and the grid points (black) of the sparse grid density function

(c) After calculating the k-nearest-neighbor graph

(d) The k-nearest-neighbor graph after being pruned

Figure 1. The application of the sparse grid clustering algorithm to a 2d dataset with three slightly overlapping clusters. After calculating the sparse grid density estimation and the k-nearest-neighbor graph, the graph is pruned using the density estimation. This splits the graph into three connected components.

supercomputers in Sec. 6. Finally, in Sec. 7, we remark on implications of the presented algorithm and discuss future work.

2. Clustering on Sparse Grids

In this section, we describe the sparse grid clustering algorithm on a high level. We describe its components in Sec. 3 and 4 in more detail.

Sparse grid clustering assumes a d-dimensional dataset T with m data points that was normalized to the unit hypercube $[0,1]^d$:

$$T := \{\mathbf{x}_i \in [0,1]^d\}_{i=1}^{m}. \tag{1}$$

We further assume that the dataset has been randomized.

The sparse grid clustering algorithm is a four step algorithm. Except for the last one, these steps are shown in Fig. 1 at the example of a 2d dataset with three clusters. The sparse grid clustering algorithm first calculates a density estimation of the dataset using the sparse grid density estimation algorithm (Fig. 1b). Then, a k-nearest-neighbor graph of the dataset is computed (Fig. 1c). In the third step, the density estimation is used to prune the graph. The algorithm prunes nodes in low-density regions and edges that intersect low-density regions (Fig. 1d). In the fourth and final step the weakly-connected components of the pruned graph are retrieved. The connected components of the graph are returned as the detected clusters.

This description immediately suggests one of the tuning parameters of the algorithm. The sparse grid clustering method requires a carefully chosen threshold value t that is used for pruning the k-nearest-neighbor graph.

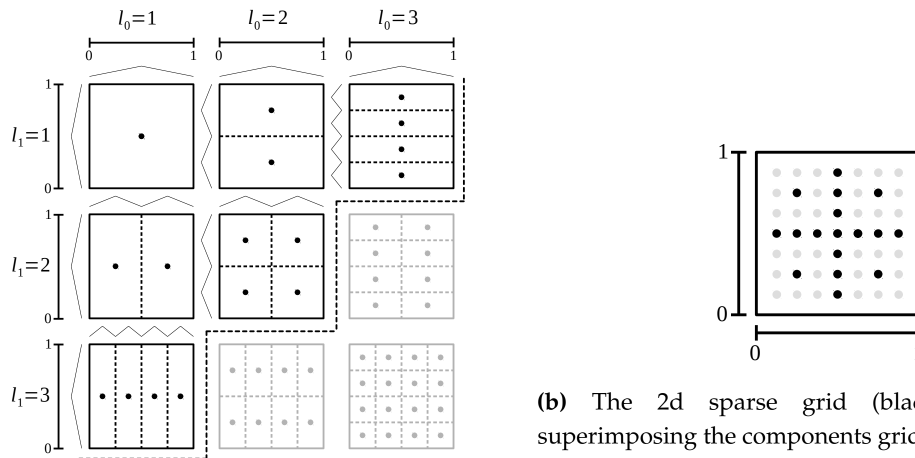
Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

(a) A 1d full grid in nodal basis

(b) A 1d sparse grid in the hierachical basis

Figure 2. The nodal and the sparse grid in Fig. 2a and Fig. 2b both have discretization level $l = 3$ and are equal for $d = 1$. Both use hat functions $\phi_{l,i}$ as basis functions, but in a nodal and in a hierarchical formulation. Note that sparse grids employ less grid points compared to full grids of the same level for d $\geq$ 2 (see Fig. 3).



(a) The subgrids of varying discretization level. Dotted lines outline the support of the basis functions centered at the grid points.

(b) The 2d sparse grid (black) obtained by superimposing the components grids.

Figure 3. The subgrids of a sparse grid with $l = 3$ (Fig. 3a) and the resulting sparse grid (Fig. 3b). Greyed out subgrids and grid points would be part of the corresponding full grid.

3. Estimating Densities on Sparse Grids

The sparse grid density estimation is build upon the concept of sparse grids. We therefore briefly introduce sparse grids and then describe how densities can be estimated with the sparse grid method.

3.1. Sparse Grids

As this work focuses on the sparse grid clustering algorithm and not on the basic sparse grid method, this introduction to sparse grids is necessarily brief. For a thorough presentation, we recommend the overview by Bungartz and Griebel [27].

A d-dimensional grid can be defined on the unit hypercube $[0,1]^d$ with an equidistant mesh width $h_n = 2^{-n}$ for a discretization level n and, therefore, 2nd grid points. With basis functions centered at the grid points, a corresponding function space is spanned by the linear combinations of the basis functions. We call this a full grid approach. Full grids can be represented in a hierarchical basis. From this representation, it is a small step to sparse grids.

The hierarchical approach constructs a final grid by superimposing a set of subgrids. First, we define an index set that is used to enumerate the grid points on the d-dimensional subgrids of discretization level $\mathbf{l} \in \mathbb{N}^d$:

$$I_{\mathbf{l}} := \{(i_1, \ldots, i_d) : 0 < i_k < 2^{l_k}, i_k \text{ odd}\}. \tag{2}$$

In this work, we employ hat functions as basis functions. The scaled and translated 1d hat functions are defined as

$$\phi_{l,i}(x) := \max(0, 1 - |2^l x - i|). \tag{3}$$

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1
Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

For $d > 1$, we use a tensor-product approach:

$$\phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) := \prod_{j=1}^{d} \phi_{l_j,i_j}(x_j). \tag{4}$$

Given an index set $I_\mathbf{l}$ and the basis functions $\phi_{\mathbf{l},\mathbf{i}}$, we can define the subspaces

$$W_\mathbf{l} := \text{span}\{\phi_{\mathbf{l},\mathbf{i}} : \mathbf{i} \in I_\mathbf{l}\}. \tag{5}$$

116 The subgrids and their grid points $x_{\mathbf{l},\mathbf{i}} := (i_1 h_{l_1}, \ldots, i_d h_{l_d})$ for the subspaces $W_{(1,1)} \ldots W_{(3,3)}$ are
117 displayed in Fig. 3a for a 2d grid.

With the direct sum $\bigoplus$, a full grid of discretization level $n \in \mathbb{N}$ in the hierarchical basis can be defined as

$$V_n := \bigoplus_{|\mathbf{l}|_\infty \leq n} W_\mathbf{l}. \tag{6}$$

118 Figure 2 shows how a 1d grid is represented in the standard (nodal) basis (Fig. 2a) and the hierarchical
119 basis (Fig. 2b).

Sparse grids are based on the observation that for sufficiently smooth functions only a small additional interpolation error is introduced if certain grid points are removed [27]. This mitigates the curse of dimensionality. As a result, the sparse grid function space $V_n^{(1)}$ is constructed from a different set of subspaces:

$$V_n^{(1)} := \bigoplus_{|\mathbf{l}|_1 \leq n+d-1} W_\mathbf{l}. \tag{7}$$

120 Figure 3 shows how a 2d sparse grid is constructed from subgrids that correspond to the subspaces of
121 the grid.

A sparse grid function $f \in V_n^{(1)}$ is given as

$$f(\mathbf{x}) = \sum_{|\mathbf{l}|_1 \leq n+d-1} \sum_{\mathbf{i} \in I_\mathbf{l}} \alpha_{\mathbf{l},\mathbf{i}} \phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) =: \sum_{j=1}^{N} \alpha_j \phi_j(\mathbf{x}), \tag{8}$$

122 where we sum up all N weighted basis functions in some order, and where N denotes the total number
123 of grid points. Since our algorithms iterate the basis functions linearly, we use the simplified notation
124 when the algorithms are presented. The coefficients $\alpha_{\mathbf{l},\mathbf{i}}$ are usually referred to as surpluses.

