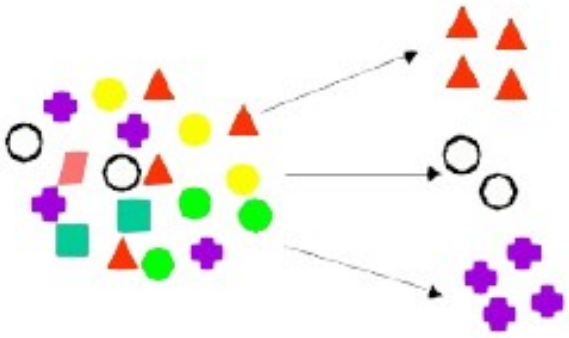


Efficient Evidence Accumulation Clustering for large datasets/big data

Diogo Alexandre Oliveira Silva
Alferes Aluno Engenheiro Electrotécnico 136787-A

Thesis to obtain the Master of Science Degree in

Aeronautical Military Sciences in the speciality of Electrical Engineering

Examination Committee

Chairperson:	Professor Full Name
Supervisor:	Dra. Ana Luísa Nobre Fred
Co-supervisor:	Dra. Helena Aidos Lopes
Member of the Committee:	Professor Full Name 3

Month 2015

Dedicated to someone special...

Acknowledgments

A few words about the university, financial support, research advisor, dissertation readers, faculty or other professors, lab mates, other friends and family...

Resumo

Inserir o resumo em Português aqui com o máximo de 250 palavras e acompanhado de 4 a 6 palavras-chave...

Palavras-chave: palavra-chave1, palavra-chave2,...

Abstract

Insert your abstract here with a maximum of 250 words, followed by 4 to 6 keywords...

Keywords: keyword1, keyword2,...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Challenges and Motivation

Advances in technology allow for the collection and storage of unprecedented amount and variety of data, a concept commonly designated by *Big Data*. Most of this data is stored electronically and there is an interest in automated analysis for generation of knowledge and new insights. The applications of such analysis are abundant and across many fields, ranging from recommender systems and customer segmentation in business, to predicting when a jet engine is likely to fail using sensor data, or even the study of gene expression in biomedics, to namy a few.

A growing body of statistical 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. Since, often, the structure of the data is unknown, clustering techniques become particularly interesting for transforming this data into knowledge and discovering its underlying structure and patterns. Clustering is a hard problem and a vast body of work on these algorithms exist. Yet, typically, no single algorithm is able to respond to the specificities of all data. Different methods are suited to datasets of different characteristics and, often, the challenge of the researcher is to find the right algorithm for the task.

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, e.g. algorithms that do not take any parameters for performing an analysis. One such algorithm is Evidence Accumulation Clustering (EAC), belonging to the wider class of ensemble methods. EAC is a state-of-the art clustering algorithm that addresses the robustness challenge. However, the current reality of capturing massive amounts of data rises new challenges. Two important challenges are efficiency and scalability, which translate on how fast the algorithms are and how well they scale when the input data multiplies in size, dimensionality and variety. The algorithms themselves are no longer the only focus of research. Much effort is being put into the scalability and performance of algorithms, which usually translates in addressing their computational complexity with parallelized computation and distributed memory being

some of the proposed solutions. Cluster analysis with EAC should be fast and able to scale to larger datasets as well as robust, so as to address the reality of big data.

This dissertation is concerned with pushing the current limits of the EAC to large datasets by addressing the problems of scalability and efficiency without compromising robustness, using technology available in a desktop workstation. Processing of huge amounts of data has been out of the range of capability of the traditional desktop workstation. This sprouted the rise of new uses of existing computing architectures (e.g. Graphic Processing Units) and development of new programming models (e.g. Hadoop, shared and distributed memory). The problem at hand is, then, to optimize the algorithm regarding both speed and memory usage. This, of course, comes with challenges. How can one keep the original accuracy while significantly increase efficiency? Is there an exploitable trade-off between the three main characteristics: speed, memory and accuracy? These are guiding questions that this dissertation addresses.

1.2 Goals

This dissertation aims to research and extend the state of the art of ensemble clustering, in what concerns the EAC method and its application to large datasets, while also assessing algorithmic solutions and parallelization techniques. The goal is to understand EAC's suitability for large datasets and find ways to respond to the stated challenges, in terms of speed and memory. The main objectives for this work are:

- Study the integration of quantum inspired methods in EAC.
- Study the integration of the General Purpose computing in a Graphics Processing Unit (GPGPU) paradigm in EAC.
- Devise strategies to reduce computation and memory complexities of EAC.
- Application of Evidence Accumulation Clustering to Big Data.
- Validation of Big Data EAC on real data.

1.3 Contributions

The main contributions are the adaptation of the three distinct stages of the EAC framework to larger datasets. In particular, an efficient parallel version for Graphics Processing Units (GPU) of the K-Means clustering algorithm is implemented for the first stage of EAC. Still in this stage, two clustering algorithms in the young field of Quantum Clustering were reviewed, tested and evaluated having EAC in mind. Different methods for the second stage were tested, using complete matrices and sparse matrices.Worthy of mention is a novel and specialized method for building a sparse matrix in the second stage. A GPU parallel version of a MST (Minimum Spanning Tree) solver algorithm was reviewed and tested for the last stage, a co-product of which was an algorithm to find the connected components of a MST. A hard

disk solution was implemented for dealing with large datasets whose space complexity in the final stage exceeded the available memory.

1.4 Outline

Chapter **??** provides an introduction to clustering nomenclature and concepts, as well as some ”traditional” clustering algorithms. Chapter **??** starts by reviewing the Evidence Accumulation Clustering algorithm in detail. It goes on to review possible approaches to the problem of scaling EAC. Based on an algorithmic approach, a review of the young field of quantum clustering is presented, with a more in-depth emphasis on two algorithms. With a parallelization approach in mind, a programming model for the GPU (CUDA) is reviewed, followed by some parallelized versions of relevant algorithms to the problem of this dissertation. The following chapter, **??**, presents the approach that was actually taken to scale EAC. It presents the steps taken on each part of the algorithm, the underlying difficulties and what was done to address them. It also includes the reference of approaches that were developed but were not deemed suited to integrate the EAC toolchain. In the results chapter, **??**, the results of the different approaches of optimizing the EAC method are presented. Chapter **??** presents an interpretation and critical discussion of those results. Finally, chapter **??** concludes the dissertation. It also offers recommendations for future work.

Chapter 2

Clustering: basic concepts, definitions and algorithms

Hundreds of methods for data analysis exist. Many of these methods fall into the realm of machine learning, which is usually divided into 2 major groups: *supervised* and *unsupervised* learning. Supervised learning deals with labeled data, i.e. data for which ground truth is known, and tries to solve the problem of classification. Examples of supervised learning algorithms are Neural Networks, Decision Trees, Linear Regression and Support Vector Machines. Unsupervised learning deals with unlabeled data for which no extra information is known. An example of algorithms within this paradigm is clustering algorithms, which are the focus of this chapter.

This chapter will serve as an introduction to clustering. It starts by defining the problem of clustering in section **??**, goes on to provide useful definitions and notation in section **??** and briefly addresses different properties of clustering algorithms in section **??**. Two very well known algorithms are presented: K-Means in section **??** and Single-Link in section **??**. Evidence Accumulation Clustering is a state of the art ensemble clustering algorithm and the focus of this dissertation. Section **??** will explain briefly the concept of ensemble clustering followed by an overview and application examples of the EAC algorithm in section **??**.

