Efficient Evidence Accumulation Clustering for large datasets/big data

Diogo Silva

***Abstract*—The unprecedented collection and storage of data in electronic format has given rise to an interested in automated analysis for generation of knowledge and new insights. Cluster analysis is a good candidate since it makes as few assumptions about the data as possible. A vast body of work on clustering methods exist, yet, typically, no single method is able to respond to the specificities of all kinds of data. Evidence Accumulation Clustering (EAC) is a robust state of the art ensemble algorithm that has shown good results. However, this robustness comes with higher computational cost. Currently, its application is slow or restricted to small datasets. The objective of the present work is to scale EAC, allowing its applicability to big datasets, with technology available at a typical workstation. Three approaches for different parts of EAC are presented: a parallel GPU K-Means implementation, a novel strategy to build a sparse CSR matrix specialized to EAC and Single-Link based on Minimum Spanning Trees using an external memory sorting algorithm. Combining these approaches, the application of EAC to much larger datasets than before possible was accomplished.**

***Index Terms*—Clustering methods, EAC, K-Means, MST, GPGPU, CUDA, Sparse matrices, Single-Link**

I. Introduction

ADVANCES in technology allow for the collection and storage of unprecedented amount and variety of data, of which there is an interest in performing automated analysis for generation of knowledge and new insights. A growing body of formal methods aiming to model, structure and/or classify data already exist, e.g. linear regression, principal component analysis, cluster analysis, support vector machines, neural networks. Cluster analysis is an interesting tool because it typically does not make assumptions on the structure of the data, which is specially interesting when no prior information about the data is known.

A vast body of work on clustering algorithms exists, but usually different methods are suited to datasets of different characteristics. Currently, there are state of the art algorithms that are more robust than "traditional" algorithms by having a wider applicability or being less dependent on input parameters. One such approach is Evidence Accumulation Clustering (EAC) [8], belonging to the wider class of ensemble methods. EAC is a state-of-the art clustering method that addresses the robustness challenge, but, currently, its computational complexity restricts its application to small datasets.

The present work is concerned with pushing the current limits of the EAC method to large datasets by addressing the challenges of scalability and efficiency without compromising robustness, using technology available in a desktop workstation. Two main approaches exist for scaling: using algorithms with better computational complexity in the EAC steps or turning to parallel and external memory computation for speeding up and addressing the space complexity. Quantum clustering algorithms [5], [10] were researched under the motivation of having better computational complexity, but it proved to be a fruitless endeavor mainly due to its prohibitive computational complexity within the EAC context. This moved the focus of research to parallel computation, more specifically General Purpose computation in Graphics Processing Units (GPGPU), and external memory (using hard drives) solutions.

Selected work from the this dissertation was compiled into a conference paper and submitted to the 5^{th} The International Conference on Pattern Recognition Applications and Methods.

This document is structured as follows. Relevant concepts to understand the work done are presented in section II. Since EAC is a three step method and each of these must be optimized individually, sections III, IV and V explain what was done in each step. Finally, the implemented optimizations are tested and the results presented in section VI.

II. Background

A. Clustering: main concepts and notation

The goal of data clustering is the discovery of the *natural grouping(s)* of a set of patterns, points or objects [12], by grouping patterns (usually represented as a vector of measurements or a point in space [11]) based on some similarity, such that patterns belonging to the same cluster are typically more similar to each other than to patterns of other clusters.

Within the clustering context, a *pattern* $\mathbf{x}$ is a single data item represented as a vector of d *features* x_i that characterize it. A *pattern set* (or dataset) $\mathcal{X}$ is then the collection of all n patterns $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. The desired clustering, typically, is one that reflects the natural structure of the data, i.e. the original ground truth labeling. *Hard* clustering (or partitional) techniques assign a class label l_i to each pattern $\mathbf{x}_i$. The whole set of labels corresponding to a pattern set $\mathcal{X}$ is given by $\mathcal{L} = \{l_1, \ldots, l_n\}$, where l_i is the label of pattern $\mathbf{x}_i$. A partition P is a collection of k *clusters*, which are an exclusive subset of nc patterns $\mathbf{x}_i$ taken from the pattern set. A clustering *ensemble* $\mathbb{P}$ is a set of N partitions P^j of a given pattern set, each of which is composed by a set of k_j clusters C_i^j, where $j = 1, \ldots, N$, $i = 1, \ldots, k_j$. Each cluster is composed by a set of nc_i^j patterns that does not intercept any other cluster of the same partition.

Typically, a clustering algorithm will use a *proximity* measure for determining how alike two patterns are, which can

Diogo Silva is affiliated with the Portuguese Air Force Academy, Portugal e-mail: dasilva@academiafa.edu.pt.

either be a *similarity* or a *dissimilarity* measure. A *distance* is a dissimilarity function d which yields non-negative real values and is also a *metric*, which means it obeys the following three properties: identity, symmetry and triangle inequality. Different proximity measures may be more appropriate in certain contexts.

Validity measures measure the quality of a partition. Several *validity measures* exist and they can placed in two main categories [1]. *External* measures use *a priori* information about the data to evaluate the clustering against some external structure. Examples of such measures include the *Consistency Index* [6] and the H-index [17]. *Internal* measures, on the other hand, determine the quality of the clustering without the use of external information about the data.

B. Evidence Accumulation Clustering

EAC is a clustering ensemble method. The underlying idea behind ensemble clustering is to take a collection of partitions, a *clustering ensemble*, and combine it into a single partition of better quality than any in the ensemble. The goal of EAC is to find an optimal partition P^* containing k^* clusters that is consistent with $\mathbb{P}$, robust to small variations in $\mathbb{P}$ and has a good fit with ground truth, when available. EAC makes no assumption on the number of clusters in each partition in $\mathbb{P}$. Its approach is divided in 3 steps:

1) **Production** of a clustering ensemble $\mathbb{P}$ (the evidence);
2) **Combination** of the ensemble into a co-association matrix;
3) **Recovery** of the natural clusters of the data.

In the first step, a clustering ensemble is produced. It is of interest to have variety in the ensemble with the intention to better capture the underlying structure of the data, which can be obtained by varying the number of clusters in $\mathbb{P}$ within an interval $[K_{min}, K_{max}]$. The ensemble is usually produced by random initialization of K-Means, specifying only the $[K_{min}, K_{max}]$ interval from which a random number of centroids will be picked. The choice of this interval and its influence on the EAC's properties will be a topic of discussion further ahead.

