Efficient Evidence Accumulation Clustering for large datasets/big data

Diogo Silva[1], Ana Fred[2] and Helena Aidos[2]
[1]*Portuguese Air Force Academy, Sintra, Portugal*
[2]*Instituto de Telecomunicacões, Instituto Superior Técnico, Lisboa, Portugal*
dasilva@academiafa.edu.pt, {afred, haidos}@lx.it.pt

Keywords: Clustering methods, EAC, K-Means, MST, GPGPU, CUDA, Sparse matrices, Single-Link

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.

1 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. Of these, cluster analysis is an interesting tool, since it typically does not rely on external information about the data. Clustering algorithms explore the structure of the data by organizing it into clusters.

A vast body of work on clustering algorithms exists (Jain, 2010), but usually different methods are suited to datasets of different characteristics and are not able to discover all possible cluster shapes. Inspired by the work on sensor fusion and classifier combination, ensemble clustering approaches (Fred, 2001; Fred and Jain, 2002; Strehl and Ghosh, 2002) have been proposed to address that challenge.

The present works builds on the Evidence Accumulation Clustering (EAC) framework presented by (Fred and Jain, 2002; Fred and Jain, 2005). The underlying idea of EAC 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. Accordingly, EAC tries to find an optimal partition P^* containing k^* clusters that is consistent with and robust to small variations in the ensemble and has a good fit with ground truth, when available. EAC makes no assumption on the number of clusters in each partition in the ensemble. Its approach is divided in 3 steps:

1. **Production** of a clustering ensemble (the evidence);
2. **Combination** of the ensemble into a co-association matrix C, by means of a voting mechanism based on co-occurrences in the clusters of the N partitions in the ensemble:

$$C_{i,j} = \frac{no.\,co-occurences\ in\ cluster}{N}$$

3. **Recovery** of the natural clusters of the data.

The ensemble is usually produced by random initialization of K-Means, to have diversity with the intention of better capturing the underlying structure of the data. That diversity can also be obtained by varying the number of clusters in the ensemble's partitions within an interval $[K_{min}, K_{max}]$. The choice of this interval and its influence on the EAC's properties will be a topic of discussion further ahead.

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, and can be cut at different levels to produce a partition.

EAC is robust, but, currently, its computational complexity restricts its application to small datasets. The two approaches for addressing the space complexity of EAC that have been proposed are (1) exploiting the sparse nature of the co-association matrix (Lourenço et al., 2010) and (2) consider only the k-Nearest Neighbors of each pattern when building the co-association matrix.

The goal of this work is to scale the EAC method to large datasets, using technology found on a typical workstation, by optimizing for both speed and memory usage. We speed-up the production of ensemble by using the power of Graphics Processing Units. The space complexity of the co-association matrix is dealt by exploiting its sparse nature, extending the work of (Lourenço et al., 2010). We choose to use the Single-Link algorithm to speed-up the final clustering for small data and make it applicable for large datasets by using external memory algorithms.

The approaches for optimizing each of the steps are presented in sections 2, 3 and 4 for the production, combination and recovery steps, respectively. The implemented optimizations are tested and the results presented in section 5. Finally, the conclusions can be found in section 6.

2 OPTIMIZATION OF THE PRODUCTION STEP OF EAC

K-Means is typically used for the production of the ensemble. The main contribution for the optimization of the production of clustering ensemble is the implementation of a parallel K-Means for GPUs using NVIDIA's Compute Unified Device Architecture (CUDA).

2.1 K-Means

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, such as K-Means++ (Arthur et al., 2007).

2.2 Programming for GPUs with CUDA

A GPU is comprised by one or several streaming processors (or multiprocessors). Each of these multiprocessors contains several simpler processors, which execute a *thread*, the basic unit of computation in CUDA. 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. CUDA employs a *single-instruction multiple-thread* execution model (Lindholm et al., 2008), where all processors in a multiprocessor execute the same instruction of their respective threads at the same time.

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.

2.3 Parallel K-Means

Several parallel implementations of K-Means for the GPU exist in literature (Bai et al., 2009; Zechner and Granitzer, 2009a; Sirotkovi et al., 2012) and all report speed-ups relative to their sequential counter-

parts. The implementation of the present work followed the version in (Zechner and Granitzer, 2009b), which only parallelizes the labeling stage. This was further motivated from empirical data which suggested that, on average, roughly 98% of the time was spent in the labeling step. 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 distances from patterns to their closest centroids. 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 recomputation, in the CPU, 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.