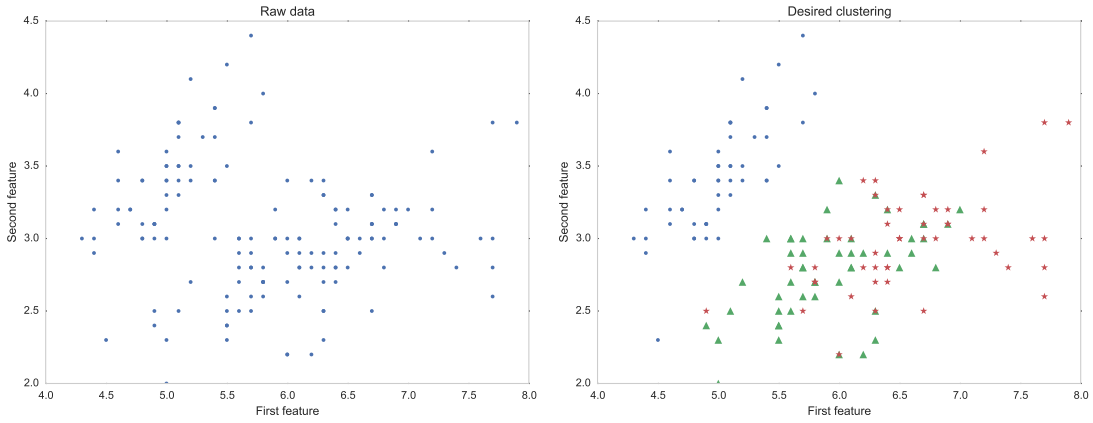
2.1 The problem of clustering

Cluster analysis methods are unsupervised and the backbone of the present work. The goal of data clustering, as defined by [**?**], is the discovery of the *natural grouping(s)* of a set of patterns, points or objects. In other words, the goal of data clustering is to discover structure on data. The methodology used is to group patterns (usually represented as a vector of measurements or a point in space [**?**]) 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. Clustering is a strictly data-driven method, in contrast with classification techniques which have a training set with the desired labels for a limited collection of patterns. Because there is very little information, as few assumptions as possible should be made

about the structure of the data (e.g. number of clusters). And, because clustering typically makes as few assumptions on the data as possible, it is appropriate to use it on exploratory structural analysis of the data. The process of clustering data has three main stages [**?**]:

- **Pattern representation** refers to the choice of representation of the input data in terms of size, scale and type of features. The input patterns may be fed directly to the algorithms or undergo *feature selection* and/or *feature extraction*. The former is simply the selection of which features of the originally available should be used. The latter deals with the transformation of the original features such that the resulting features will produce more accurate and insightful clusterings, e.g. Principal Component Analysis.
- **Pattern similarity** refers to the definition of a measure for computing the similarity between two patterns.
- **Grouping** refers to the algorithm that will perform the actual clustering on the dataset with the defined pattern representation, using the appropriate similarity measure.

As an example, Figure **??** shows the plot of the Iris data set [**?** **?**], a small well-known Machine Learning data set. This data set has 4 features, of which only 2 are represented, and 3 classes, of which 2 are overlapping. A class is overlapping another if they share part of the feature space, i.e. there is a zone in the feature space whose patterns might belong to either class. Figure **??** presents the desired clustering for this data set.



(a) Input data, unlabeled. (b) Desired labels.

Figure 2.1: First and second features of the Iris dataset. Fig. **??** shows the raw input data, i.e. how the algorithms ”see” the data. Fig. **??** shows the desired labels for each point, where each color is coded to a class.

2.2 Definitions and Notation

This section will introduce relevant definitions and notation within the clustering context that will be used throughout the rest of this document and were largely adopted from [**?**].

A *pattern* $\mathbf{x}$ is a single data item and, without loss of generality, can be represented as a vector of d *features* x_i that characterize that data item, $\mathbf{x} = (x_1, \ldots, x_d)$, where d is referred to as the dimensionality of the pattern. A *pattern set* (or data set) $\mathcal{X}$ is then the collection of all n patterns $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$. The number of features is usually the same for all patterns in a given pattern set.

In cluster analysis, the desired clustering, typically, is one that reflects the natural structure of the data, i.e. the original ground truth labeling. In other words, one wants to group the patterns that came from the same state of nature when they were generated, the same *class*. A class, then, can be viewed as a source of patterns and the effort of the clustering algorithm is to group patterns from the same source. Throughout this work, these classes will also be referred to as the "natural" or "true" clusterings. *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 $\mathbf{x}_i$. Closely related to the whole set of labels is the concept of a *partition*, which completely describes a clustering. A partition P is a collection of k *clusters*. A cluster C is a subset of nc patterns $\mathbf{x}_i$ taken from the pattern set, where the patterns belonging to one subset do not belong to any other in the same partition. A clustering *ensemble* $\mathbb{P}$ is a set N partitions P^j from 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. The relationship between the above concepts is condensed in the following expressions:

$$\begin{aligned} ensemble \quad & \mathbb{P} = \left\{P^1, P^2, \ldots P^N\right\} \\ partition \quad & P^j = \left\{C_1^j, C_2^j, \ldots C_{k_j}^j\right\} \\ cluster \quad & C_i^j = \left\{x_1, x_2, \ldots x_{nc_i^j}\right\} \end{aligned}$$

Typically, a clustering algorithm will use a *proximity* measure for determining how alike are two patterns. A proximity measure can either be a *similarity* or a *dissimilarity* measure. One can easily be converted to the other and the main difference is that the former increases in value as patterns are more alike, while the latter decreases in value. 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:

$$\begin{aligned} identity \quad & d(\mathbf{x}_i, \mathbf{x}_i) = 0 \\ symmetry \quad & d(\mathbf{x}_i, \mathbf{x}_j) = d(\mathbf{x}_j, \mathbf{x}_i), i \neq j \\ triangle\ inequality \quad & d(\mathbf{x}_i, \mathbf{x}_j) + d(\mathbf{x}_j, \mathbf{x}_z) \geq d(\mathbf{x}_x, \mathbf{x}_z) \end{aligned}$$

where $\mathbf{x}_i$, $\mathbf{x}_j$ and $\mathbf{x}_z$ are 3 unique patterns belonging to the pattern set $\mathcal{X}$. Examples of proximity measures include the Euclidean distance, the Pearson's correlation coefficien and Mutual Shared Neighbors [**?**]. It should be noted that different proximity measures may be more appropriate in different contexts, such as document, biological or time-series clustering. Furthermore, data can come in different

types such as numerical (discrete or continuous) or categorical (binary or multinomial) attributes. The researcher should take these factors into account as different proximity measures are more appropriate for some type or even heterogeneous type data.

An introduction of clustering would be incomplete without a discussion on how good is a partition after clustering. Several *validation measures* exist and they can placed in two main categories [**?**]. *External* measures use *a priori* information about the data to evaluate the clustering against some external structure. An application of an external measure could be to test how accurate a clustering algorithm is for a particular dataset by matching the output partition against the ground truth. Examples of such measures include the *Consistency Index* [**?**] and a measure of accuracy based on the problem of minimum weighted bipartite graphs matching [**?**], which throughout the remainder of the present work will be refered to as *H-index*. *Internal* measures, on the other hand, determine the quality of the clustering without the use of external information about the data. The Davies-Bouldin index [**?**] is such a measure.

2.3 Characteristics of clustering techniques

Clustering algorithms may be categorized and described according to different properties. For the sake of completeness, a brief discussion of some of their properties will be layed out in this section.

It is common to organize cluster algorithms into two distinct types: *partitional* and *hierarchical*. A partitional algorithm, such as K-Means, is a hard clustering algorithm that will output a partiton where each pattern belongs exclusively to one cluster. A hierarchical algorithm produces a tree-based data structure called *dendrogram*. A dendrogram contains different partitions at different levels of the tree which means that the user can easily change the desired number of clusters by simply traversing the different levels. This is an advantage over a partitional algorithm since a user can analyze different partitions with different numbers of clusters without having to rerun the algorithm. Hierarchical algorithms can be further split into two approaches: bottom-up (or *agglomerative*) and top-down (or *divisive*). The former starts with all patterns as distinct clusters and will group them together according to some dissimilarity measure, building the dendrogram from the ground up; examples of algorithms that take this approach are Single-Link and Average-Link. The latter will start will all patterns in the same cluster and continuosly split it until all patterns are separated, building the dendrogram from the top level to the bottom; this approach is taken by the Principal Directon Divisive Partitioning[**?**] and Bisecting K-Means [**?**] algorithms.