The ensemble of partitions is combined in the second step, where a non-linear transformation turns the ensemble into a co-association matrix [8], i.e. a matrix $\mathcal{C}$ where each of its elements n_{ij} is the association value between the pattern pair (i, j), which is computed as the number of co-occurrences in the same cluster. The rationale is that pairs that are frequently clustered together are more likely to be representative of a true link between the patterns [7], revealing the underlying structure of the data. The construction of the co-association matrix is at the very core of this method.

The co-association matrix is then used in the final step for obtaining the final partition. The co-association between any two patterns can be interpreted as a proximity measure. Hierarchical algorithms such as Single-Link or Average-Link have been used, since they operate over pair-wise dissimilarity matrices. These output a dendrogram, which codes a hierarchy of a pattern set, can be cut at different levels to produce a partition. Furthermore, the lifetime criteria can automatically decide the number of clusters by cutting a dendrogram in the interval corresponding to the longest lifetime. The k-cluster lifetime is defined as the range of threshold values on the dendrogram that lead to the identification of k clusters [8].

C. K-Means

K-Means is briefly reviewed here, since it is used by EAC. K-Means is a hard clustering algorithm that takes two initialization parameters: the number of clusters and the centroid initializations. It starts by assigning each pattern to its closer cluster based on the cluster's centroid. This is called the **labeling** step since one usually uses cluster labels for this assignment. The centroids are then recomputed based on this assignment, in the **update** step. The new centroids are the mean of all the patterns belonging to the clusters. These two steps are executed iteratively until a stopping condition is met, usually the number of iterations, a convergence criteria or both. The initial centroids are usually chosen randomly, but other schemes exist to improve the overall accuracy of the algorithm.

D. Single-Link

Single-Link (SL) is a Hierarchical Agglomerative Clusteirng (HAC) algorithm that has been used for the last step of EAC. HAC algorithms operate over a pair-wise dissimilarity matrix and output a dendrogram. The main steps of an agglomerative hierarchical clustering algorithm are [11] (1) the creation of a pair-wise dissimilarity matrix of all patterns, where each pattern is a distinct cluster singleton; (2) finding the closest clusters, merge them and update the matrix to reflect this change; and, (3) repeating step 2 until all patterns belong to the same cluster. The algorithm stops when $n - 1$ merges have been performed, which is when all patterns have been connected in the same cluster. The proximity measure between clusters in the second step distinguishes between the different HAC linkage algorithms, such as Single-Link , Average-Link, Complete-Link, among others. In SL, the proximity between any two clusters is the the dissimilarity between their closest patterns.

An interesting property of SL, which will be relevant further on, is its equivalence with a Minimum Spanning Tree (MST) [9]. In graph theory, a MST is a tree that connects all vertices together while minimizing the sum of all the distanced between them. In this context, the edges of the MST are the distances between the patterns and the vertices are the patterns themselves. A MST contains all the information necessary to build a SL dendrogram. One of the advantages of using an MST based clustering is that it processes only non-zero values while a typical SL algorithm will process all pair-wise proximities, even if they are null.

E. Programming for GPUs with CUDA

Parallel processing with Graphics Processing Units was one of the lines of research pursued. Implemented work for the GPU used the Compute Unified Device Architecture (CUDA). This section will introduce a very brief overview of the main concepts to keep in mind for programming GPUs with CUDA.

A GPU is comprised by one or several streaming processors (or multiprocessor). Each of these multiprocessors contains several simpler processors, each of which execute the same instruction at the same time at any given time. In the CUDA programming model, the basic unit of computation is a *thread*. Threads are grouped into *blocks* which are part of the block *grid*. The number of threads in a block is typically higher than the number of processors in a multiprocessor. For that reason, the hardware automatically divides the threads from a block into smaller batches, called *warps*. The computation of one block is independent from other blocks and each block is scheduled to one multiprocessor.

Depending on the architecture, GPUs have several types of memories. Accessible to all processors (and threads) are the global memory, constant memory and texture memory, of which the last two are read-only. Blocks share a smaller but significantly faster memory called shared memory, which is a memory inside a multiprocessor to which all processors have access to, enabling inter-thread communication inside a block. Finally, each thread has access to local memory, which resides in global memory space and has the same latency for read and write operations. However, if the thread is using only single variables or constant sized arrays, it uses register space, which is very fast.

Typically, CUDA applications follow the same flow. First, the host starts by transferring any necessary data to the device memory. The next step is selecting the *kernel* (the function that will run on the GPU processors) and the thread topology (configuration of threads in a block and blocks in the grid). The set-up phase is followed by the computation phase in the GPU. Finally, the host will transfer back the results from the device.

III. Optimization of the production step of EAC

The main contribution for the optimization of the production of clustering ensemble is the implementation of a parallel K-Means for GPUs. Several parallel implementations of this algorithm for the GPU exist in literature [4], [19], [22] and all report speed-ups relative to their sequential counterparts.

The implementation of the present work followed the version in [23], which only parallelizes the labeling stage. This was further motivated from empirical data that suggested the average theoretical maximum speed-up for the labeling stage was 880, whereas for the the update phase the maximum was only 1.5. The CUDA implementation of the labeling step starts by transferring the data to the GPU (pattern set and initial centroids) and allocates space for the labels and corresponding distances. Still, several parameters are accessible to the user, such as the shape of the block of threads and grid of blocks, and also how many patterns to process per thread. The computation of a label for one pattern is done by iteratively computing the distance from the pattern to each centroid and storing the label and the distance corresponding to the closest centroid to that pattern. Finally, the labels and distances are sent back to the host for computation of the new centroids. The implementation of the centroid computation starts by counting the number of patterns attributed to each centroid. Afterwards, it checks if there are any "empty" clusters, i.e. if there are centroids that are not the closest ones to any pattern. Dealing with empty clusters is important because the target use expects that the output number of clusters be the same as defined in the input parameter. Centroids corresponding to empty clusters will be the patterns that are furthest away from their centroids. Any other centroid c_i will be the mean of the patterns that were labeled i.