125 *3.2. The Sparse Grid Density Estimation*

The sparse grid density estimation, originally proposed by Hegland *et al.* [28], uses an initial density guess f_ϵ than is smoothed using a spline-smoothing approach:

$$\hat{f} = \arg\min_{u \in V} \int_\Omega (u(\mathbf{x}) - f_\epsilon(\mathbf{x}))^2 d\mathbf{x} + \lambda ||Lu||_{L_2}^2. \tag{9}$$

This approach results in a function $\hat{f} \in V$ that balances closeness to the initial density guess with the regularization term $||Lu||_{L_2}^2$ that enforces smoothness on the resulting density function. The regularization parameter λ controls the degree of smoothness of $\hat{f}$. L usually is some differential operator. We use the initial density guess proposed by Hegland *et al.* that places a Dirac delta function $\delta_{\mathbf{x}_i}$ at every data point $\mathbf{x}_i$:

$$f_\epsilon := \frac{1}{m} \sum_{i=1}^{m} \delta_{\mathbf{x}_i}. \tag{10}$$

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

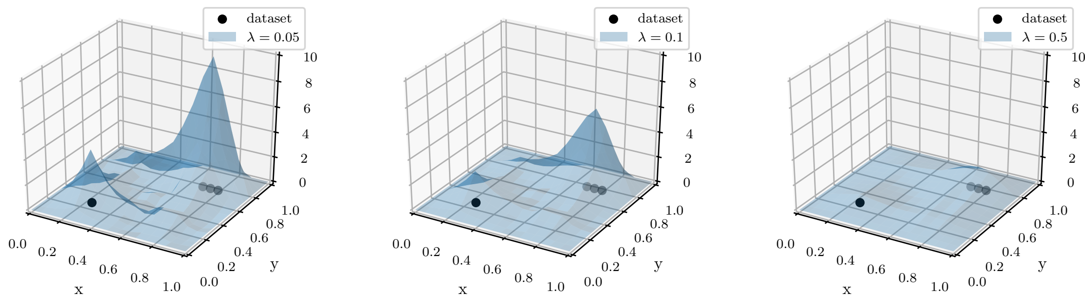


Figure 4. The effect of the regularization parameter λ on a 2d dataset with four data points. For smaller λ values, the function becomes more similar to the initial density guess of Dirac δ functions. The function becomes smoother for higher values of λ.

As in prior work, we compute the best sparse grid function $u \in V_n^{(1)}$ and use a surplus-based regularization approach [22]. Therefore, the problem to solve is

$$\hat{f} = \underset{u \in V_n^{(1)}}{\arg\min} \int_\Omega (u(\mathbf{x}) - f_\epsilon(\mathbf{x}))^2 d\mathbf{x} + \lambda \sum_{i=1}^{N} \alpha_i^2. \tag{11}$$

This formulation leads to a system of linear equations

$$(B + \lambda I)\boldsymbol{\alpha} = \mathbf{b}, \tag{12}$$

126 with $B_{ij} = (\phi_i, \phi_j)_{L_2}$, the identity matrix I and $b_i = \frac{1}{m} \sum_{j=1}^{m} \phi_i(\mathbf{x}_j)$.

We solve this system of linear equations with a conjugate gradient solver (CG). Given this iterative solver, two major operations need to be performed: calculating the right-hand side once and computing the matrix-vector product $\mathbf{v}' = (B + \lambda I)\mathbf{v}$ in every CG iteration. The calculation of the right-hand side is straightforward. However, the matrix-vector product requires efficient computations of the L_2 inner product of pairs of basis functions:

$$(\phi_{\mathbf{l},\mathbf{i}}, \phi_{\mathbf{l}',\mathbf{i}'})_{L_2} = \int_\Omega \phi_{\mathbf{l},\mathbf{i}}(\mathbf{x}) \phi_{\mathbf{l}',\mathbf{i}'}(\mathbf{x}) d\mathbf{x} \tag{13}$$

$$= \int_0^1 \phi_{l_1,i_1}(x_1) \phi_{l_1',i_1'}(x_1) dx_1 \cdots \int_0^1 \phi_{l_d,i_d}(x_d) \phi_{l_d',i_d'}(x_d) dx_d. \tag{14}$$

The 1d integrals can be computed directly:

$$\int_0^1 \phi_{l,i}(x) \phi_{l',i'}(x) dx = \begin{cases} \frac{2}{3} h_l, & x_{li} = x_{l'i'}, \\ h_{l'} \phi_{l,i}(x_{l'i'}) + h_l \phi_{l'i'}(x_{li}), & \text{else}. \end{cases} \tag{15}$$

127 We note that in many instances the integral will be zero due to the non-overlapping support of the hat
128 functions.

129 Figure 4 shows the effect of varying the regularization parameter λ for a 2d dataset. λ has to be
130 chosen with care, as too small values might split a single cluster into multiple clusters. On the other
131 hand, if λ is too large separate clusters could be part of the same high-density region.

132 *3.3. Streaming Algorithms for the Sparse Grid Density Estimation*

133 A high-performance sparse grid density estimation algorithm needs to efficiently compute the
134 two operations described above: the computations of the matrix-vector product $\mathbf{v}' = (B + \lambda I)\mathbf{v}$ and the
135 right-hand side $\mathbf{b}$ with $b_i = \frac{1}{m} \sum_{j=1}^{m} \phi_i(\mathbf{x}_j)$. To efficiently perform these operations, we use an implicit
136 matrix approach. That is, components of the matrix B get re-computed whenever they are accessed.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

This approach might seem wasteful at first glance. However, as the size of B scales quadratically in the number of grid points, it quickly becomes infeasible to store the matrix in memory.

Algorithm 1: The streaming algorithm for computing the right-hand side **b**

```
for i = 0...N do
    b_i ← 0
    for j = 0...m do
        b_i += ∏_{d=1}^{dim} φ_i^(d)(x_j^(d))
    b_i ← (1/m) b_i
```

Algorithm 2: The streaming algorithm for computing the matrix-vector multiplication $\mathbf{v}' = (B + \lambda I)\mathbf{v}$

```
for i = 0...N do
    v'_i ← 0
    for j = 0...N do
        v'_i += ∏_{d=1}^{dim} ∫_0^1 φ_i^(d) φ_j^(d) dx · v_j
    v'_i += λ · v_i
```

The computation of the right-hand side requires the computation of m vector components. Algorithm 1 shows the loop structure of a scalar version of this operation. As the basis function evaluations in the innermost loop are independent, we can parallelize this algorithm over the outer loop, *i.e.* the iteration over the grid points. The evaluation of hat functions is branch-free. Therefore, this algorithm is well-suited for vectorization.

We can formulate the matrix-vector operation as a second streaming algorithm with two nested loops over the grid points. This is shown in Algorithm 2. Similar to the hat function evaluations of the right-hand side, the L_2 norms can be independently computed as well. Thus, we can parallelize Algorithm 2 by processing the outer loop in parallel. The two cases for computing the 1d integral according to Eq. 15 slightly complicate vectorization. Our approach is the computation of both cases in a vectorized algorithm and a single conditional move to return the correct result. Computing the $x_{li} = x_{l'i'}$ case is only a single multiplication, as we can move the computation of $h_l = 2^{-l}$ to a precomputation step. Therefore, and because the $x_{li} = x_{l'i'}$ case rarely occurs, the overhead in each integration step is low compared to an optimal algorithm that would only evaluate the correct integration case.

4. Other Steps

In this section, we first present the algorithm for computing the k-nearest-neighbor graph. Then, we show how we apply the sparse grid density estimation to prune it. Finally, we briefly describe how we perform the connected component search in the pruned graph.

4.1. Computing the k-Nearest-Neighbor Graph

Algorithm 3: A variant of the $\mathcal{O}(m^2)$ k-nearest-neighbor algorithm that uses b bins.

```
Input  : dataset T = {x_i ∈ [0,1]^d}_{i=1}^m
Output: k-nearest-neighbor graph g as neighborhood list
for c = 1...b do
    dists_c ← 0
for i = 1...m do
    c ← 0 // c iterates over the number of bins
    for j = 1...m do
        dist ← distance(x_i, x_j)
        if dist < dists_{c+1} then
            bins_{c+1} ← j
            dists_{c+1} ← dist
        c ← (c + 1) mod b
    g_i ← extract_nearest_k(dists, bins, i)
```

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

To create the k-nearest-neighbor graph, we have developed an approximate variant of the $\mathcal{O}((k+d)m^2)$ algorithm that compares all pairs of data points. Instead of creating a neighborhood list with k entries directly, we employ an approach with b bins that implicitly splits the dataset into b ranges. For every data point i the dataset is iterated. Thereafter, each bin contains the nearest neighbor of its assigned range of data points. To obtain an approximate k-nearest-neighbor solution, the k indices with the smallest associated distances are selected from the b bins. Pseudocode for this approach is displayed in Algorithm 3.