In the current implementation several parameters can be adjusted by the user. These include the block and grid topology and the number of patterns that each thread should process.

3 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 infeasible to fit these co-association matrices in the main memory of a typical workstation. (Lourenço et al., 2010) approaches the problem by exploiting the sparse nature of the co-association matrix, but doesn't cover the efficiency of building a sparse matrix or the 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 (Jones et al., 2001) 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**.

3.1 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 (n is the number of patterns):

- **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 keeps track of the number of associations of each pattern. Throughout this section, the interval of the *indices* array corresponding to a specific pattern 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.

3.2 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.

3.3 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.

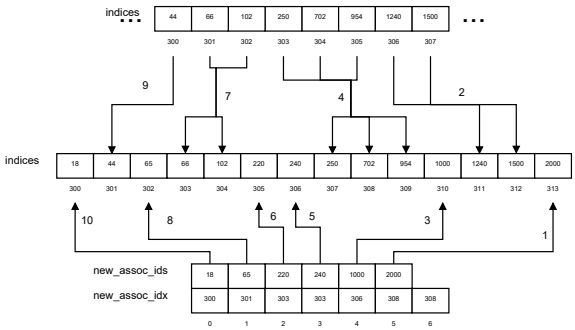

Figure 1: Inserting a cluster from a partition in the co-association matrix. The arrows indicate to where the indices are moved. The numbers indicate the order of the operation.

The binary search operation requires that each pattern's *indices* interval be sorted. Accordingly, new associations corresponding to each pattern are added in a sorted manner. The sorting mechanism looks at the insertion indices of two consecutive new associations in the *new_assoc_idx* array, starting from the end. Whenever two consecutive insertion indices are not the same, it means that old associations must be moved as seen in the example. More specifically, if the *i*-th element of the *new_assoc_idx* array, a, is greater than the $(i-1)$-th element, b, then all the old associations in the index interval $[a,b[$ are shifted to the right by i positions. The *i*-th element will never be smaller than the $(i-1)$-th because clusters are sorted. Afterwards, or when two consecutive insertion indices are the same, the $(i-1)$-th element of the *new_assoc_ids* is inserted in the position indicated by a pointer, *o_ptr*. The *o_ptr* pointer is initiazed with the number of old associations plus the number of new asociations and is decremented anytime an associa-

tion is moved or inserted. This showed to be roughly twice as fast as using an implementation of *quicksort*.

3.4 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%.

4 OPTIMIZATION OF THE RECOVERY STEP OF EAC

4.1 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 (Jain et al., 1999) (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) (Gower and Ross, 1969). In graph theory, a MST is a tree that connects all vertices together while minimizing the sum of all the distances between them. In the context of EAC, 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.

4.2 Kruskal's algorithm and implementation

Kruskal's algorithm was used for computing the MST. The original paper of Kruskal's (Kruskal, 1956) algorithm describes three approaches for finding a MST. The SciPy scientific computing Python library (Jones et al., 2001) 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 motivated the storage of the co-association matrix in disk and usage an external sorting algorithm.

4.3 Kruskal's implementation with an external sorting algorithm

The *PyTables* library (Alted et al.,), which is built on top of the *HDF5* format (The HDF Group,), was used for storing the co-association matrix in graph format, 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) (Altet i Abad and Balaguer, 2007) 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 than that of computing the *argsort* operation, but the formal time complexity is not reported in the source (Altet i Abad and Balaguer, 2007). 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.

5 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.

5.1 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 (e.g. 800 clusters for a dataset with 1000 pattern), that particular test case was not executed.

Observing Figures 2 and **??**, 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 for a superior 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.

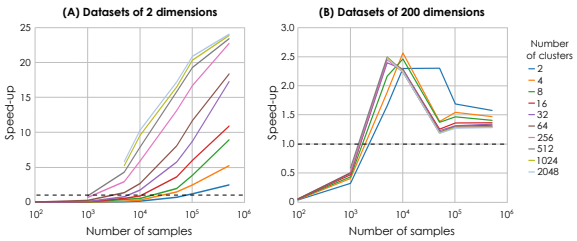

Figure 2: Speed-up of the labeling phase for datasets of 2 dimensions and varying the number of patterns and clusters. The dotted black line represents a speed-up of one.

However, as the dimensionality increases (observe Fig. **??**), 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 and the number of clusters plays less an influence on the speed-up. We believe 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%.