IV. Optimization of the combination step of EAC

Space complexity is the main challenge building the co-association matrix. A complete pair-wise co-association matrix has $O(n^2)$ complexity but can be reduced to $O(\frac{n(n-1)}{2})$ without loss of information, due to its symmetry. Still, these complexities are rather high when large datasets are contemplated, since it becomes impossible to fit these "conventional" co-association matrices in main memory. Two solutions in literature address this challenge. [8] approaches it by using a k-Nearest Neighbor approach, only considering associations between the k closest neighbors of each pattern. [16] approaches the problem by exploiting the sparse nature of the co-association matrix for reducing space complexity, but doesn't cover the efficiency of building a sparse matrix or the space overhead associated with sparse data structures. The effort of the present work was focused on further exploiting the sparse nature of EAC, building on previous insights and exploring the topics that literature has neglected so far.

Building a non-sparse matrix is easy and fast since the memory for the whole matrix is allocated and indexing the matrix is direct. When using sparse matrices, neither is true. In the specific case of EAC, there is no way to know what is the number of associations the co-association matrix will have which means it is not possible to pre-allocate the memory to the correct size of the matrix. This translates in allocating the memory gradually which may result in fragmentation, depending on the implementation, and more complex data structures, which incurs significant computational overhead. For building a matrix, the DOK (Dictionary of Keys) and LIL (List of Lists) formats are recommended in the documentation of the SciPy [13] scientific computation library. These were briefly tested on simple EAC problems and not only was their execution time several orders of magnitude higher than a traditional fully allocated matrix, but the overhead of the sparse data structures resulted in a space complexity higher than what would be needed to process large datasets. For operating over a matrix, the documentation recommends converting from one of the previous formats to either CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column). Building with the CSR format had a low space complexity but the execution time was much higher than either one of the other sparse formats.

The bad performance of the above sparse formats combined with the fact that no relevant literature was found on efficiently building sparse matrices, led to the design and implementation of a novel strategy for a CSR matrix specialized to the EAC context, the **EAC CSR**.

A. Underlying structure of EAC CSR

The first step of EAC CSR is making an initial assumption on the maximum number of associations *max_assocs* that each pattern can have. A possible rule is

$$max\_assocs = 3 \times bgs$$

where bgs is the biggest cluster size in the ensemble. The biggest cluster size is a good heuristic for trying to predict the number of associations, since it is the limit of how many associations each pattern will have in any partition of the clustering ensemble. Furthermore, one would expect that the neighbors of each pattern will not vary significantly, i.e. the same neighbors will be clustered together repeatedly in many partitions. This scheme for building the matrix uses 4 supporting arrays:

- **indices** - an array of size $n \times max\_assocs$ that stores the columns of each non-zero value of the matrix, i.e. the destination pattern to which each pattern associates with;
- **data** - an array of size $n \times max\_assocs$ that stores all the non-zero association values;
- **indptr** - an array of size n where the *i-th* element is the pointer to the first non-zero value in the *i-th* row.
- **degree** - an array of size n that stores the number of non-zero values of each row.

Each pattern (row) has a maximum of *max_assocs* pre-allocated that it can use to fill with new associations. New associations that exceed this pre-allocated maximum are discarded. The *degree* array is the one keeping track of the number of associations of each pattern. Throughout this section, the interval of the *indices* array corresponding to a specific pattern (row) is referred to as that pattern's *indices* interval. If it is said that an association is added to the end of the *indices* interval, this refers to the beginning of the part of the interval that is free for new associations. The term indices is used either in the context of the array, in which case it will appear as *indices*, or as the plural of index, in which case it will appear as indices. Furthermore, it should be noted that this method assumes that the clusters received come sorted in a increasing order.

B. Updating the matrix with partitions

The first partition is inserted in a special way. Since it is the first and the clusters are sorted, it is a matter of copying the cluster to the *indices* interval of each of its member patterns, excluding self-associations. The *data* array is set to 1 on the positions where the associations were added. Because it is the fastest partition to be inserted, when the whole ensemble is provided at once, the first partition is picked to be the one with the least amount of clusters (more patterns per cluster) so that each pattern gets the most amount of associations in the beginning (on average). This increases the probability that any new cluster will have more patterns that correspond to already established associations.

For the remaining partitions, the process is different. For each pattern in a cluster it is necessary to add or increment the association to every other pattern in the cluster. However, before adding or incrementing any new association, it is necessary to check if it already exists. This is done using a binary search in the *indices* interval. This is necessary because it is not possible to index directly a specific position of a row in the CSR format. Since a binary search is performed $(ns-1)^2$ times for each cluster, where ns is the number of patterns in any given cluster, the indices of each pattern must be in a sorted state at the beginning of each partition insertion.

C. Keeping the indices sorted

The implemented strategy results from the observation that, if one could know in which position each new association should be inserted, it would be possible to move all old associations to their final sorted positions and insert the new ones in an efficient manner with minimum number of comparisons (and thus branches in the execution of the code). For this end, the implemented binary search returns the index of the searched value (key) if it is found or the index for a sorted insertion of the key in the array, if it is not found. New associations are stored in two auxiliary arrays of size max_assocs: one (*new_assoc_ids*) for storing the patterns of the associations and the other (*new_assoc_idx*) to store the indices where the new associations should be inserted (the result of the binary search). The process is illustrated with an example in Fig. 1.

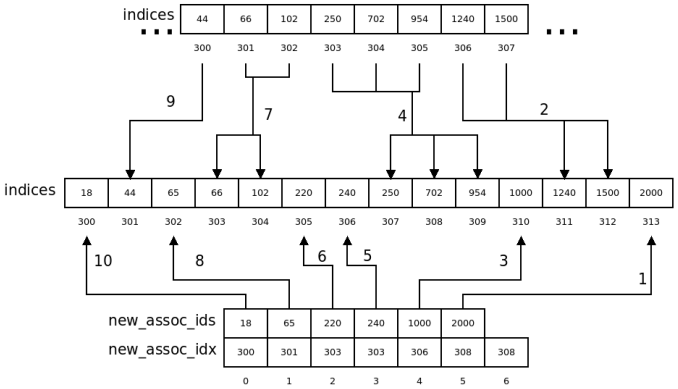

Fig. 1. Inserting a cluster from a partition in the co-association matrix. The arrows indicate to where the indices are moved. The numbers indicate the order of the operation.

The new associations have to be added to the pattern's *indices* interval in a sorted manner. The number of associations corresponding to the i-th pattern (*degree[i]*) is incremented by the amount of new associations to be added. During the sorting process a pointer to the current index to add associations o_ptr is kept (it is initialized to the new total number of associations of a pattern). The sorting mechanism looks at two consecutive elements in the *new_assoc_idx* array, starting from the end. If the i-th element of the *new_assoc_idx* array is greater or equal than the $(i-1)$-th element, then all the associations in the *indices* array between them (including the first element) are shifted to the right by i positions. Then, or in case the comparison fails, the $(i-1)$-th element of the *new_assoc_ids* is copied to the *indices* array in the position specified by *o_ptr*. The *o_ptr* pointer is decremented anytime an association is

written in the *indices* array. This showed to be roughly twice as fast as using an implementation of *quicksort*.