This k-nearest-neighbor algorithm offers several advantages. It is not affected by the curse of dimensionality and therefore works well for the higher-dimensional datasets we target. In contrast, spatial partitioning approaches such as k-d-trees tend to suffer from the curse of dimensionality. Furthermore, it maps well to modern hardware architectures as it is straightforward to parallelize and vectorize. Through cache blocking of the outer loop that iterates i, the resulting algorithm is highly cache-efficient as well. Finally, the number of bins b is the only parameter to specify.

Binning was introduced for performance reasons. It allows us to only perform a single comparison in the innermost loop instead of k comparisons and, therefore, reduces the complexity to $\mathcal{O}(dm^2)$. The effect on the detected clusters is minimal, as it is very likely that nodes are still connected to close-by nodes of the same density region and therefore the same cluster. Furthermore, edges that intersect low-density regions get pruned, as we describe in the next section.

The overall clustering algorithm is relatively robust with regard to different values of k. However, k should not be too small. Otherwise, the k-nearest-neighbor graph might be split into more connected components than are desired. For larger values of k, performance decreases slightly in the subsequent pruning step as the pruning algorithm has linear complexity in k. In our experience, choosing k with values between five and ten balances this trade-off. Consequently, we set b to 16, as it is larger than expected values of k and leads to a good-enough approximation of the k-nearest-neighbor graph. On modern hardware platforms, this choice of b should not increase the cache or register memory requirements of the algorithm to an extent that would affect performance.

4.2. Pruning the k-Nearest-Neighbor Graph

Algorithm 4: A streaming algorithm for pruning low-density nodes and edges of the k-nearest-neighbor graph. The density function is evaluated at the location of the nodes and at the midpoints of the edges.

Input : k-nearest-neighbor graph g as neighborhood list, dataset T, density estimation $\hat{f}(\mathbf{x}) = \sum_j^N \alpha_j \phi_j(\mathbf{x})$, threshold t

Output: pruned k-nearest-neighbor graph g

```
for i = 0...m do
    if f̂(x_i) < t then
        prune_node(g_i)
        continue
    p_1,...,p_k ← load_midpoints(T, g_i)
    for j = 1...k do
        if f̂(p_j) < t then
            prune_edge(g_{i,j})
```

To prune the k-nearest-neighbor graph, we use two criteria. The density function is evaluated at the position $\mathbf{x}_i$ that corresponds to the current graph node g_i. If the density is below a threshold t, the node and its edges are removed. Furthermore, we evaluate at the midpoints of all outgoing edges and prune all edges where the density is below t. By evaluating the midpoints, clusters can be successfully separated even if two data points are in high-density regions that belong to different clusters. Our pruning approach is shown in Algorithm 4.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

Similar to the other algorithms presented, iterations of the outer loop are independent and can therefore be parallelized. The most expensive operations in this loop, multiple evaluations of the density function, are branch-free. Therefore, this algorithm is straightforward to vectorize as well. As there are only $\mathcal{O}(m \cdot (k+1))$ conditionals compared to overall $\mathcal{O}(m(k+1)N)$ operations, the conditionals do not significantly impact performance.

4.3. Connected Component Detection

To detect the weakly connected components in the pruned graph, we first convert the directed graph to an undirected graph by adding all inverted edges. Then, we perform a depth-first search to detect the connected components. This classical algorithm performs $\mathcal{O}(km)$ memory operations. Because the complexity of this algorithm is significantly lower compared to all other steps, this algorithm is only shared-memory parallelized and not distributed.

5. Implementation

In this section, we describe how our sparse grid clustering approach was implemented. To that end, we first consider the OpenCL-based node-level implementation and then present our distributed manager-worker approach.

5.1. Node-Level Implementation

Except for the connected component detection, all steps of the clustering algorithm have been implemented as OpenCL kernels. There are two OpenCL kernels for the density estimation: one for calculating the right-hand side and one for the matrix-vector multiplication. A third OpenCL kernel implements the k-nearest-neighbor graph creation and a fourth kernel implements the density-based graph pruning.

From a performance engineering perspective, these OpenCL kernels have some commonalities. All kernels were parallelized over the outermost loop, exploiting the fact that the loop iterations are independent. Furthermore, all OpenCL kernels were designed to be branch-free. The only exception is the density matrix-vector multiplication kernel that has a single branch in the innermost loop. This branch is implemented using the OpenCL *select* function to differentiate between the integration cases. On modern OpenCL platforms, this should be compiled to a conditional move. Because only standard arithmetic is used and because of the regular control flow, the four compute kernels get vectorized on all OpenCL platforms we tested. Due to the design of the compute kernels, we expect this to be the case on many untested OpenCL platforms as well.

The local memory is used in all kernels to either share grid points or data points between all threads of the work-group. For example, the prune graph kernel evaluates 1d sparse grid basis functions in its innermost loop. The threads of a work-group process different data points, but all always evaluate their data point with the same basis function. Therefore, the data of the currently processed basis function can be shared efficiently through the local memory. Furthermore, the data point assigned to a thread remains constant throughout the lifetime of the thread. It can therefore be stored in *private* memory, which translates to the register file on GPU devices and the L1 cache (or the registers) on processors.

Table 1 shows the number of floating-point operations for the different OpenCL kernels. As both the number of grid points N and the size of the dataset m can be large, all operations are potentially expensive. In most data mining scenarios, the sparse grid will have significantly fewer grid points than there are data points. Therefore, the k-nearest-neighbor graph creation is expected to be the most expensive operation. Depending on the number of CG iterations, the density matrix-vector product can be moderately expensive as well. However, it only depends on the grid points and therefore benefits from $N < m$.

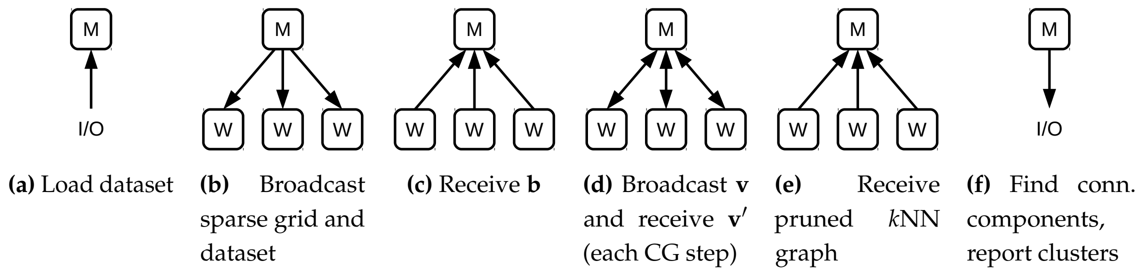
To estimate the achievable performance of our compute kernels, we calculated the arithmetic intensities of our compute kernels. As Tab. 1 shows, the arithmetic intensities of a work-group with a

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at *Algorithms* **2019**, *12*, 60; doi:10.3390/a12030060

Kernel	FP ops./complexity	Arith. int. (ws=1)	Arith. int. (ws=128)	Peak lim. (%)
density right-hand side	$N \cdot m \cdot d \cdot 6$	$1.5\,\mathrm{F\,B}^{-1}$	$192\,\mathrm{F\,B}^{-1}$	67%
density matrix-vector	CG-iter. $\cdot N^2 \cdot d \cdot 14$	$1.2\,\mathrm{F\,B}^{-1}$	$149\,\mathrm{F\,B}^{-1}$	64%
create graph	$m^2 \cdot d \cdot 3$	$1.0\,\mathrm{F\,B}^{-1}$	$129\,\mathrm{F\,B}^{-1}$	83%
prune graph	$m \cdot N \cdot (k+1) \cdot d \cdot 6$	$4.5\,\mathrm{F\,B}^{-1}$	$576\,\mathrm{F\,B}^{-1}$	67%

Table 1. The number of floating-point operations for the different OpenCL kernels and the arithmetic intensities (in floating point operations per byte) for a work-group size (ws) of one thread and 128 threads. The peak limit states the achievable fraction of the peak performance given the instruction mix of the compute kernels.