5.2 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 (Lourenço et al., 2010) and taken from the UCI Machine Learning repository (Lichman, 2013), 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 1 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. Moreover, a speed-up as low as 6 and as high as 200 was obtained over the original version in the various steps, including the production step.

Table 1: Difference between accuracy of the two implementations of EAC, using the same ensemble. Accuracy was measured using the H-index (Meila, 2003).

	Number of clusters scheme	
Dataset	Fixed	Lifetime
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

5.3 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 2. 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* (Sibson, 1973), *SL-MST* (for using the Kruskal implementation in SciPy) and *SL-MST-Disk* for the modified version that performs an external memory sort.

Table 2: 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. The number of clusters was chosen with the lifetime criteria (Fred and Jain, 2005). 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.

5.4 Execution times

Execution times are related with the K_{min} parameter, whose evolution is presented in Fig. 3. Rules *sqrt*, *2sqrt* and *sk=sqrt2* never intersect but rule $sk = 300$ intersects all of them, finishing with the highest K_{min}. Observing Fig. 4, 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. **??**). 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.

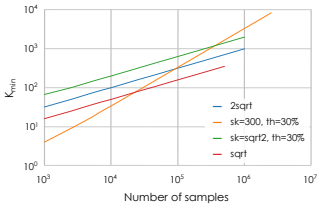

Figure 3: Evolution of K_{min} with the number of patterns for different rules (type of matrix = sparse condensed linear).

Fig. **??** 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.

The clustering times of the different methods of SL discussed previously (SLINK, SL-MST and SL-MST-Disk) are presented in Figures 5 and **??**. 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, 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.

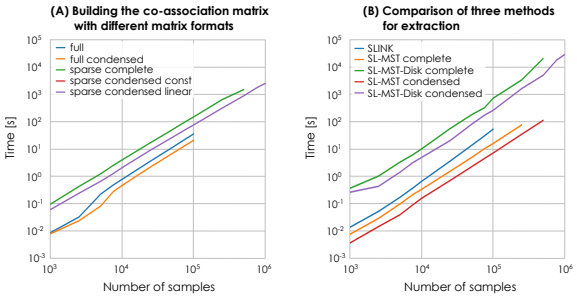

Figure 5: Execution time with the rule 2sqrt for: (A) building the co-association matrix with different matrix formats; and (B) comparing three methods for extraction the final partition: SLINK runs over fully allocated condensed matrix while SL-MST and SL-MST-Disk run over the condensed and complete sparse matrices.

The execution times of all phases combined are presented in Figures 6 and **??**. 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. **??** follow a pattern similar to that presented in Fig. **??**.

5.5 Analysis of the number of associations

The sparse nature of EAC has been referred to before and is clearer in Fig. 7. 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

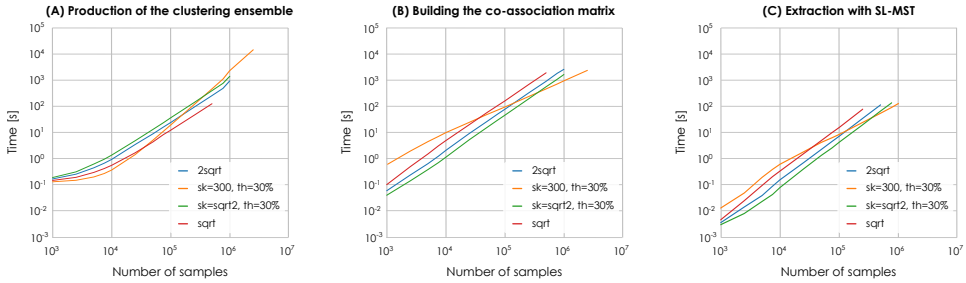

Figure 4: Execution time with different rules for: (A) production of the clustering ensemble; (B) building the co-association matrix; and (C) extraction of the final partition with SL-MST.

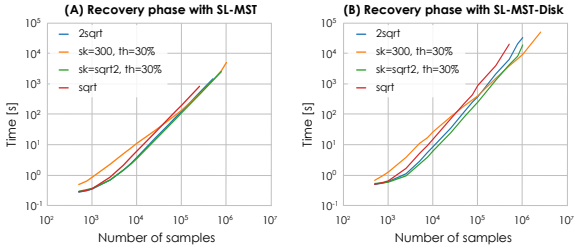

Figure 6: Execution times for all phases combined, using (A) SL-MST and (B) SL-MST-Disk in the recovery phase.

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, 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.