D. EAC CSR Condensed

A further reduction of space complexity in the EAC CSR scheme is possible by building only the upper triangular, since this completely describes the co-association matrix. This means that the amount of associations decreases as one goes further down the matrix. Instead of pre-allocating the same amount of associations for each pattern, the pre-allocation follows the same pattern of the number of associations, effectively reducing the space complexity. The strategy used was a linear one, where the first 5% of patterns have access to 100% of the the estimated maximum number of associations, the last pattern has access to 5% of that value and the number available for the patterns in between decreases linearly from 100% to 5%.

V. Optimization of the recovery step of EAC

Two candidates were considered for the final step of EAC. A SL based on a GPU version of MST [20] was implemented. Although the GPU MST algorithm showed speed-up on benchmark datasets, the co-association matrix is typically less sparse than those, which made any speed-up impossible. External algorithms were the second candidate solution. This solution is based on storing the co-association matrix on disk, performing the expensive computation (memory and speed wise) of *argsort* and then processing the matrix in batches until the final MST is extracted.

A. Kruskal's algorithm and implementation

The MST algorithm used was Kruskal. The original paper of Kruskal's [14] algorithm describes three approaches for finding a MST. The SciPy library offers an efficient implementation for the following construct (taken directly from the original paper of Kruskal's algorithm):

> CONSTRUCTION A. Perform the following step as many times as possible: Among the edges of G not yet chosen, choose the shortest edge which does not form any loops with those edges already chosen. Clearly the set of edges eventually chosen must form a spanning tree of G, and in fact it forms a shortest spanning tree.

If the graph G if connected, the algorithm will stop before processing all edges when $|V|-1$ edges are added to the MST, where V is the set of edges. This implementation works on a sparse matrix in the CSR format.

One of the main steps of the implementation is computing the order of the edges of the graph without sorting the edges themselves, an operation called *argsort*. To illustrate this, the *argsort* operation on the array $[4, 5, 2, 1, 3]$ would yield $[3, 2, 4, 0, 1]$ since the the smallest element is at position 3 (starting from 0), the second smalled at position 2, etc. This operation is much less time intensive than computing the shortest edge at each iteration. However, the total space used is typically 8 times larger for EAC since the data type of the weights uses only one byte and the number of associations is very large, forcing the use of a 8 byte integer for the *argsort* output array. This is the real motivation to store the co-association matrix in disk and use an external sorting algorithm.

B. Kruskal's implementation with an external sorting algorithm

The *PyTables* library [2], which is built on top of the *HDF5* format [21], was used for storing the graph, performing the external sorting for the *argsort* operation and loading the graph in batches for processing. This implementation starts by storing the CSR graph to disk. However, instead of saving the *indptr* array directly, it stores an expanded version of the same length as the *indices* array, where the i-th element contains the origin vertex (or row) of the i-th edge. This way, a binary search for discovering the origin vertex becomes unnecessary.

Afterwards, the *argsort* operation is performed by building a completely sorted index (CSI) [3] of the *data* array of the CSR matrix. It should be noted that the arrays themselves are not sorted. Instead, the CSI allows for a fast indexing of the arrays in a sorted manner (according to the order of the edges). The process of building the CSI has a very low main memory usage and can be disregarded in comparison to the co-association matrix.

The SciPy implementation of Kruskal's algorithm was modified to work with batches of the graph. This was easily implemented just by making the additional data structures used in the building of MST persistent between iterations. The new implementation loads the graph in batches and in a sorted manner, e.g. first load a batch of the 1000 shortest edges, then a batch of the next 1000 shortest edges, etc. Each batch must be processed sequentially since the edges must be processed in a sorted manner, which means there is no possibility for parallelism in this process. Typically, the batch size is a very small fraction of the size of the edges, so the total memory usage for building the MST is overshadowed by the size of the co-association matrix. The time complexity for building the CSI is higher that of computing the *argsort* operation, but the formal time complexity is not reported in the source [3]. As an example, for a $500\,000$ vertex graph the SL-MST approach took 54.9 seconds while the external memory approach took 2613.5 seconds - 2 orders of magnitude higher.

VI. Results

Results presented here originated from one of three machines, that will be referred to as Alpha, Bravo and Charlie. The main specifications of each machine are as follows:

- **Alpha**: 4 GBs of main memory, a 2 core Intel i3-2310M 2.1GHz CPU and a NVIDIA GT520M with 1 GB;
- **Bravo**: 32 GBs of main memory, a 6 core Intel i7-4930K 3.4GHz CPU and a NVIDIA Quadro K600 with 1 GB;
- **Charlie**: 32 GBs of main memory, a 4 core Intel i7-4770K 3.5GHz CPU and a NVIDIA K40c with 12 GB.

A. GPU Parallel K-Means

Both sequential and parallel versions of K-Means were executed over a wide spectrum of datasets varying number of patterns, features and centroids. All tests were executed on machine Charlie and the block size was maintained constant at 512. Whenever the number of clusters was superior to 70% of the number of patterns, that particular test case was not executed.

Observing Figures 2 and 3, it is clear that the number of patterns, features and clusters influence the speed-up. For the simple case of 2 dimensions (Fig 2), the speed-up increases with the number of patterns. However, there is no speed-up when the overall complexity of the datasets is low. For 2 clusters, there is no speed-up before 100 000 patterns. And even after that mark, the speed-up is not significant. On the other hand, for a large number of clusters, there is speed-up for any number of patterns executed. Not only that, that speed-up is highest of any other case with inferior number of clusters. The reason for this is that the total amount of work increases linearly with the number of clusters but is diluted by the number of threads that can execute simultaneously.