(a) Load dataset **(b)** Broadcast sparse grid and dataset **(c)** Receive **b** **(d)** Broadcast **v** and receive **v**′ (each CG step) **(e)** Receive pruned *k*NN graph **(f)** Find conn. components, report clusters

Figure 5. The distributed clustering algorithm from the perspective of the manager node. The (inexpensive) assignment of index ranges is not shown.

single thread would be too low to achieve a significant fraction of the peak performance on modern hardware platforms (see Tab. 2 for the machine balances of the hardware platforms we used). However, with a larger work-group size of 128 threads, and because we efficiently use the shared memory, the arithmetic intensity is strongly improved. As a consequence, memory accesses do not limit the performance of these compute kernels on modern hardware platforms. On processor-based platforms, the L1 cache enables similarly-high arithmetic intensity values.

The arithmetic intensity values would allow our compute kernels to achieve peak performance. However, as our compute kernels make use of instructions other than fused-multiply-add (FMA) operations, the instruction mix reduces the achievable performance. To calculate the peak limit given in Tab. 1, we make the (realistic) assumption that the remaining vector floating-point instructions run at half the performance of the FMA instructions [29].

5.2. Distributed Implementation

For distributed computing, we developed a manager-worker model that was implemented with MPI. To create work that can be assigned to the workers, we split the loops that were used for parallelization (the outermost loops of the compute kernels) once again. We use a static load balancing scheme that distributes the work equally to the workers. Our implementation supports single as well as double precision. Currently, we transfer double precision data even if single precision is used in the compute kernels.

From the perspective of the manager node, the distributed algorithm consists of four major steps: creating the right-hand side of the density estimation, the density matrix-vector products, an integrated graph-creation-and-prune step and the connected component search. These steps are shown in Fig. 5. Note that the matrix-vector multiplication step (Fig. 5d) is repeated once per CG iteration.

When the application is started, the dataset is read by the manager node and sent to all workers, requiring $m \cdot d \cdot 8\,\mathrm{B}$ of communication per worker. Then, the manager node creates the grid and sends it to the workers as well. This requires $2 \cdot N \cdot d \cdot 8\,\mathrm{B}$ per worker to be communicated. After these relatively expensive transfers are completed, grid and dataset are held by the workers. Therefore, most remaining communication steps only require small amounts of data to be transferred. We demonstrate in Sec. 6 that the overhead of these communication steps is indeed very low compared to the computational requirements of the remainder of the algorithm.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

Device	Type	Cores/ Shaders	Frequency	Peak (SP)	Mem. Bandw.	Machine Balance
Tesla P100	GPU	3584	1.3 GHz (boost)	9.5 TF	720 GB s^{-1}	12.9 F B^{-1}
FirePro W8100	GPU	2560	0.8 GHz (max)	4.2 TF	320 GB s^{-1}	13.2 F B^{-1}
2xXeon E5-2680v3	CPU	24	2.5 GHz (base)	1.9 TF	137 GB s^{-1}	14.0 F B^{-1}

Table 2. The hardware platforms used in the distributed and node-level experiments. We list the frequency type that best matches our observations during the experiments.

To compute the density right-hand-side operation, every worker computes an index range of the components of $\mathbf{b}$. As $\mathbf{b}$ is aggregated on the manager node, $N \cdot 8$ B need to be transferred. During each CG step and after the final CG step, the manager sends $\mathbf{v}$ ($\boldsymbol{\alpha}$ after the final iteration) to all workers. Each worker calculates the result of an index range of $\mathbf{v}'$ and communicates the partial result back to the manager node. Therefore, $N \cdot 8$ B per worker are communicated during each iteration and after the final iteration. Collecting the partial results for $\mathbf{v}'$ requires another $N \cdot 8$ B to be transferred per CG step.

The creation of the k-nearest-neighbor graph only requires the assignment of index ranges and no further communication. Because the pruning of the k-nearest-neighbor graph reuses the same index ranges that were assigned in the graph creation step, this step only requires the pruned graph to be sent to the manager node. This step requires $k \cdot m \cdot 8$ B to be transferred. Having received the pruned graph, the manager node performs the connected component detection and has thereby computed the clusters.

6. Results

In this section, we evaluate our distributed and performance-portable sparse grid clustering approach. We first present the hardware platforms and datasets that were used in the experiments. Then we provide the results of our node-level experiments that demonstrate performance portability. The quality of the clustering is discussed in the context of the node-level experiments as well. Finally, we present distributed performance results for two supercomputers: Hazel Hen and Piz Daint.

6.1. Hardware Platforms

On the node level, we used three different hardware platforms. Two of them are GPUs: the Nvidia Tesla P100 and the AMD FirePro W8100. To represent standard processor platforms, we used a dual socket machine with two Intel Xeon E5-2680v3 processors. The relevant technical details of these hardware platforms are summarized in Tab. 2.

Our distributed experiments were conducted on two supercomputers. The Cray XC40 Hazel Hen is an Intel processor-based machine with 7712 nodes for a peak performance of 7.4 PF. Hazel Hen is located at the High Performance Computing Center Stuttgart (HLRS) in Stuttgart, Germany. It has dual socket nodes with Xeon E5-2680v3 processors and 128 GB of memory per node.

The Cray XC40/XC50 Piz Daint is a mostly GPU-based supercomputer with a peak performance of 27 PF. Piz Daint is located at the Swiss National Supercomputing Centre (CSCS) in Lugano, Switzerland. Each of the XC50 nodes that we used have a single Intel Xeon E5-2690v3 processor with 64 GB of memory and a single Nvidia Tesla P100 GPU. In our experiments, we only used the Tesla P100 to compute the main compute kernels of our application.

6.2. Datasets and Experimental Setup

In all of our experiments, we used synthetic datasets with clusters drawn from Gaussian distributions. The cluster centers μ were drawn randomly. We normalized the datasets to $[0.1, 0.9]^d$. As this moves data points sufficiently towards the center of the domain, we can use a sparse grid without boundary grid points.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

Name	Size	Clust.	σ	Dim.	Dist.	Noise	Type
10M-3C	10M	3	0.12	10	$3 \cdot \sigma$	0%	distributed
100M-3C	100M	3	0.12	10	$3 \cdot \sigma$	0%	distributed
1M-10C	1M	10	0.05	10	$7 \cdot \sigma$	2%	node-level
1M-100C	1M	100	0.05	10	$7 \cdot \sigma$	2%	node-level
10M-100C	10M	100	0.05	10	$7 \cdot \sigma$	2%	node-level

Table 3. The Gaussian datasets for the distributed and the node-level runs

Name	λ	Threshold t	Level	Grid Points	CG ϵ	k	ARI	Type
1M-10C	1E-5	667	6	76k	1E-2	6	1.0	node-level
1M-100C	1E-6	556	7	0.4M	1E-2	6	0.85	node-level
10M-10C	1E-5	1167	7	0.4M	1E-2	6	1.0	node-level
10M-100C	1E-6	1000	7	0.4M	1E-2	6	0.90	node-level
10M-3C	1E-6	$0.7 \cdot \max(\alpha)$	7	0.4M	1E-3	5	-	distributed
100M-3C	1E-6	$0.7 \cdot \max(\alpha)$	8	1.9M	1E-3	5	-	distributed

Table 4. The parameters used for configuring the clustering algorithm and the adjusted Rand index (ARI) for the node-level experiments. In the distributed runs, the threshold was specified as a fraction of the maximum surplus of the density function. The node-level runs used an absolute threshold value.