Another characteristic relates to how algorithms use the features for computing similarities. If all features are used simultaniously the algorithm is called *polithetic*, e.g. K-Means. Otherwise, if the features are used sequentially, it is called *monothetic*, e.g. [**?**].

Contrasting *hard* clustering algorithm are the *fuzzy* algorithms. A fuzzy algorithm will attribute to each pattern a degree of membership to each cluster. A partition can still be extracted from this output by choosing, for each pattern, the cluster with higher degree of membership. An example of a fuzzy algorithm is the Fuzzy C-Means [**?**].

Another characteristic is an algorithm's stochasticity. A *stochastic* algorithms uses a probabilistic process at some point in the algorithms, possibly yielding different results in each run, e.g. K-Means can use a random initialization. As an example, the K-Means algorithm typically picks the initialization centroids randomly. A *deterministic* algorithm, on the other hand, will always produce the same result for a given input, e.g. Single-Link.

Finally, the last characteristic discussed is how an algorithm processes the input data. An algorithm is said to be *incremental* if it processes the input incrementally, i.e. taking part of the data, processing it and then doing the same for the remaining parts, e.g. PEGASUS [**?**]. A *non-incremental* algorithm, on the other hand, will process the whole input in each run, e.g. K-Means. This discussion is specially relevant when considering large datasets that may not fit in memory or whose processing would take too long for a single run and is therefore done in parallel.

2.4 K-Means

One of the most famous non-optimal solutions for the problem of partitional clustering is the K-Means algorithm [**?**]. The K-Means algorithm uses K *centroid* representatives, c_k, for K clusters. Patterns are assigned to a cluster such that the squared error (or, more accurately, squared dissimilarity measure) between the cluster representatives and the patterns is minimized. In essence, K-Means is a solution (although not necessarily an optimal one) to an optimization problem having the Sum of Squared Errors as its objective function, which is known to be a computationally NP hard problem [**?**]. It can be mathematically demonstrated that the optimal representatives for the clusters are the means of the patterns of each cluster [**?**]. K-Means, then, minimizes the following expression, where the proximity measure used is the Euclidean distance:

$$\sum_{k=1}^{K} \sum_{\mathbf{x}_i \in C_k} \|\mathbf{x}_i - c_k\|^2 \tag{2.1}$$

K-Means needs 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, hence the name of the algorithm. 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, e.g. K-Means++ [**?**]. There are also methods to automatically choose the number of clusters [**?**].

The proximity measure used is typically the Euclidean distance. This tends to produce hyperspherical clusters [**?**]. Still, according to [**?**], other measures have been used such as the L1 norm, Mahalanobis distance, as well as the cosine similarity [**?**]. The choice of similarity measure must be made carefully

as it may not guarantee that the algorithm will converge.

A detail of implementation is what to do with clusters that have no patterns assigned to them. One approach to this situation is to drop the empty clusters in further iterations. However, allowing the existence of empty clusters or dropping empty clusters is undesirable since the number of clusters is an input parameter and it is expected that the output contains the specified number of clusters. Other approaches exist dealing with this problem, such as equaling the centroid of an empty cluster to the pattern furthest away from its assigned centroid or reusing the old centroids as in [**?**].

K-Means is a simple algorithm with reduced complexity $O(ntk)$, where n is the number of patterns in the pattern set, k is the number of clusters and t is the number of iterations that it executes. Accordingly, K-Means is often used as foundational step of more complex and robust algorithms, such as the EAC algorithm.

As an example, the evolution and output of the K-means algorithm to the data presented in Fig. **??** is represented in Fig. **??**. The algorithm was executed with 3 random centroids.

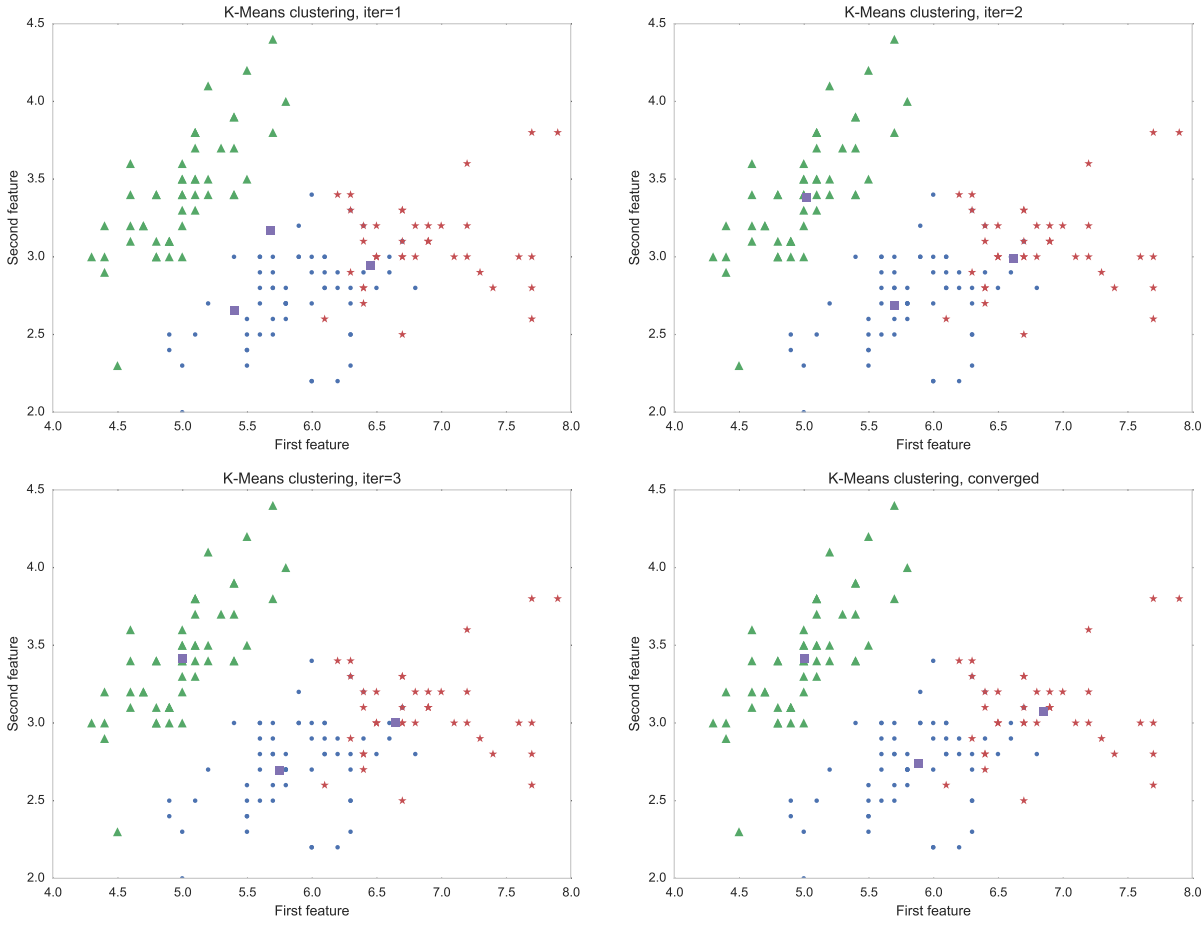


Figure 2.2: The output labels of the K-Means algorithm with the number of clusters (input parameter) set to 3. The different plots show the centroids (squares) evolution on each iteration. Between iteration 3 and the converged state 2 more iterations were executed.

Even with the correct number of clusters, the clustering results do not match 100% the natural clusters. The accuracy relative to the natural clusters of Fig. **??** is 88% as measured by the Consistency Index (CI) [**?**]. In this example, the problem is the two overlapping clusters. It is hard for an algorithm to

discriminate between two clusters when they have similar patterns. When no prior information about the dataset is given, the number of clusters can be hard to discover. This is why, when available, a domain expert may provide valuable insight on tuning the initialization parameters.