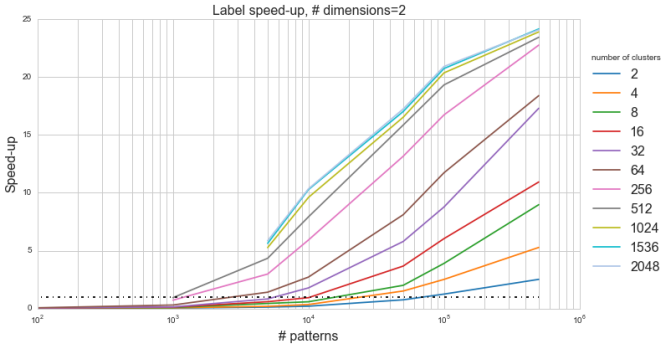

Fig. 2. Speed-up of the labeling phase for datasets of 2 dimensions and varying the number of patterns and clusters. The dotted black line represents a speed-up of one.

However, as the dimensionality increases (observe Fig. 3), the speed-up increases until a certain number of patterns and then decreases. Here, the initial number of patterns for which there is a speed-up is lower than in the low dimensionality case and the number of clusters plays less an influence on the speed-up. It is believed that the reason for this is related to the implementation itself. The current parallel implementation does not use shared memory, which is fast. As such, for every computation, each thread fetches the relevant data from global memory which is significantly slower. As the number of dimensions increases, the amount of data that each thread must fetch also increases. Furthermore, since the number of dimensions affects both data points and centroids, if the number of dimensions increases by 2 the number of fetches to memory increases by 4. So, the speed-up increases with the dataset complexity until a point where the number of fetches to memory starts having a very significant effect on the execution time and it decreases close to 50%.

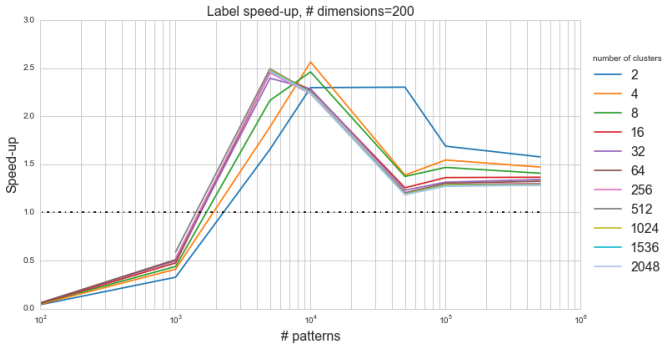

Fig. 3. Speed-up of the labeling phase for datasets of 200 dimensions and varying cardinality and number of clusters. The dotted black line represents a speed-up of one.

B. Validation with original implementation

The results of the original version of EAC, implemented in Matlab, are compared with those of the proposed solution. Several small datasets, chosen from the datasets used in [16] and taken from the UCI Machine Learning repository [15], were processed by the two versions of EAC. Furthermore, since the generation of the ensemble is probabilistic and can change the results between runs, the proposed version is processed with the ensembles created by the original version as well. This guarantees that the combination and recovery phases of EAC, which are deterministic when using SL, are equivalent to the original. All data in this section refers to processing done in machine Alpha. Table I presents the difference between the accuracies of the two versions. Analyzing these results, it is apparent that the difference is minimal, most likely due the original version using Matlab and the proposed using Python. Furthermore, a speed-up as low as 6 and as high as 200 was obtained over the original version in the various steps, even for these small datasets.

TABLE I
DIFFERENCE BETWEEN ACCURACIES FROM THE TWO IMPLEMENTATIONS OF EAC, USING THE SAME ENSEMBLE. ACCURACY WAS MEASURED USING THE H-INDEX.

dataset	Difference between accuracies of implementations True number of clusters	Lifetime criteria
breast_cancer	4.948755e-06	2.825769e-06
ionosphere	1.652422e-06	1.452991e-06
iris	3.333333e-06	3.333333e-06
isolet	1.038861e-07	4.084904e-07
optdigits	3.795449e-06	1.480513e-06
pima	3.333333e-06	3.333333e-06
pima_norm	4.166667e-07	4.166667e-07
wine_norm	1.123596e-07	1.910112e-06

C. Set-up for large dataset testing

The results here presented refer to a synthetic large dataset comprised by a mixture of 6 Gaussians, where 2 pairs are overlapped and 1 pair is touching. This dataset was sampled into several smaller datasets to cover a wider range of number of patterns.

Different rules for computing the K_{min}, different co-association matrix formats and different approaches for the final clustering will be mentioned. The different rules and their aliases are presented in Table II. The different formats for the co-association matrix are the *full* (for fully allocated $n \times n$ matrix), *full condensed* (for a fully allocated $\frac{n(n-1)}{2}$ array to build the upper triangular matrix), *sparse complete* (for EAC CSR), *sparse condensed const* (for EAC CSR building only the upper triangular matrix) and *sparse condensed linear* (for EAC CSR condensed). The different approaches for the final clustering are *SLINK* [18], *SL-MST* (for using the Kruskal implementation in SciPy) and *SL-MST-Disk* for the modified version that performs an external memory sort.

TABLE II
DIFFERENT RULES FOR COMPUTING K_{min} AND K_{max}. n IS THE NUMBER OF PATTERNS AND sk IS THE NUMBER OF PATTERNS PER CLUSTER.

Rule	K_{min}	K_{max}
sqrt	$\frac{\sqrt{n}}{2}$	$\sqrt{n}$
2sqrt	$\sqrt{n}$	$2\sqrt{n}$
sk=sqrt2	$sk = \frac{\sqrt{n}}{2}$	$1.3K_{min}$
sk=300	$sk = 300$	$1.3K_{min}$

The experiment that generated the results of these section was set up as follows. A large dataset was generated. The dataset was sampled uniformly to produce a smaller dataset with the desired number of patterns. A clustering ensemble was produced (production phase) for each of these smaller datasets and for each of the rules, using K-Means. From each ensemble, co-association matrices of every applicable format were built (combination phase). A matrix format was not applicable when the dataset complexity would make its correspondent co-association matrix too big to fit in main memory. The final clustering (recovery phase) was also done for each of the matrix formats. SL-MST was not executed if its space complexity was too big to fit in main memory. Furthermore, the combination and recovery phases were repeated several times for smaller datasets for statistical relevant of the execution times, so as to make the influence of any background process less salient. For big datasets, the execution times are big enough that the influence of background processes is negligible. All results presented here originated from machine Bravo. The same analysis that is presented here was performed on a similar dataset with separated Gaussians, from which similar conclusions were drawn.