307 The parameters used to generate the datasets are listed in Tab. 3. The datasets with 100 clusters
308 are challenging, as the density estimation needs to correctly separate 100 high-density regions in a
309 moderately-high dimensional setting. Furthermore, to make it possible to assess the quality of the
310 clustering, we generated the node-level dataset so that the clusters are well-separated by forcing a
311 minimum distance of $7 \cdot \sigma$ between the cluster centers. We verified that the noise connects all clusters
312 in the unpruned k-nearest-neighbor graph.
313 As the clustering algorithm requires parameterization as well, these parameters are shown in
314 Tab. 4. The adjusted Rand index (ARI) is a quality measure for clustering and is addressed in Sec. 6.4.
315 In all of our experiments, we used single-precision floating-point arithmetic.

316 *6.3. Node-Level Performance and Performance-Portability*

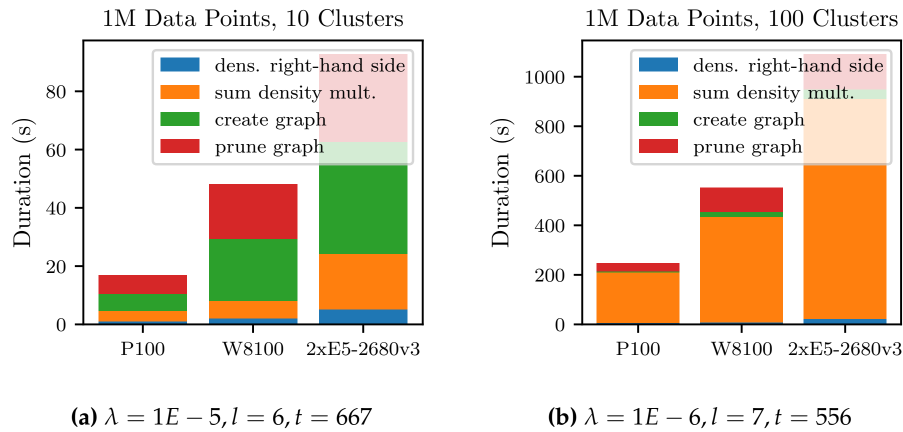
317 The tables 6 and 7 show the runtimes of the node-level experiments. For more consistent results,
318 the runs were repeated four times and the measurements averaged. The 1M-10C dataset could be
319 processed on a Tesla P100 in less than 20 s. Processing the 1M-100C dataset is more time-consuming
320 and required 248 s again using a Tesla P100. The main reason for the time increase is because the
321 density estimation requires more time due to a larger sparse grid and a smaller λ, which leads to more
322 CG iterations.

		Tesla P100	FirePro W8100	2xE5-2680v3
dens. right-hand side	GFLOPS	4584	2271 (753 MHz)	1177
limit: 67% peak	peak (of lim.)	48% (72%)	59% (88%)	61% (91%)
dens. matrix-vector	GFLOPS	4090	1939 (759 MHz)	919
limit: 64% peak	peak (of lim.)	43% (67%)	50% (78%)	48% (75%)
create graph	GFLOPS	5474	1433 (467 MHz)	852
limit: 83% peak	peak (of lim.)	58% (70%)	60% (72%)	44% (53%)
prune graph	GFLOPS	5360	1817 (822 MHz)	1265
limit: 67% peak	peak (of lim.)	56% (84%)	43% (64%)	66% (99%)

Table 5. The node-level performance of the clustering algorithm. All results are for single-precision arithmetic. The performance was measured with the 10M-10C dataset and the parameters listed in Tab. 4. Note that the achievable peak performance is limited by the instruction mix to values significantly below 100%.

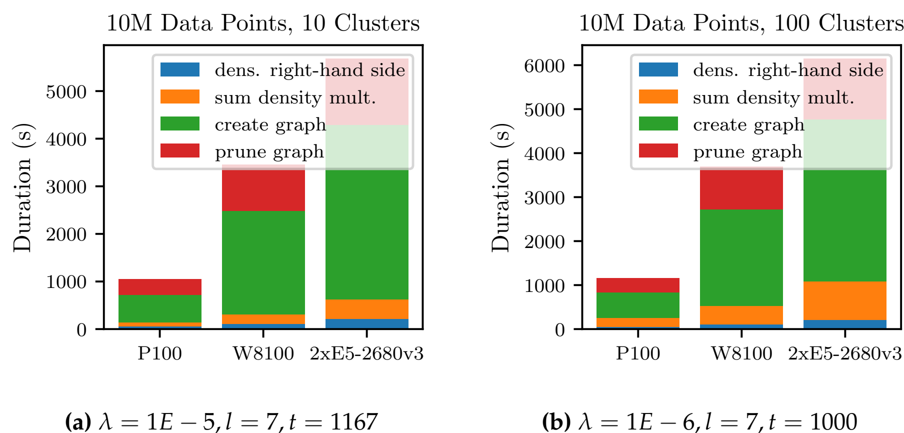
Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060



(a) $\lambda = 1E-5, l = 6, t = 667$ **(b)** $\lambda = 1E-6, l = 7, t = 556$

Figure 6. The duration of the node-level experiments with one million data points. Because the 1M-100C dataset requires a larger grid, the density estimation takes up most of the overall runtime.



(a) $\lambda = 1E-5, l = 7, t = 1167$ **(b)** $\lambda = 1E-6, l = 7, t = 1000$

Figure 7. The duration of the node-level experiments with 10 million data points

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

The experiments with the 10 million data points datasets are shown in Tab. 7. Due to the increased size of the datasets, the k-nearest-neighbor graph creation takes up the largest fraction of the runtime in both experiments. This illustrates that for large datasets, because of its quadratic complexity, the k-nearest-neighbor graph creation step will dominate the overall runtime. In these two experiments, increasing the number of clusters has only a small effect on the runtime. Mainly, because in both cases a sparse grid with level $l = 7$ was used. On the P100 platform, the experiments with the 10M-100C dataset took 1162 s. The other hardware platforms took longer, proportional to their lower raw performance.

Table 5 shows the performance achieved in the node-level experiments. It displays the performance in GFLOPS and the achieved fraction of peak performance. The achieved fraction of the peak performance relative to the instruction-mix-based limit is displayed as well. These results were calculated from the runs with the 10M-10C dataset as specified in Tab. 4.

Our implementation achieved a significant fraction of the peak performance across all devices. Additionally, if the limit imposed by the instruction mix is taken into account, we see that many combinations of kernels and devices run close to their maximally achievable performance. The only kernel that reaches less than two-thirds of its achievable performance is the create graph kernel on the Xeon E5 platform. We suspect that this is due to throttling of the processor, as this operation puts extreme stress on the vector units.

The fastest device by a significant margin is the Tesla P100, as it is the most recent of the devices and has the highest theoretical peak performance. It is $2.23 - 3.29$x faster than the W8100 and $4.41 - 5.49$x faster than the Xeon E5 pair.

The FirePro W8100 achieves similar fractions of the peak performance compared to the P100 at a lower absolute level of performance. It is still $1.67 - 1.98$x faster than the pair of Xeon E5 processors. During our experiments, the FirePro W8100 displayed strong throttling which is why we list the average frequencies observed for the invidivual compute kernels. The reduced frequencies imply lower achievable peak performance ($2 \cdot 2560 \cdot f_{\text{avr}}$) which we take into account for the calculation of the peak performance and the resulting achieved fraction of peak performance. The average frequencies reported were measured in a separate run of the 10M-10C experiment. In case of the k-nearest-neighbor graph kernel, a frequency of only 492 MHz was measured. This nearly halves the achievable performance of this compute kernel.

Because it has the lowest absolute performance, the pair of Xeon E5 processors scores lowest. However, the achieved fractions of the peak performance are similar to the other devices. This indicates that performance is not only portable across GPU platforms, but processor-based platforms as well.

6.4. Clustering Quality and Parameter Tuning

This work mostly focuses on the performance of our sparse grid clustering approach. Nevertheless, to make our evaluation more realistic, we tuned the clustering parameters of our node-level runs for (nearly) optimal clustering quality. For a more detailed discussion of the achievable level of quality, we refer to prior work which compared sparse grid clustering to other clustering algorithms [21]. A comparison of the sparse grid density estimation to other density estimation methods is available as well [30].