2.5 Single-Link

Single-Link [**?**] is one of the most popular hierarchical agglomerative clustering (HAC) algorithms. HAC algorithms operate over a pair-wise dissimilarity matrix and outputs a dendrogram (e.g. Fig **??**). The main steps of an agglomerative hierarchical clustering algorithm are the following [**?**]:

1. Create a pair-wise dissimilarity matrix of all patterns, where each pattern is a distinct cluster singleton;

2. Find the closest clusters, merge them and update the matrix to reflect this change. The rows and columns of the two merged clusters are deleted and a new row and column are created to store the new cluster.

3. Stop if all patterns belong to a single cluster, otherwise continue to step 2.

The algorithm stops when $n-1$ merges have been performed, which is when all patterns have been connected in the same cluster. Just like in the K-Means algorithm, different similarity measures can be used for the distances.

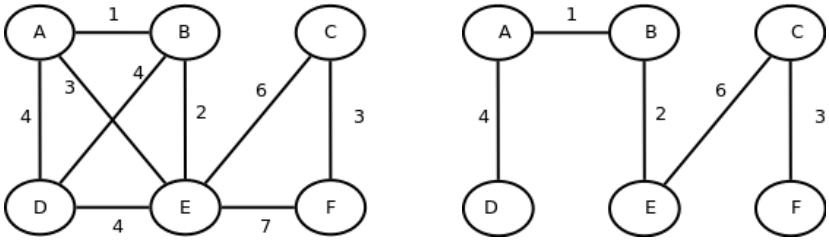
The proximity between clusters in the second step is distinguishes between the different HAC linkage algorithms, such as Single-Link , Average-Link, Complete-Link, among others. In Single-Link (SL), the proximity between any two clusters is the the dissimilarity between their closest patterns. On the other hand, in Complete-Link, it is the proximity between their most distant patterns and, in Average-Link, is the proximity between the average point of each cluster. In SL, because the algorithm connects first clusters that are more similar, it naturally gives more importance to regions of higher density [**?**].

The total time complexity of a naive implementation is $O(n^3)$ since it performs a $O(n^2)$ search in step two and it does it $n-1$ times. Over time, more efficient implementations have been proposed, such as SLINK [**?**]. SLINK needs no working copy of $O(n^2)$ the pair-wise similarity matrix (if the original can be modified), has a working memory of $O(n^2)$ and time complexity of $O(n^2)$. This increase in performance comes from the observation that the $O(n^2)$ search can be transformed in a $O(n)$ search at the expense of keeping two arrays of length n that will store the most similar cluster for each pattern and the corresponding similarity measure. This way, to find the two closest clusters, the algorithm will not search the entire similarity matrix, but only the new similarity array since this array keeps the closest cluster of each cluster. Naturally these arrays must be updated upon a cluster merge.

An interesting property of the SL algorithm is its equivalence with a Minimum Spanning Tree (MST), an observation first made by [**?**]. In graph theory, a MST is a tree that connects all vertices together while minimizing the sum of all the distance between them. An example of a graph and its corresponding MST can be seen in Fig. **??**. 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 Single-Link dendrogram. To walk down through the levels of the dendrogram from the MST, one cuts the least similar edges. Furthermore, this approach can be used to apply Single-Link clustering to graphs-encoded problems in a straight-forward way. Furthermore, the performance properties of this method are roughly the same as SLINK [**?**].

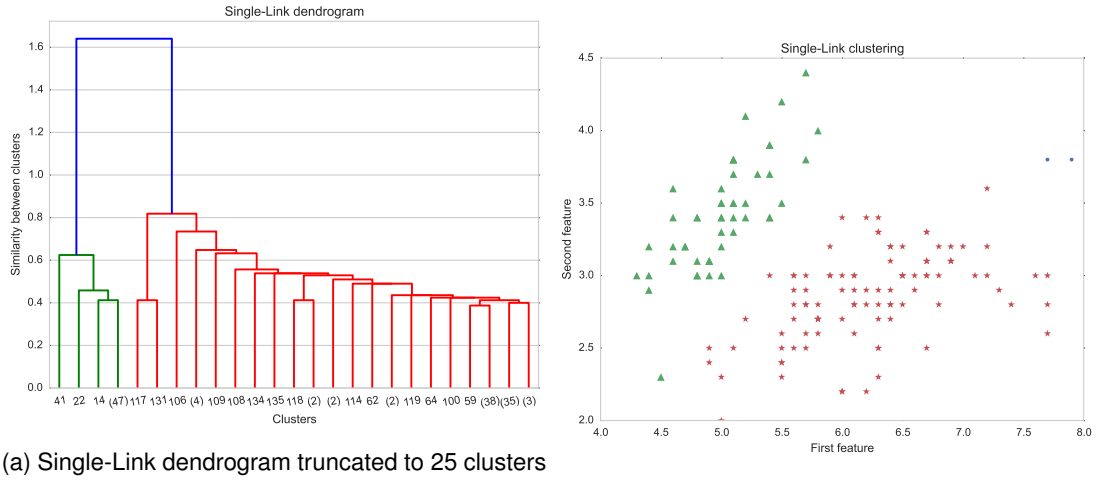
The true advantage of using an MST based approach comes when the number of edges (similarities) m of the MST is less than $\frac{n(n-1)}{2}$, where n is the number of nodes (patterns) [**?**]. This is because SLINK works over a inter pattern similarity matrix, meaning that the similarity between every pair of patterns must be explicitly represented. The minimum number of similarities is $\frac{n(n-1)}{2}$, which is equivalent to the upper or lower half triangular matrices of the similarity matrix. The MST, on the other hand, works over a graph that may or may not have edges between every pair of nodes. Fast MST algorithms have a time complexity of $O(mlogn)$, which is an improvement over $O(n^2)$ when $m << \frac{n(n-1)}{2}$.



(a) Example of a graph. (b) MST of the graph to the left.

Figure 2.3: The above figures show an example of a graph (left) and its corresponding Minimum Spanning Tree (right). The circles are vertices and the edges are the lines linking the vertices.

An example of a Single-Link dendrogram and resulting cluster can be observed in Fig. **??**. The dendrogram in Fig. **??** has been truncated to 25 clusters in the bottom level for the sake of readability. The clustering presented on Fig. **??** is the result of cutting the dendrogram such that only 3 clusters exist (the number of classes). The accuracy, as measured by the CI, is of 58%.



(a) Single-Link dendrogram truncated to 25 clusters in the bottom level.

(b) Clustering with 3 clusters.

Figure 2.4: The above plots show the dendrogram and a possible clustering taken from a Single-Link run over the Iris data set. Fig. **??** was obtained by performing a cut on a level that would yield a partition of 3 clusters.

2.6 Ensemble Clustering

The underlying idea behind ensemble clustering is to take a collection of partitions, a *clustering ensemble*, and combine it into a single partition. There are several motivations for ensemble clustering. Data from real world problems appear in different configurations regarding shape, cardinality, dimensionality, sparsity, etc. Different clustering algorithms are appropriate for different data configurations, e.g. K-Means tends to group patterns in hyperspheres [**?**] so it is more appropriate for data whose structure is formed by hypershere like clusters. If the true structure of the data at hand is heterogeneous in its configuration, a single clustering algorithm might perform well for some part of the data while other performs better for some other part. Since different algorithms can be used to produce the partitions in the ensemble, one can use a mix of algorithms to address different properties of the data such that the combination is more **robust** to noise and outliers [**?**] and the final clustering has a **better quality** [**?**]. Ensemble clustering can also be particularly useful in situations where one does not have direct access to all the features of a given data set but can have access to partitions from different subsets and later combining with an ensemble algorithm. Furthermore, the generation of the clustering ensemble can be **parallelized and distributed** since each partition is independent from every other partition.