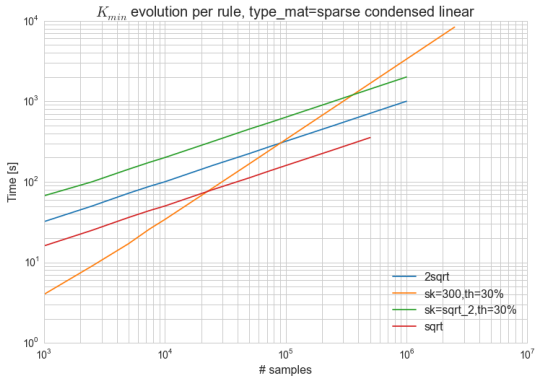

Fig. 4. Evolution of K_{min} with cardinality for different rules.

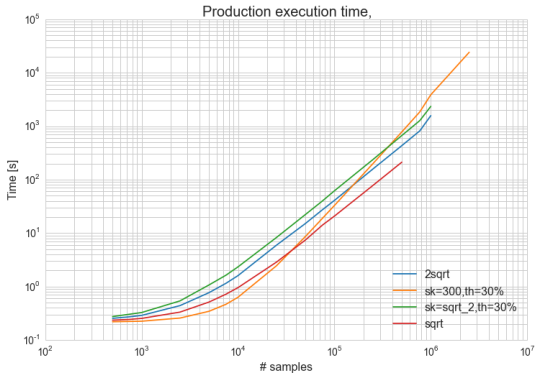

Fig. 5. Execution time for the production of the clustering ensemble.

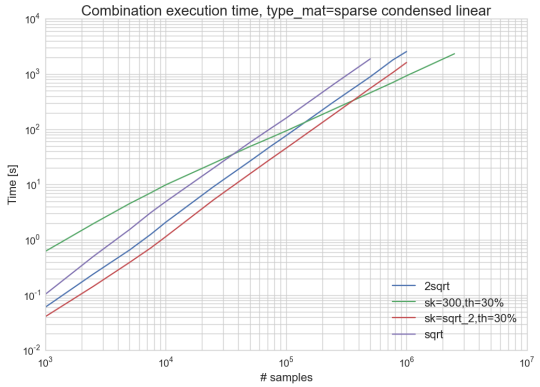

Fig. 6. Execution time for building the co-association matrix from ensemble with different rules.

D. Execution times

Execution times are related with the K_{min} parameter, whose evolution is presented in Fig. 4. Rules *sqrt*, *2sqrt* and *sk=sqrt2* never intersect but rule $sk = 300$ intersects all of them, finishing with the highest K_{min}. Observing Fig. 5, one can see that the same thing happens to the production execution time associated with the $sk = 300$ rule and the inverse happens to the combination time (Fig. 6). A higher K_{min} means more centroids for each K-Means run to compute, so it is not surprising that the execution time for computing the ensemble increases as K_{min} increases.

Fig. 7 shows the execution times on a longitudinal study for optimized matrix formats. It is clear that the sparse formats are significantly slower than the fully allocated ones, specially for smaller datasets. The *full condensed* format usually takes close to half the time than the *full* format, which is natural given that it performs half the operations. Idem for the *sparse condensed* formats compared to the *sparse complete*. The big discrepancy between the sparse and full formats is due to the fact that the former needs to do a binary search at each association update and needs to keep the internal sparse data structure sorted.

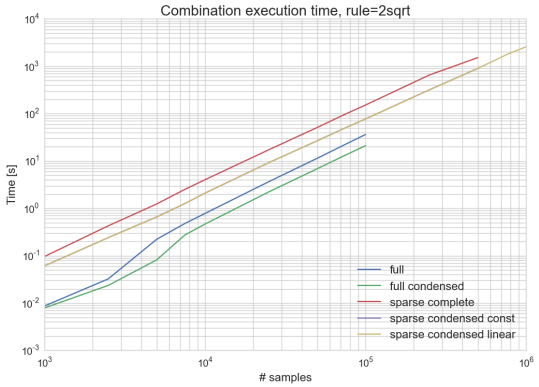

Fig. 7. Execution time for building the co-association matrix with different matrix formats.

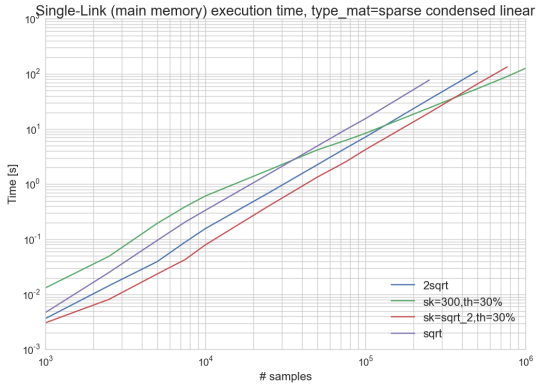

Fig. 9. Comparison between the execution times of SL-MST for different rules.

E. Performance comparison between SLINK, SL-MST and SL-MST-Disk

The clustering times of the different methods of SL discussed previously (SLINK, SL-MST and SL-MST-Disk) are presented in Figures 8 and 9. The SL-MST-Disk method is significantly slower than any of the other methods. This is expected, since it uses the hard drive which has very slow access times compared to main memory. SL-MST is faster than SLINK, since it processes zero associations while SL-MST takes advantage of a graph representation and only processes the non-zero associations. In resemblance to what happened with combination times, the condensed variants take roughly half the time has their complete counterparts. This is expected, since SL-MST and SL-MST-Disk over condensed co-association matrices only process half the number of associations. Although this is not depicted, SLINK takes roughly the same time for every rule, which means K_{min} has no influence. This comes as no surprise, since SLINK processes the whole matrix, regardless of its association sparsity. The same rationale can be applied to SL-MST, where different rules can have significant influence over execution time, since they change the total number of associations. As with the combination phase, the execution time referent to the *sk=300* rule started with the greatest time and decreased with an increase in the number of patterns until it was the fastest.

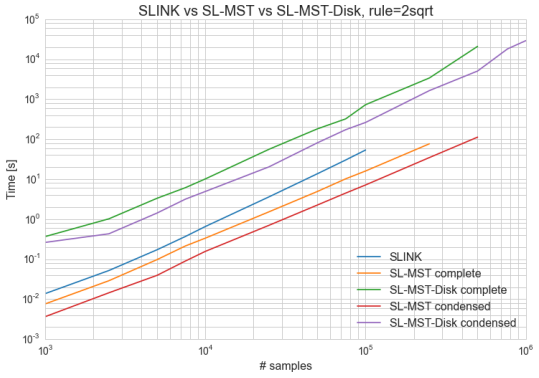

Fig. 8. Comparison between the execution times of the three methods of SL. SLINK runs over fully allocated condensed matrix while SL-MST and SL-MST-Disk run over the condensed and complete sparse matrices.