To assess the quality, we used the adjusted Rand index (ARI) which compares two cluster mappings. Because we know the mapping of data points to clusters of each of our synthetic datasets, these reference cluster mappings were compared to the output of the sparse grid clustering algorithm. The calculated ARI of the node-level experiments is shown in Tab. 4. These results show that we can nearly perfectly reconstruct the clusters of both datasets with ten clusters. The datasets with 100 clusters are more challenging and would require slightly larger grids for further improvements.

We used a parameter tuning approach to fit the computed cluster mappings to the reference mappings. During parameter tuning, we first select a value for the regularization parameter λ and then search for the best pruning threshold t. This was implemented as two nested binary searches.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

To speed up parameter tuning in general, our sparse grid clustering implementation allows for reusing of the k-nearest-neighbor graph and calculated density estimations. As the k-nearest-neighbor graph is the same independent of all parameters, it can be calculated once overall. Moreover, the density estimation changes only if λ is changed. Thus, the density estimation can be reused while an optimal value for t is searched. Only the comparably cheap graph pruning operation and the connected component search are performed at every parameter tuning step.

6.5. Distributed Results on Hazel Hen

Figure 8 shows the results of the distributed experiments conducted on Hazel Hen. Results are given for the individual compute kernels as well as the whole application run. The total runtime, and the average application TFLOPS derived from it, is based on the wall-clock time of the application and not only on the three major distributed operations. At the highest node count, it took 4226 s to process the 100M-3C dataset and 259 s to process the 10M-3C dataset. We achieved up to 100 TFLOPS for the 100M-3C dataset using 128 nodes and up to 23 TFLOPS for the 10M-3C using 32 nodes. Therefore, we achieved 41% and 37% of the peak performance at the highest number of nodes for the whole application including all communication and file input-output operations.

The creation and pruning of the k-nearest-neighbor graph scales nearly linearly. Calculating the density estimation scales slightly worse. As the grid is much smaller than the dataset, there is too little work available per node during the density estimation step to achieve optimal performance at high node counts.

Figure 8c displays the duration of the initial loading and distribution of the dataset, the creation and the transfer of the sparse grid and the duration of the connected component search. As Fig. 8c shows, loading and communicating the dataset does not take up significant amounts of time. The same is true for creating and transferring the sparse grid. However, the connected component search becomes relatively expensive for the 100M-3C dataset, as it is performed on a single node and therefore cannot scale with an increasing number of nodes. Nevertheless, at 128 nodes the connected component search still only requires 107 s or 2.5% of the total runtime for the 100M-3C dataset. For the 10M-3C dataset and 32 nodes, the connected component search takes up 2.2% of the total runtime.

6.6. Distributed Results on Piz Daint

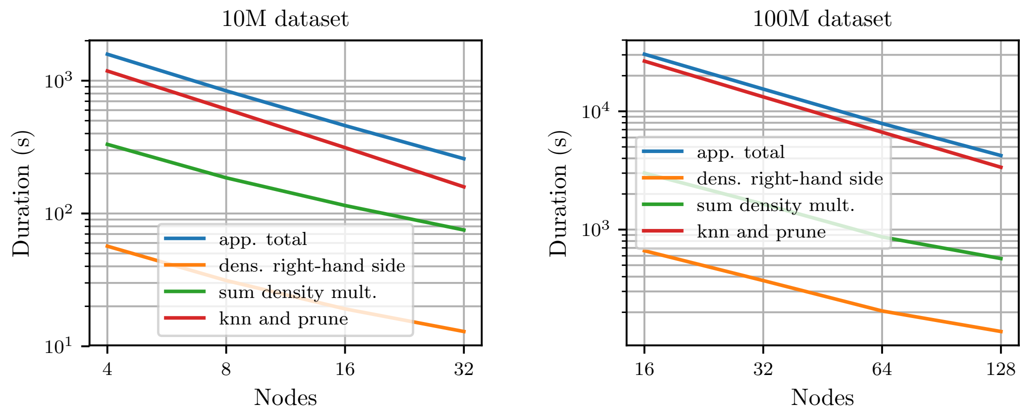
We conducted the distributed experiments before we were able to do some final node-level optimizations, and due to compute time limitations we were not able to recompute the experiments. Thus, the results of these experiments are not directly comparable to the node-level performance results. Since these experiments, the node-level performance of all compute kernels was improved. Because of this, scalability might by slightly overestimated. Furthermore, the duration of the connected component search is not listed in these results, as we used a different algorithm at the time of the experiments.

Figure 9 shows duration and performance of the experiments performed on Piz Daint for both the 10M-3C and 100M-3C datasets. Again, results are given for the individual compute kernels as well as the whole application run. As Fig. 9a shows, the application scales well to 128 nodes. Similar to the Hazel Hen results, the integrated graph-creation-and-prune step scales nearly linearly, whereas the density estimation scales slightly worse. Using 32 nodes, the clustering of the 10M-3C dataset takes 100 s. It takes 1198 s to cluster the 100M-3C dataset using 128 nodes. This translates to an average performance of 59 TFLOPS for the 10M-3C dataset and 352 TFLOPS for the 100M-3C dataset as Fig. 9b shows. Thus, at 128 nodes our implementation still achieves 29% of the peak performance for the whole application including all communication and the loading of the dataset.

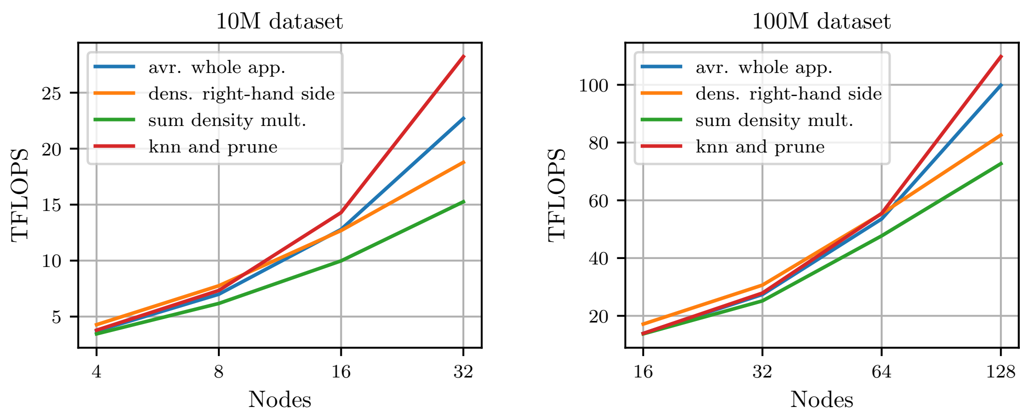
Figure 9c displays the duration of the initial loading and distribution of the dataset and the creation and transfer of the sparse grid to the workers. Only the loading of the dataset is somewhat expensive.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

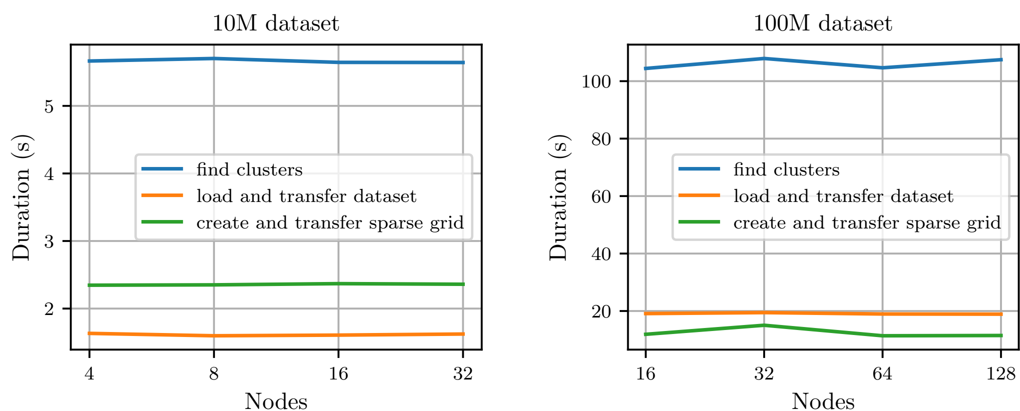
Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060