A clustering ensemble, according to [**?**], can be produced from (1) different data representations, e.g. choice of preprocessing, feature selection and extraction, sampling; or (2) different partitions of the data, e.g. output of different algorithms, varying the initialization parameters on the same algorithm.

Ensemble clustering algorithms can take three main distinct approaches [**?**]: based on pair-wise similarities, probabilistic or direct. EAC [**?**] and CSPA [**?**] are examples of pair-wise similarity based approach, where the algorithms use a co-associations matrix. The MMCE [**?**] and BCE [**?**] are examples of a probabilistic approach. This approach will be further clarified when the EAC algorithm is explained. HGPA [**?**], MCLA [**?**] and bagging [**?**] are examples of a direct approach to combining the ensemble clusterings, where the algorithms work directly with the labels without creating a co-association matrix. A detailed and thorough review of the similarity measures that can be used on with clustering ensembles and the state of the art algorithms can be consulted in [**?**].

2.7 Evidence Accumulation Clustering

2.7.1 Overview

The goal of EAC is to find an optimal partition P^* containing k^* clusters, from the clustering ensemble $\mathbb{P}$. The optimal partition should have the following properties [**?**]:

- **Consistency** with the clustering ensemble;
- **Robustness** to small variations in the ensemble; and,
- **Goodness** of fit with ground truth information, when available.

Ground truth is the true labels of each sample of the dataset, when such exists, and is used for validation purposes. Since EAC is an unsupervised method, this typically will not be available. EAC makes no assumption on the number of clusters in each data partition. 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. Within the context of EAC, it is of interest to have variety in the ensemble with the intention to better capture the underlying structure of the data. One such parameter to measure that variety is the number of clusters in the partitions of the ensemble. Typically, the number of clusters in each partition is drawn from an interval $[K_{min}, K_{max}]$ with uniform probability. This influences other properties of other parts of the algorithm such as the sparsity of the co-association matrix as will become clearer in future chapters. Reviewing the literature [**? ? ? ?**], it is clear the ensemble is usually produced by random initialization of K-Means (specifying only the number of centroids within the above interval). Still, other clustering algorithms have been used for the production of the ensemble [**?**] such as Single-Link, Average-Link and CLARANS.

The ensemble of partitions is combined in the second step, where a non-linear transformation turns the ensemble into a co-association matrix [**?**], i.e. a matrix $\mathcal{C}$ where each of its elements n_{ij} is the association value between the pattern pair (i, j). The association between any pair of patterns is given by the number of times those two patterns appear clustered together in any cluster of any partition of the ensemble, i.e. 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 [**?**], revealing the underlying structure of the data. In other words, a high association n_{ij} means it is more likely that patterns i and j belong to the same class. The construction of the co-association matrix is at the very core of this method.

The co-association matrix itself is not the output of EAC. Instead, it is used as input to other methods to obtain the final partition. The co-association between any two patterns can be interpreted as a similarity measure. Thus, since this matrix is a similarity matrix it's appropriate to use algorithms that take this type of matrices as input, e.g. K-Medoids or hierarchical algorithms such as Single-Link or Average-Link, to name a few. Typically, algorithms use a distance as the dissimilarity, which means that they minimize the distance to obtain the highest similarity between objects. However, a low value on the co-association matrix translates in a low similarity between a pair of objects, which means that the co-association matrix requires prior transformation for accurate clustering results, e.g. replace every similarity value n_{ij} between every pair of object (i, j) by $max\{\mathcal{C}\} - n_{ij}$.

Although any algorithm can be used, the final clustering is usually done using SL or AL. Each of this algorithms will take as input the transformed co-association matrix as the dissimilarity matrix. Furthermore, not knowing the "natural" number of clusters one can use the lifetime criteria, i.e. the number of

clusters k should be such that it maximizes the cost of cutting the dendrogram from $k-1$ clusters to k. Further details on the lifetime strategy for picking the number of clusters falls outside the scope of this work and are presented in [**?**].

Related work to EAC has been developed. The Weighted EAC (WEAC) algorithm [**?**] and a study on the sparsity of the co-association matrix [**?**] should be mentioned. The latter is discussed in more depth in chapter **??**. The former introduces the novelty of having weights associated to each partition such that good quality partitions are more relevant than their counterparts. These weights are based on internal validity measures. Weighing the partitions in terms of quality has shown to improve the original algorithm, accuracy wise.

2.7.2 Examples of applications

EAC has been used with success in several areas. Some of its applications are:

- in the field of bioinformatics it was used for the automatic identification of chronic lymphocyt leukemia [**?**];
- also in bioinformatics it was used for the unsupervised analysis of ECG-based biometric database to highlight natural groups and gain further insight [**?**];
- in computer vision it was used as a solution to the problem of clustering of contour images (from hardware tools) [**?**].

Chapter 3

State of the art

Scalability of EAC to large data sets is the concern of this work and, because of that, this chapter starts by reviewing what has been done in terms of scaling EAC in section **??**. EAC is a method of three parts and this dissertation is concerned with the scalability of the whole algorithm which means that each step must be optimized. Scaling an algorithm means one has to take into account both speed of execution and memory requirements. Increasing speed can be attained with either faster algorithms and/or faster computation of existing algorithms. This chapter reflects research done within both approaches.

Although research on the application of EAC to large data sets has not been pursued before, cluster analysis of large data sets has. Since EAC uses traditional clustering algorithms (e.g. K-Means, Single-Link) in its approach, it is useful to understand how scalable the individual algorithms are as they will have a big impact in the scalability of EAC. Furthermore, valuable insights may be taken from the techniques used in the scalability of other algorithms. To this end, section **??** presents a brief review on cluster analysis of large data sets, with a focus on parallelization with GPUs. Furthermore, it offers a more detailed description of a GPU parallel version of K-Means and an approach for parallelizing Single-Link with the GPU.

An alternate approach on clustering for scaling with faster algorithms, the still young field of quantum clustering, was reviewed in section **??**. This line of research was taken mostly with the first step of EAC in mind.

3.1 Scalability of EAC

The quadratic space and time complexity of processing the $n \times n$ co-association matrix is an obstacle to an efficient scaling of EAC. Two approaches have been proposed to address this obstacle: one dealing with reducing the co-association matrix by considering only the distances of patterns to their p neighbors and the other by using a sparse co-association matrix and maximizing its sparsity.

3.1.1 p neighbors approach

The first approach, [**?**], proposes an alternative $n \times p$ co-association matrix, where only the p nearest neighbors of each pattern are considered in the evidence combination step. This comes at the cost of having to keep track of the neighbors of each pattern in a separate data structure of $O(np)$ memory complexity and also of pre-computing the p neighbors, which has a time complexity of $O(n^2)$ to compute the similarity measure from each pattern to every other pattern. The quadratic space complexity of the co-association matrix is then transformed to $O(2np)$: $O(np)$ for the actual co-association matrix and $O(np)$ for keeping track of the neighbors. Since usually one has $p < \frac{n}{2}$ (value for which both this approach and the original $n \times p$ matrix would take the same space), the cost of storing the extra data structure is lower than that of storing an $n \times n$ matrix, e.g. for a dataset with 10^6 patterns and $p = \sqrt{10^6}$ (a value much higher than the 20 neighbors used in [**?**]), the total memory required for the co-association matrix would decrease from $3725.29GB$ to $7.45GB$ (0.18% of the memory occupied by the complete matrix).