The execution times of all phases combined are presented in Figures 10 and 11. The results are presented for the *sparse condensed linear* format but the remaining results follow the same pattern. It is interesting to note that, when using the SL-MST method in the recovery phase, the execution time for three of the rules do not differ much. This is due to a sort of balancing between a slowing down of the production phase and a speeding up of the combination and recovery phases as the K_{min} increases at a higher rate for $sk = 300$ than for other rules. This is not observed for the $sqrt$ rule as K_{min} is always low enough that the total time is always dominated by the combination and recovery phases. The same does not happen when using the SL-MST-Disk method, as the total time is completely dominated by the recovery phase. This is clear, since the results in Fig. 11 follow a pattern similar to that presented in Fig. 9.

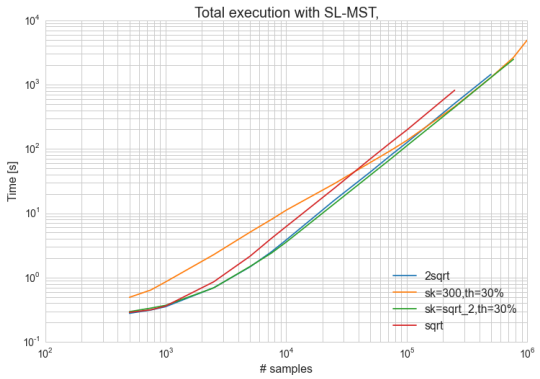

Fig. 10. Execution times for all phases combined, using SL-MST in the recovery phase.

F. Analysis of the number of associations

The sparse nature of EAC has been referred to before and is clearer in Fig. 12. This figure shows the association density, i.e. number of associations relative to the n^2 associations in a full matrix. The *full condensed* format has a constant density of 49.5%. Idem for the *sparse complete* and *sparse condensed* formats, as long as no associations are discarded. The overall tendency is for the density to decrease as the number of patterns of the dataset increases. This is to be

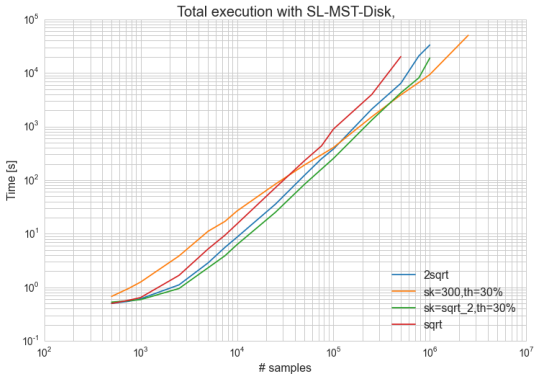

Fig. 11. Execution times for all phases combined, using SL-MST-Disk in the recovery phase.

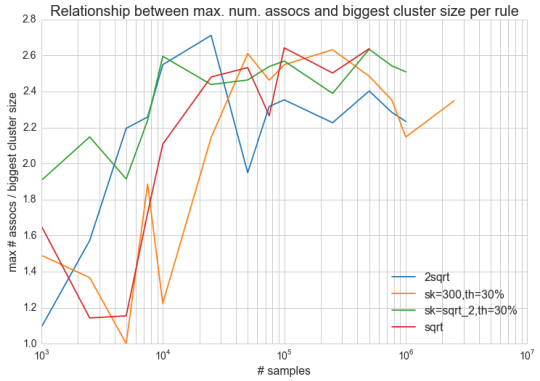

Fig. 13. Maximum number of associations of any pattern divided by the number of patterns in the biggest cluster of the ensemble.

expected since the *full* matrix grows quadratically. Besides, it would be expected that the same associations would be grouped together more frequently in partitions and simply make previous connections stronger instead of creating new ones, if the relationship between the number of patterns and K_{min} is constant.

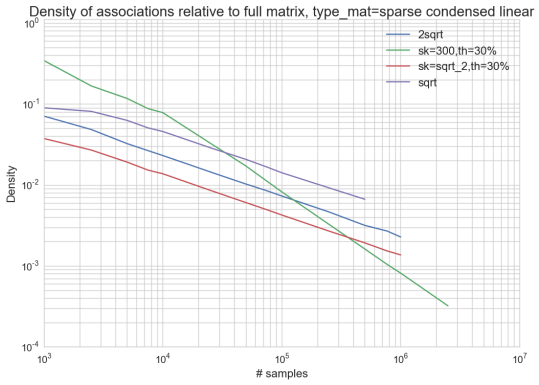

Fig. 12. Density of associations relative to the full co-association matrix, which hold n^2 associations.

Predicting the number of associations before building the co-association matrix is useful for coming up with combination schemes that are both memory and speed efficient. It was stated before that the biggest cluster size in any partition of the ensemble is a good parameter for this end. Fig. 13 presents the relationship between the biggest cluster size and the maximum number of associations of any pattern. These ratio increases with the number of patterns in the beginning, but as the number of patterns increases it never goes over 3.

However, the number of features of the used datasets is rather reduced. It might be the case that this ratio would increase with the number of features, since there would be more degrees where the clusters might include other neighbors. With this in mind, further studies ranging a wider spectrum of datasets should yield more enlightening conclusions or reinforce those presented here.

G. Space complexity

As explained previously, the allocated space for the space formats is based on a prediction that uses the biggest cluster size of the ensemble. This allocated space is usually more than what is necessary to store the total number of associations, to keep a safety margin. Furthermore, the CSR sparse format, on which the EAC CSR strategy is based, requires an array of the same size of the predicted number of associations. This overhead may in fact make the sparse format pre-allocate more associations than are actually possible for some rules and in very small datasets. Still, the allocated number of associations becomes a very small fraction compared to the *full* matrix as the dataset complexity increases, which is the typical case for using a sparse format. The actual memory used is presented in Fig. 14. Here, the data types used play a big role in the amount of memory that is required. The associations can be stored in a single byte, since the number of partitions is usually less than 255. This means that the memory used by the fully allocated formats is n^2 and $\frac{n(n-1)}{2}$ Bytes for the complete and condensed versions, respectively. In the sparse formats, the values of the associations are also stored in an array of unsigned integers of 1 Byte. However, an array of integers of 4 bytes of the same size must also be kept to keep track of the destination pattern each association belongs to. Besides, one other array of integers of 8 bytes is kept but it is negligible compared to the other two arrays. The impact of the data types can be seen for smaller datasets where the total memory used is actually significantly higher than that of the *full* matrix. It should be noted that this discrepancy is not as high for other rules as for $sk = 300$. Still, the sparse formats, and in particular the condensed sparse format, is preferred since the memory used for large datasets is a small fraction of what would be necessary if using any of the fully allocated formats.