(a) The durations in seconds of the clustering experiments on Hazel Hen



(b) The performance in TFLOPS of the clustering experiments on Hazel Hen

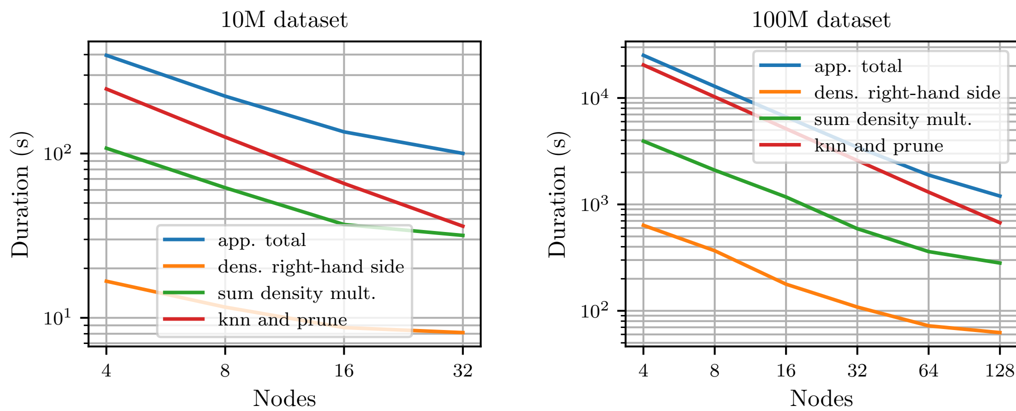


(c) The durations for loading the dataset, creating the sparse grid and transferring both to the workers. We additionally show the time needed to perform the connected component search.

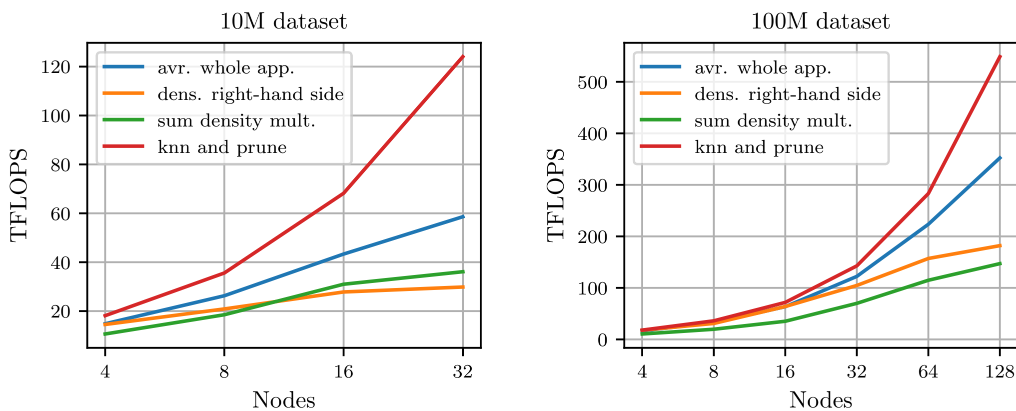
Figure 8. Strong scaling results for both the 10M (left graphs) and 100M (right graphs) Gaussian dataset on Hazel Hen. Figure 8a and Fig. 8b show duration and performance of the major compute kernels and the application as a whole. The duration of other (minor) tasks is shown in Fig. 8c.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

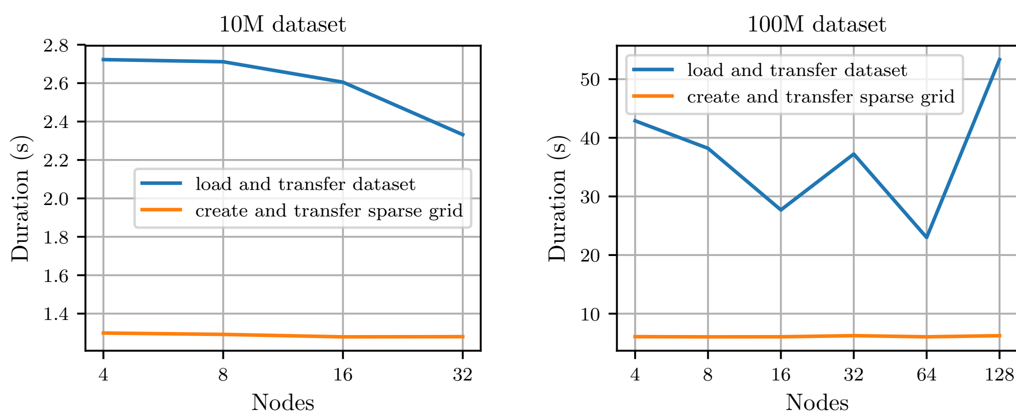
Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060



(a) The durations in seconds of the clustering experiments on Piz Daint



(b) The performance in TFLOPS of the clustering experiments on Piz Daint



(c) The durations for loading the dataset, creating the sparse grid and transferring both to the workers.

Figure 9. Strong scaling results for both the 10M (left graphs) and 100M (right graphs) Gaussian dataset on Piz Daint. Figure 9a and Fig. 9b show duration and performance of the major compute kernels and the application as a whole. The duration of other (minor) tasks is shown in Fig. 9c.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

Compared to Hazel Hen, the performance on Piz Daint is consistently higher, which is explained by the difference in node-level performance. However, due to the processor-based architecture, Hazel Hen nodes require less work per node to be fully utilized. Therefore, on Hazel Hen a slightly higher fraction of peak performance was achieved.

7. Discussion and Future Work

Sparse grid clustering, as implemented in the open source library SG^{++}, is one the few clustering methods available for the clustering of large datasets on HPC machines. The underlying density estimation approach enables the detection of clusters with non-convex shapes and without a predetermined number of clusters. Due to the sparse grid discretization of the underlying feature space, the grid or discretization points are chosen independently of the data points. This is key to the linear complexity with respect size of the data of all sparse-grid-related algorithms. This property is highly useful for addressing big data challenges.

With our optimized implementation, we have demonstrated performance portability across three hardware platforms. Due to the use of OpenCL as the programming language, careful and highly-tuned performance optimization, and algorithms that map very well to the capabilities of modern hardware platforms, we expect similar performance on related platforms. Our strong scaling experiments show that even on 128 nodes of Piz Daint, scalability is mainly limited by the available work per node.

Our method achieves a significant fraction of the peak performance on all devices tested. This shows that OpenCL is a good choice for developing performance-portable software. Furthermore, our GPU results illustrate how the higher raw performance of GPUs in contrast to CPUs translates to similarly improved time-to-solution.

As our next steps, we plan to further improve the performance of our approach by addressing two key issues: First, the k-nearest-neighbor graph creation currently uses an $\mathcal{O}(m^2)$ algorithm and thus represents the bottle-neck. We already have an early implementation of a GPU-enabled variant of the locality-sensitive hashing algorithm. The locality-sensitive hashing algorithm can calculate an approximate k-nearest-neighbor graph in sub-quadratic complexity [31]. Adopting this algorithm, sparse grid clustering can be performed in sub-quadratic complexity as well.

Second, our implementation supports the use of spatially-adaptive sparse grids [22,30]. They enable the placement of grid points only where they significantly contribute to the overall solution. An adaptive approach will significantly increase the dimensionality of the datasets that can be clustered as it has been demonstrated for standard learning tasks before. Currently, creating an adaptively-refined sparse grid is itself expensive as it requires the system of linear equations of the density estimation to be solved repeatedly after each refinement. Thus, a priori refinement strategies that create well-adapted sparse grids with less effort are another important direction of future research.

8. Materials and Methods

The source code of this study will be made available as part of the sparse grid toolbox SG^{++} at the time of publication [32]. We archive the scripts for creating the synthetic datasets at the same location.