3.1.2 Increased sparsity approach

The second approach, presented in [**?**], exploits the sparse nature of the co-association matrix. The co-association matrix is symmetric and with a varying degree of sparsity. The former property translates in the ability of storing only the upper triangular of the matrix without any loss on the quality of the results. The latter property is further studied with regards to its relationship with the minimum K_{min} and maximum K_{max} number of clusters in the partitions of the input ensemble. The core of this approach is to only store the non-zero values of the upper triangular of the co-association matrix. The authors study 3 models for the choice of these parameters:

- choice of K_{min} based on the minimum number of gaussians in a gaussian mixture decomposition of the data;
- based on the square root of the number of patterns ($\{K_{min}, K_{max}\} = \{\frac{\sqrt{n}}{2}, \sqrt{n}\}$);
- or based on a linear transformation of the number of patterns ($\{K_{min}, K_{max}\} = \{\frac{n}{A}, \frac{n}{B}\}, A < B$).

where A and B are two suitable constants chosen by the researcher. The study compared the impact of each model in the sparsity of the co-association matrix (and, thus, the space complexity) and in the relative accuracy of the final clusterings. Both theoretical predictions and results revealed that the linear model produces the highest sparsity in the co-association matrix, under a dataset consisting of a mixture of Gaussians. Furthermore, it is true for both linear and square root models that the sparsity increases as the number of samples increases.

For real data sets, the performance of the three models became increasingly similar with the increase of the cardinality of the problem. It was found that the chosen granularity of the input partitions (K_{min}) is the variable with most impact, affecting both accuracy and sparsity. The authors reported this technique to have linear space and time complexity on benchmark data.

The number of samples of the data sets analysed in [**?**] was under 10^4. Furthermore, it should be noted that the remarks concerning the sparsity of the co-association matrix in the aforementioned study refer to the number of non-zero elements in the matrix and does not take into account extra data structures that accompany real sparse matrices implementations.

3.2 Clustering with large datasets

When large datasets, and big data, is in discussion, two perspectives should be taken into account [**?**]. The first deals with the applications where data is too large to be stored efficiently. This is the problem that streaming algorithms such as LOCALSEARCH [**?**] try to solve by analyzing data as it is produced, close to real-time processing. The other perspective is data that is actually stored for later processing which is the perspective relevant to the present work and will be further discussed below.

The flow of clustering algorithms typically involves some initialization step (e.g. choosing the number of centroids in K-Means) followed by an iterative process until some stopping criteria is met, where each iteration updates the clustering of the data [**?**]. In light of this, to speed up and/or scale up an algorithm, three approaches are available: (1) reduce the number of iterations, (2) reduce the number of patterns and/or features to process or (3) parallelizing and distributing the computation. The solutions for each of this approaches are, respectively, one-pass algorithms (e.g. CLARANS [**?**], BIRCH [**?**], CURE [**?**]), randomized techniques that reduce the input space complexity (e.g. PCA, CX/CUR [**?**]) and parallel algorithms (parallel K-Means [**?**], parallel spectral clustering [**?**]).

Parallelization can be attained by adapting algorithms to multi core CPU, GPU, distributed over several machines (a *cluster*) or a combination of the former, e.g. parallel and distributed processing using GPU in a cluster of hybrid workstations. Each approach has its advantages and disadvantages. The CPU approach has access to a larger memory but the number of computation units is reduced when compared with the GPU or cluster approach. Furthermore, CPUs have advanced techniques such as branch prediction, multiple level caching and out of order execution - techniques for optimized sequential computation. GPU have hundreds or thousands of computing units but typically the available device memory is reduced which entails an increased overhead of memory transfer between host (workstation) and device (GPU) for computation of large datasets. In addition, it is harder to scale the above solutions for even bigger datasets. On the other hand, GPUs can be found on a large variety of computing platforms, from mobile devices to workstations and datacenters. A cluster offers a parallelized and distributed solution, which is easier to scale. According to [**?**], the two algorithmic approaches for cluster solutions are (1) memory-based, where the problem data fits in the main memory of the machines of the cluster and each machine loads part of the data; or (2) disk-based, comprising the widely used MapReduce framework capable of processing massive amounts of data in a distributed way. The main disadvantage is that there is a high communication and memory I/O cost to pay. Communication is usually done over the network with TCP/IP, which is several orders of magnitude slower than the direct access of the CPU or GPU to memory (host or device).

The present work is oriented towards GPU based parallelization, since GPUs are an easily accessible

commodity and the goals of the dissertation are oriented towards computation on a single machine. Taking that into consideration, this section starts by covering a review of the General Purpose computing on GPUs paradigm. It goes on to review a GPU parallel version of the K-Means algorithm and a GPU parallel approach for performing Single-Link clustering.

3.2.1 General Purpose computing on Graphical Processing Units

The original intended used for GPUs was graphics processing, hence its name. Recently, GPUs have been increasingly used for other purposes - a trend commonly known as General Purpose processing in Graphics Processing Units (GPGPU). GPU present a solution for ”extreme-scale, cost-effective, and power-efficient high performance computing” [**?**]. Furthermore, GPU are typically found in consumer desktops and laptops, effectively bringing this computation power to the masses.

GPUs were typically useful for users that required high performance graphics computation. Other applications were soon explored as users from different fields realized the high parallel computation power of these devices. However, the architecture of the GPUs themselves has been strictly oriented toward the graphics computing until the appearance of specialized GPU models designed for data computation (e.g. NVIDIA Tesla).

GPGPU application on several fields and algorithms has been reported with significant performance increase, e.g. application on the K-Means algorithm [**? ? ? ?**], hierarchical clustering [**? ?**], document clustering [**?**], image segmentation [**?**], integration in Hadoop clusters [**? ?**], among other applications.

Current GPUs pack hundreds of cores and have a better energy/area ratio than traditional infrastructure. GPU work under the SIMD framework, i.e. all the cores in the device execute the same code at the same time and only the data changes over time.

Programming GPUs

In the very beginning of GPGPU, programming was done directly through graphics APIs. Programming for GPUs was traditionally done within the paradigm of graphics processing, such as DirectX and OpenGL. If researchers and programmers wanted to tap into the computing power of a GPU they had to learn and use these APIs and frameworks, which is a challenging task since their general problems had to be modelled to the graphics-oriented primitives [**?**]. With the appearance of DirectX 9, shader programming languages of higher level became available (e.g. C for graphics, DirectX High Level Shader Language, OpenGL Shading Language), but they were still inherently graphics programming languages, where computation must be expressed in graphics terms.

More recent programming models, such as CUDA and OpenCL, removed a lot of that burden by exposing the power of GPUs in a way closer to traditional programming. Currently, the major programming models used for computation in GPU are OpenCL and CUDA. While the first is widely available in most devices, the latter is only available for NVIDIA devices.

As Google’s MapReduce computing model has increasingly become a standard for scalable and distributed computing over big data, attempts have been made to port the model to the GPU [**? ? ?**].

This translates in using the same programming model over a wide array of computing platforms.

OpenCL vs CUDA

Presently, the most mature programming models are CUDA and OpenCL. CUDA appeared first and is supported only by NVidia devices. It is also the most mature of the two and performs well since it was designed alongside with the hardware architecture of the supporting devices. OpenCL has the advantage of portability, but that comes with issues of performance portability. Both models are, in fact, very similar and literature suggests that porting the code from one to the other requires minimal changes [**?** **?**]. Literature also reports that CUDA performs better than OpenCL [**?**] for equivalent code.

Overview of CUDA

This section presents an overview of the CUDA programming model and its main concepts and characteristics. For a more thorough and extensive explanation of these topic, the CUDA C Programming Guides [**?**], the source of the present review, should be consulted. A GPU is comprised by one or several streaming processors (or multiprocessor). Each of this processors 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*. This hierarchy is represented in Figure **??**. The computation of one block is independent from other blocks and each block is scheduled to one multiprocessor, which means that more multiprocessors results in more blocks being processed at the same time, as represented in Figure **??**.