H. Accuracy

The accuracy of each produced clustering was measured using the Consistency Index. The number of clusters was chosen with the lifetime criteria. The accuracy for all rules, matrix formats and SL approaches was the same for each of the datasets, with the notable exception of the $sk = 300$ rule, for which the partitions in ensembles of datasets smaller than 1000 patterns had less clusters than the true amount of clusters. For small datasets the accuracy was roughly 84%, since only one pair of Gaussians were overlapped at that point. The accuracy

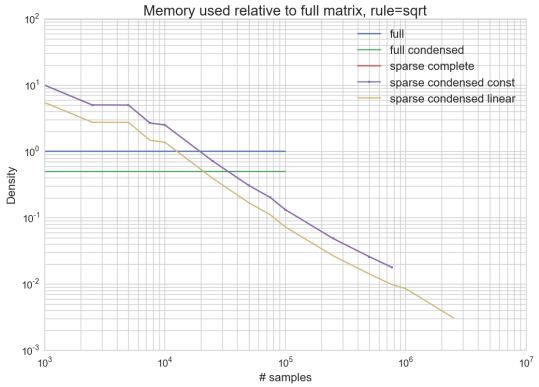

Fig. 14. Memory used relative to the full n^2 matrix. The *sparse complete* and *sparse condensed const* curves are overlapped.

lowers to around 66% for datasets higher than 7500, since 2 pairs of Gaussians are now overlapped. There were two datasets (of $75\,000$ and $2\,500\,000$ patterns) for which the accuracy was 50%, due to existence of patterns there made a "bridge" between the other pair of Gaussians.

VII. Conclusion

The main goal of scaling the EAC method for larger datasets than was previously possible was achieved. In the process, it was also optimized for smaller datasets. The EAC method is composed by three steps and, to scale the whole method, each step was optimized separately but maintaining interoperability. In essence, the main contributions to the EAC method, by step, were the GPU parallel K-Means in the production step, the EAC CSR strategy in the combination step and the SL-MST-Disk in the recovery step. These contributions together allow for the application of the EAC method to datasets whose complexity was not handled by the original implementation. However, it should be noted that the present GPU parallel K-Means implementation yielded worst results than those found in literature for similar datasets. The main reason for this are the burdens imposed by the language used and the absence of possible optimizations, such as shared memory utilization. Finally, new K_{min} rules to optimize sparsity were proposed, accompanied with a deeper understanding on how these affect EAC properties.

References

[1] C. C. Aggarwal and C. K. Reddy, *Data clustering: algorithms and applications*. CRC Press, 2013.

[2] F. Alted, I. Vilata, and Others, "PyTables: Hierarchical Datasets in Python." [Online]. Available: http://www.pytables.org/

[3] F. Altet i Abad and I. V. Balaguer, "OPSI : The indexing system of PyTables 2 Professional Edition," pp. 1–27, 2007.

[4] H. T. Bai, L. L. He, D. T. Ouyang, Z. S. Li, and H. Li, "K-means on commodity GPUs with CUDA," *2009 WRI World Congress on Computer Science and Information Engineering, CSIE 2009*, vol. 3, pp. 651–655, 2009.

[5] E. Casper, C.-C. Hung, E. Jung, and M. Yang, "A Quantum-Modeled K-Means Clustering Algorithm for Multi-band Image Segmentation," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, vol. 1, no. 3, 2012, pp. 158–163.

[6] A. Fred, "Finding consistent clusters in data partitions," *Multiple classifier systems*, pp. 309–318, 2001.

[7] A. N. L. Fred and A. K. Jain, "Data clustering using evidence accumulation," *Object recognition supported by user interaction for service robots*, vol. 4, 2002.

[8] ——, "Combining multiple clusterings using evidence accumulation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 6, pp. 835–850, 2005.

[9] J. C. Gower and G. J. S. Ross, "Minimum Spanning Trees and Single Linkage Cluster Analysis," *Journal of the Royal Statistical Society*, vol. 18, no. 1, pp. 54–64, 1969.

[10] D. Horn and A. Gottlieb, "The Method of Quantum Clustering." *NIPS*, no. 1, 2001.

[11] a. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264–323, 1999.

[12] A. K. Jain, "Data clustering: 50 years beyond K-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, 2010.

[13] E. Jones, T. Oliphant, P. Peterson, and Others, "SciPy: Open source scientific tools for Python." [Online]. Available: http://www.scipy.org/

[14] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[15] M. Lichman, "{UCI} Machine Learning Repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[16] A. Lourenço, A. L. N. Fred, and A. K. Jain, "On the scalability of evidence accumulation clustering," *Proceedings - International Conference on Pattern Recognition*, vol. 0, pp. 782–785, 2010.

[17] M. Meila, "Comparing clusterings by the variation of information," *Learning theory and Kernel machines: 16th Annual Conference on Learning Theory and 7th Kernel Workshop, COLT/Kernel 2003, Washington, DC, USA, August 24-27, 2003: proceedings*, p. 173, 2003.

[18] R. Sibson, "SLINK: an optimally efficient algorithm for the single-link cluster method," *The Computer Journal*, vol. 16, no. 1, pp. 30–34, 1973.

[19] J. Sirotkovi, H. Dujmi, and V. Papi, "K-Means Image Segmentation on Massively Parallel GPU Architecture," pp. 489–494, 2012.

[20] C. d. S. Sousa, A. Mariano, and A. Proença, "A Generic and Highly Efficient Parallel Variant of Boruvka s Algorithm," 2015.

[21] The HDF Group. Hierarchical Data Format, version 5.

[22] M. Zechner and M. Granitzer, "Accelerating k-means on the graphics processor via CUDA," *Proceedings of the 1st International Conference on Intensive Applications and Services, INTENSIVE 2009*, pp. 7–15, 2009.

[23] ——, "K-Means on the Graphics Processor: Design And Experimental Analysis," *International Journal on Advances in System and Measurements*, vol. 2, no. 2, pp. 224–235, 2009.