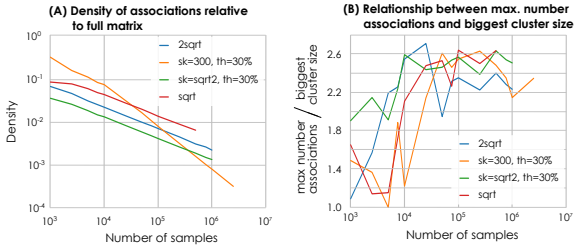

Figure 7: (A) Density of associations relative to the full co-association matrix, which hold n^2 associations. (B) Maximum number of associations of any pattern divided by the number of patterns in the biggest cluster of the ensemble.

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. **??** 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.

5.6 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 preallocate 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. 8. 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.

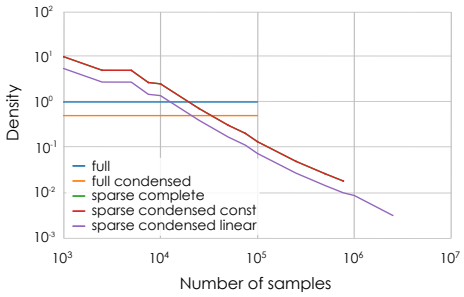

Figure 8: Memory used relative to the full n^2 matrix. The *sparse complete* and *sparse condensed const* curves are overlapped.

6 CONCLUSIONS

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.

Further work involves thorough evaluation in real world.deproblems and datasets. Additional work may include further optimization of the Parallel GPU K-means by using shared memory and a better sorting of the co-association graph in the *SL-MST-Disk*. By using using bigger chunks in main memory and possibly even using the GPU, sorting the co-association graph should become significantly faster.

Acknowledgments

This work was supported by the Portuguese Foundation for Science and Technology, scholarship number SFRH/BPD/103127/2014, and grant PTDC/EEI-SII/7092/2014.

REFERENCES

Alted, F., Vilata, I., and Others. PyTables: Hierarchical Datasets in Python.

Altet i Abad, F. and Balaguer, I. V. (2007). OPSI : The indexing system of PyTables 2 Professional Edition. pages 1–27.

Arthur, D., Arthur, D., Vassilvitskii, S., and Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 8:1027–1035.

Bai, H. T., He, L. L., Ouyang, D. T., Li, Z. S., and Li, H. (2009). K-means on commodity GPUs with CUDA. *2009 WRI World Congress on Computer Science and Information Engineering, CSIE 2009*, 3:651–655.

Fred, A. (2001). Finding consistent clusters in data partitions. *Multiple classifier systems*, pages 309–318.

Fred, A. N. L. and Jain, A. K. (2002). Data clustering using evidence accumulation. *Object recognition supported by user interaction for service robots*, 4.

Fred, A. N. L. and Jain, A. K. (2005). Combining multiple clusterings using evidence accumulation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(6):835–850.

Gower, J. C. and Ross, G. J. S. (1969). Minimum Spanning Trees and Single Linkage Cluster Analysis. *Journal of the Royal Statistical Society*, 18(1):54–64.

Jain, A. K. (2010). Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651–666.

Jain, a. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323.

Jones, E., Oliphant, T., Peterson, P., and Others (2001). SciPy: Open source scientific tools for Python.

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

Lichman, M. (2013). {UCI} Machine Learning Repository.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, (2):39–55.

Lourenço, A., Fred, A. L. N., and Jain, A. K. (2010). On the scalability of evidence accumulation clustering. *Proceedings - International Conference on Pattern Recognition*, 0:782–785.

Meila, M. (2003). 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, page 173.

Sibson, R. (1973). SLINK: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34.

Sirotkovi, J., Dujmi, H., and Papi, V. (2012). K-Means Image Segmentation on Massively Parallel GPU Architecture. pages 489–494.

Strehl, A. and Ghosh, J. (2002). Cluster Ensembles – A Knowledge Reuse Framework for. *Journal of Machine Learning Research*, 3:583–617.

The HDF Group. Hierarchical Data Format, version 5.

Zechner, M. and Granitzer, M. (2009a). Accelerating k-means on the graphics processor via CUDA. *Proceedings of the 1st International Conference on Intensive Applications and Services, INTENSIVE 2009*, pages 7–15.

Zechner, M. and Granitzer, M. (2009b). K-Means on the Graphics Processor: Design And Experimental Analysis. *International Journal on Advances in System and Measurements*, 2(2):224–235.