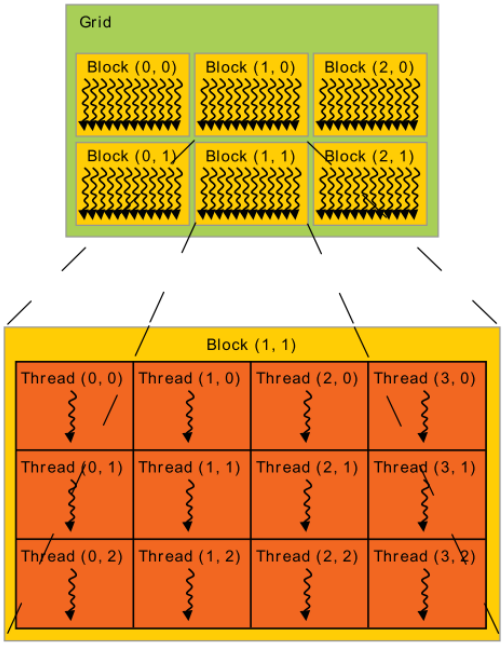


Figure 3.1: Thread hierarchy [**?**].

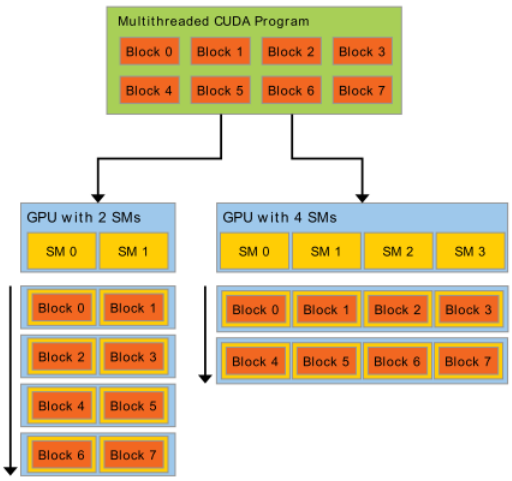


Figure 3.2: Distribution of thread blocks is automatically scaled with the increase of the number of multiprocessors [**?**].

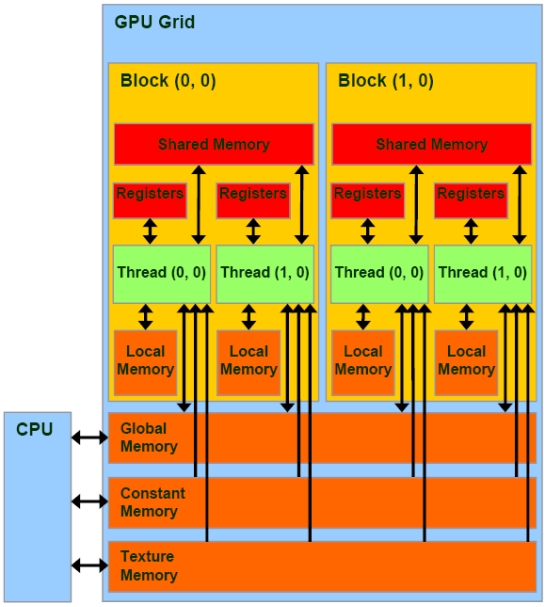


Figure 3.3: Memory model used by CUDA [**?**].

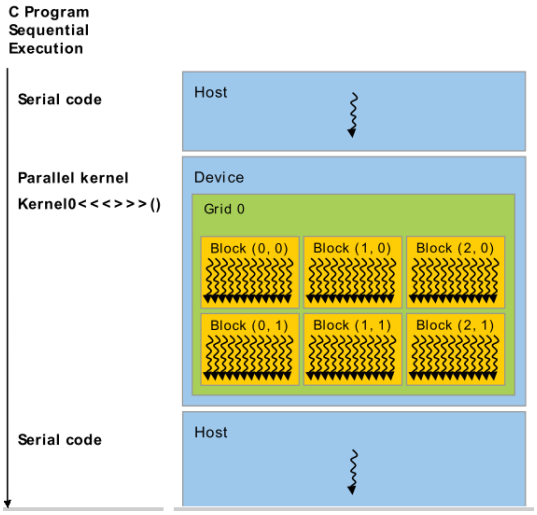


Figure 3.4: Sample execution flow of a CUDA application [**?**].

Block configuration can be multidimensional, up to and including 3 dimensions. Furthermore, there is a limit to the amount of threads in each dimension that varies with the version of CUDA being used, e.g. for GPUs with CUDA compute capability 2.x the number of threads is 1024 for the x- or y-dimensions, 64 for the z-dimension, an overall maximum number of threads is 1024 and a warp size of 32 threads. For the previous example, it is wise for the number of threads used in a block to be a multiple of 32 to maximize processor utilization, otherwise some blocks will have processors that will do no work.

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. Local memory 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 to, which is very fast. If the memory used exceeds the available register space the local memory used. This memory hierarchy is represented in Figure **??**.

The typical flow of a CUDA application (and, typically, any modern GPU application) is explained in this paragraph and can be observed in Figure **??**. First, the host CPU is responsible for several steps in the set-up phase. The host starts by transferring any necessary to the device memory (global, texture or constant). 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 if followed by the computation phase in the GPU. Finally, the host will transfer back the results from the device. It should be noted that the latest architectures support *Dynamic Parallelism*. This functionality allows the device to start other kernels without the intervention of the host CPU, which would alter the typical execution flow explained above if used. It can be particularly useful if several kernels have to be

executed in an application from the same input data but with dependencies. In such a scenario, a block of the second kernel could be executed as soon as all dependencies from the first kernel were met, effectively cutting overheads for kernel calling from the host CPU.

3.2.2 Parallel K-Means

K-Means is an obvious candidate to generate the ensemble of the first step of EAC because it uses different initializations and parameters and due to its simplicity. Besides, K-Means is a very good candidate for parallelization. Still, other algorithms can be used to produce ensembles.

Several parallel implementations of this algorithm for the GPU exist [**? ? ? ? ?**] and all report significant speed-ups relative to their sequential counterparts in certain conditions, usually after the input data set goes above a certain cardinality, dimensionality or number of clusters threshold. The first step is inherently parallel as the computation of the label of the i-th pattern is not dependent on any other pattern, but only on the centroids. Two approaches to parallelize this step on the GPU are possible, a centroid-centric or a data-centric [**?**]. In the former each thread is responsible for a centroid and will compute the distance from its centroid to every pattern. These distances must be stored and, in the end, the patterns are assigned to the closest centroid. In the latter, each thread will compute the distance from one or more data points to every centroid and determines to which centroid they are closest. This strategy has the advantage of using less memory since it does not need to store all the pair-wise distances to perform the labeling - it only needs to store the best distance for each pattern. According to [**?**], the former approach is suitable for devices with a low number of processors so as to stream the data to each one, while the latter is better suited to devices with more processors.

The approach taken in [**?**] only parallelizes the labeling stage and takes a data-centric approach to the problem. Each thread computes the distance from a set of data points to every centroid and determines the labels. The remaining steps are performed by the host CPU. This study reported speed ups up to 14, for input datasets of $500\,000$ points. Furthermore, it should be noted that the speed up was measured against a sequential version with all C++ compiler optimizations turned on, including vector operations (which, by themselves, are a way of parallelizing computation). The parallelized algorithm's flow can be observed in Figure **??**.

It should be noted that the literature reports that the performance of K-Means using Dynamic Parallelism is slightly worse than its standard GPU counterpart [**?**].

3.2.3 Parallel Single-Link Clustering

Single-Link (SL) is an important step in the EAC chain. Given the new similarity metric (how many times a pair of patterns are clustered together in the ensemble), SL provides an intuitive way of obtaining the final partition: patterns that are clustered together often in the ensemble should remain clustered together in the final solution.