Author Contributions: Conceptualization, D.P, G.D. and D.Pf.; Methodology, D.P and G.D.; Software, D.P and G.D.; Validation, D.P and G.D.; Formal Analysis, D.P and G.D.; Investigation, D.P; Resources, D.P; Data Curation, D.P and G.D.; Writing–Original Draft Preparation, D.P; Writing–Review and Editing, D.P., G.D. and D.Pf.; Visualization, D.P.; Supervision, D.Pf.; Project Administration, D.Pf.; Funding Acquisition, D.Pf..

Funding: This research was partially funded by the German Research Foundation (DFG) within the Cluster of Excellence in Simulation Technology (EXC 310/2).

Acknowledgments: We thank John Biddiscombe from the Swiss National Supercomputing Centre (CSCS) for his help in getting sparse grid clustering running on Piz Daint. Furthermore, we thank Martin Bernreuther from the High Performance Computing Center Stuttgart (HLRS) for his support on Hazel Hen.

Conflicts of Interest: The authors declare no conflict of interest.

Peer-reviewed version available at *Algorithms* **2019**, *12*, 60; doi:10.3390/a12030060

1. Hastie, T.; Tibshirani, R.; Friedman, J. *The Elements of Statistical Learning*, 2 ed.; Springer Series in Statistics, Springer-Verlag New York, 2009.
2. Kanungo, T.; Mount, D.M.; Netanyahu, N.S.; Piatko, C.D.; Silverman, R.; Wu, A.Y. An Efficient *k*-Means Clustering Algorithm: Analysis and Implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **2002**, *24*, 881–892.
3. Arthur, D.; Vassilvitskii, S. K-means++: The Advantages of Careful Seeding. Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2007; SODA'07, pp. 1027–1035.
4. Ester, M.; Kriegel, H.P.; Sander, J.; Xu, X. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. AAAI Press, 1996, KDD'96, pp. 226–231.
5. Song, H.; Lee, J.G. RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning. Proceedings of the 2018 International Conference on Management of Data; ACM: New York, NY, USA, 2018; SIGMOD'18, pp. 1173–1187.
6. Gan, J.; Tao, Y. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data; ACM: New York, NY, USA, 2015; SIGMOD '15, pp. 519–530.
7. Hinneburg, A.; Gabriel, H.H. DENCLUE 2.0: Fast Clustering Based on Kernel Density Estimation. Proceedings of the 7th International Conference on Intelligent Data Analysis; Springer-Verlag: Berlin, Heidelberg, 2007; IDA'07, pp. 70–80.
8. von Luxburg, U. A tutorial on spectral clustering. *Statistics and Computing* **2007**, *17*, 395–416.
9. Zupan, J.; Novič, M.; Li, X.; Gasteiger, J. Classification of multicomponent analytical data of olive oils using different neural networks. *Analytica Chimica Acta* **1994**, *292*, 219–234.
10. Estivill-Castro, V. Why So Many Clustering Algorithms: A Position Paper. *SIGKDD Explor. Newsl.* **2002**, *4*, 65–75.
11. Takizawa, H.; Kobayashi, H. Hierarchical parallel processing of large scale data clustering on a PC cluster with GPU co-processing. *The Journal of Supercomputing* **2006**, *36*, 219–234.
12. Fang, W.; Lau, K.K.; Lu, M.; Xiao, X.; Lam, C.K.; Yang, P.Y.; He, B.; Luo, Q.; Sander, P.V.; Yang, K. Parallel Data Mining on Graphics Processors. *Hong Kong Univ. Sci. and Technology, Hong Kong, China, Tech. Rep. HKUST-CS08-07* **2008**.
13. Jian, L.; Wang, C.; Liu, Y.; Liang, S.; Yi, W.; Shi, Y. Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA). *The Journal of Supercomputing* **2013**, *64*, 942–967.
14. Bhimani, J.; Leeser, M.; Mi, N. Accelerating K-Means Clustering with Parallel Implementations and GPU Computing. High Performance Extreme Computing Conference (HPEC), 2015 IEEE. IEEE, 2015, pp. 1–6.
15. Farivar, R.; Rebolledo, D.; Chan, E.; Campbell, R.H. A Parallel Implementation of K-Means Clustering on GPUs. Proceedings of the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2008, 2008, pp. 340–345.
16. Böhm, C.; Noll, R.; Plant, C.; Wackersreuther, B. Density-based Clustering Using Graphics Processors. Proceedings of the 18th ACM Conference on Information and Knowledge Management; ACM: New York, NY, USA, 2009; CIKM '09, pp. 661–670.
17. Andrade, G.; Ramos, G.; Madeira, D.; Sachetto, R.; Ferreira, R.; Rocha, L. G-DBSCAN: A GPU Accelerated Algorithm for Density-based Clustering. *Procedia Computer Science* **2013**, *18*, 369–378.
18. Bahmani, B.; Moseley, B.; Vattani, A.; Kumar, R.; Vassilvitskii, S. Scalable K-Means++. *Proc. VLDB Endow.* **2012**, *5*, 622–633.
19. He, Y.; Tan, H.; Luo, W.; Feng, S.; Fan, J. MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. *Frontiers of Computer Science* **2014**, *8*, 83–99.
20. Bellman, R. *Adaptive Control Processes: A Guided Tour*; 'Rand Corporation. Research studies, Princeton University Press, 1961.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 2 February 2019 doi:10.20944/preprints201902.0019.v1

Peer-reviewed version available at Algorithms 2019, 12, 60; doi:10.3390/a12030060

517 21. Peherstorfer, B.; Pflüger, D.; Bungartz, H.J. Clustering Based on Density Estimation with Sparse Grids. In *KI 2012: Advances in Artificial Intelligence*; Glimm, B.; Krüger, A., Eds.; Springer Berlin Heidelberg, 2012; Vol. 7526, *Lecture Notes in Computer Science*, pp. 131–142.

520 22. Pflüger, D. Spatially Adaptive Sparse Grids for High-Dimensional Problems. PhD thesis, Verlag Dr.Hut, Technische Universität München, 2010.

522 23. Garcke, J. Maschinelles Lernen durch Funktionsrekonstruktion mit verallgemeinerten dünnen Gittern. PhD thesis, Universität Bonn, Institut für Numerische Simulation, 2004.

524 24. Heinecke, A.; Pflüger, D. Emerging Architectures Enable to Boost Massively Parallel Data Mining Using Adaptive Sparse Grids. *International Journal of Parallel Programming* **2012**, *41*, 357–399.

526 25. Heinecke, A.; Karlstetter, R.; Pflüger, D.; Bungartz, H.J. Data Mining on Vast Datasets as a Cluster System Benchmark. *Concurrency and Computation: Practice and Experience*, *28*, 2145–2165.

528 26. Pfander, D.; Heinecke, A.; Pflüger, D. A new Subspace-Based Algorithm for Efficient Spatially Adaptive Sparse Grid Regression, Classification and Multi-evaluation. Sparse Grids and Applications - Stuttgart 2014; Garcke, J.; Pflüger, D., Eds. Springer International Publishing, 2016, pp. 221–246.

531 27. Bungartz, H.J.; Griebel, M. Sparse Grids. *Acta Numerica* **2004**, *13*, 1–123.

532 28. Hegland, M.; Hooker, G.; Roberts, S. Finite Element Thin Plate Splines In Density Estimation. *ANZIAM Journal* **2000**, *42*, 712–734.

534 29. Fog, A. Instruction tables. Technical report, Technical University of Denmark, 2018.

535 30. Peherstorfer, B.; Pflüger, D.; Bungartz, H.J. Density Estimation with Adaptive Sparse Grids for Large Data Sets. Proceedings of the 2014 SIAM International Conference on Data Mining, 2014, pp. 443–451.

537 31. Datar, M.; Immorlica, N.; Indyk, P.; Mirrokni, V.S. Locality-Sensitive Hashing Scheme Based on P-stable Distributions. Proceedings of the Twentieth Annual Symposium on Computational Geometry; ACM: New York, NY, USA, 2004; SCG'04, pp. 253–262.

540 32. SG++: General Sparse Grid Toolbox. https://github.com/SGpp/SGpp. Accessed: 2019-1-14.