SL is not easily parallelized since a new cluster generated at each step may include the one generated in the previous iteration. The most parallelizable part is the computation of the pair-wise similarity

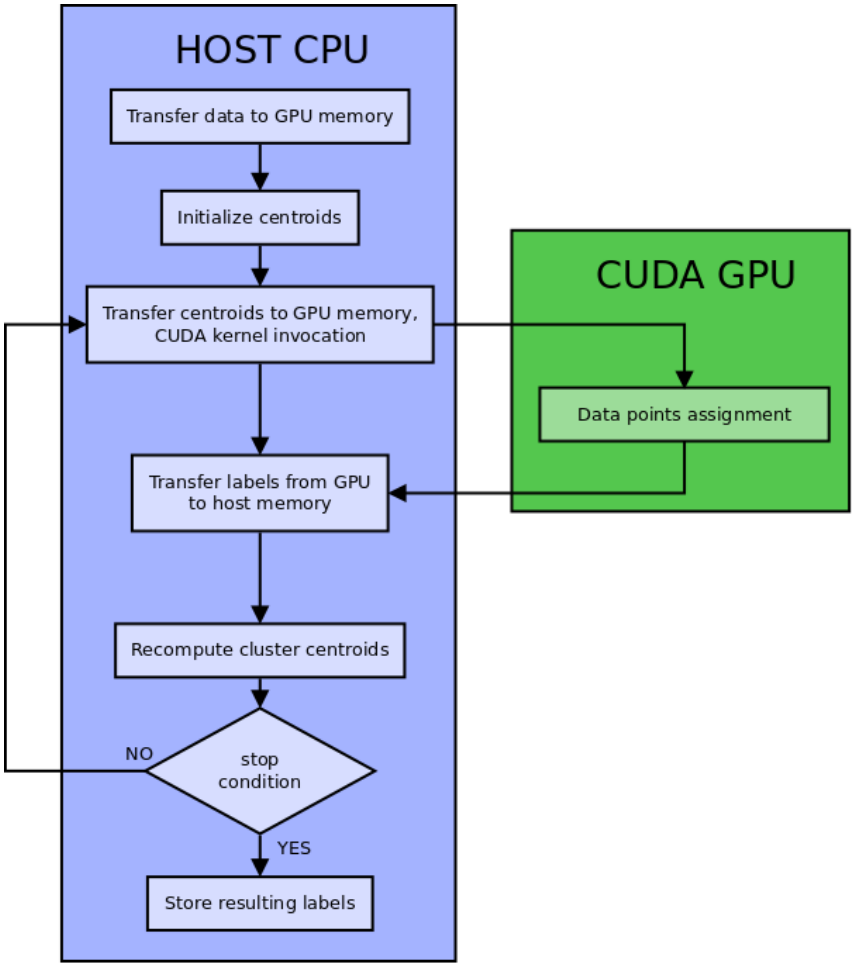


Figure 3.5: Flow execution of the GPU parallel K-Means algorithm.

matrix, which is only useful if the input is raw data instead of a similarity matrix as in the case of EAC. The relationship between SL and the Minimum Spanning Tree, explained in chapter **??**, is the key to parallelize it. If one takes this approach for solving the SL problem, it becomes easier to parallelize it since parallel MST algorithms are abundant in literature [**? ? ?**]. The same approach for extracting the final clustering in EAC was used in [**?**].

Algorithm for finding Minimum Spanning Trees

There are several algorithms for computing an MST. The most famous are Kruskal [**?**], Prim [**?**] and Borůvka [**?**]. Borůvka's algorithm is also known as Sollin's algorithm. The first two are mostly sequential, while the latter has the highest potential for parallelization, specially in the first iterations. As such, even though GPU parallel variants of Kruskal's [**?**] and Prim's [**?**] algorithms exist, the focus will be on Borůvka's.

Several parallel implementations of this algorithm for the GPU exist, e.g. [**?**], [**?**] and [**?**]. **?**] provides a more in-depth review over the current state of the art of MST solvers for the GPU and proposes an algorithm reported to be the fastest. This section will review the algorithm proposed in [**?**], referred to as *Sousa2015* from henceforth.

CSR format

Sousa2015 takes in a graph as input, represented in the CSR format (a format used for sparse matrices). This representation is equivalent to having a square matrix G with zeroed diagonal where the g_{ij} element of the matrix is the weight of the link connecting the node i with the node j. This format is represented in Fig. **??**. It requires three arrays to fully describe the graph:

- a *data* array containing all the non-zero values, where values from the same row appear sequentially from left to right and top to bottom, i.e. in *row-major* order, e.g. if the first row has 20 non-zero values, then the first 20 elements from this array belong to the first row;
- a *indices* array of the same size as *data* containing the column index of each non-zero value;
- a *indptr* array of the size of the number of rows containing a pointer to the first element in the *data* and *indices* arrays that belongs to each row, e.g. if the $i-th$ element (row) of *indptr* is k and it has 10 values, then all the elements from k to $k+10$ in *data* belong to the $i-th$ row.

Within the algorithm's context, these three arrays are denominated as *first_edge*, *destination* and *weight*, respectively. There change in denomination is for making their purposes clearer. Although these three arrays can completely describe a graph, the algorithm uses an extra array *outdegree* that stores the number of non-zero values of each row and can be deduced from the *first_edge* array. The length and purpose of each of this arrays are:

- *first_edge* is an array of size $\|V\|$, where the *i-th* element points to the first edge corresponding to the *i-th* edge.

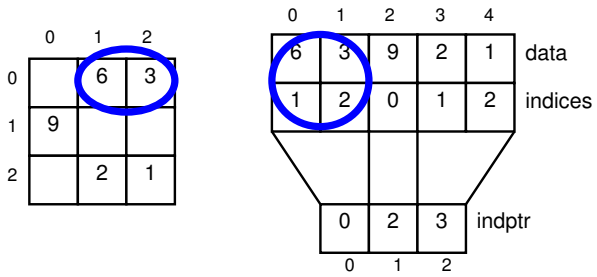


Figure 3.6: Correspondence between a sparse matrix and its CSR counterpart.

- *outdegree* is an array of size $\|V\|$, where the *i-th* element contains the number of edges attached to the *i-th* edge.
- *destination* is an array of size $\|E\|$, where the *j-th* element points to the destination vertex of the *j-th* edge.
- *weight* is an array of size $\|E\|$, where the *j-th* element contains the weight of the *j-th* edge.

V is the number of vertices and E is the number of edges. The number of edges is duplicated to cover both directions, since the algorithm works with undirected graphs. This basically means that instead of using the the upper (of lower) triangular matrix (which can also completely describe the graph), it uses the complete one resulting in double redundancy of each edge. The edges in the *destination* array are grouped together by the vertex they originate from, e.g. if edge j is the first edge of vertex i and this vertex has 3 edges, then edges $\{j, j+1, j+2\}$ are the outgoing edges of vertex i.

Steps of the algorithm

Within the context of the algorithm the *id* of a vertex is its index in the *first_edge* array, the *color* is related to the *ids* and the *successor* of a vertex is the destination vertex of one of its edges. The algorithm's flow is represented in Figure **??** and its main steps are explained below:

1. *Find minimum edge per vertex*: select and store the minimum weighted edge for each vertex and resolve same weight conflict by picking the edge with lower destination vertex id.
2. *Remove mirrored edges*: a mirrored edge the successor of its destination vertex is its origin vertex. All mirrored edges are removed from the selected edges in the first step. All edges that are not removed are added to the resulting MST.
3. *Initialize and propagate colors*: this step is responsible for identifying connected components so the graph may be contracted. Each connected component will be a super-vertex in the contracted graph, which means it will be a single vertex that is representing a subgraph. Each vertex is initialized with the same color of its successor's id. If a vertex has no successor because that edge was removed in the previous step than its color is initialized with its own id. The colors are then propagated by setting each vertex's color to the color of its successor, until convergence.

4. *Create new vertex ids*: only the super-vertices will be propagated to the next iteration but they will have new ids for building the new contracted graph. The new ids will range from 0 to s, where s is the number of super-vertices. The vertex that will represent a super-vertex is the vertex whose color is its own id. The representative vertices will take the new ids in increasing order according to their own ids in increasing order, e.g. vertex 2 is the representative with lowest id so it will have the id 0 in the contracted graph. This step relied on the *exclusive scan* operation, which is explained in section **??**.

5. *Count, assign, and insert new edges*: the final step is where the final operations for building the contracted graph are performed. The algorithm will count the number of edges that each super-vertex has connecting to other super-vertices, i.e. the connections between subgraphs. This is simply accomplished by selecting every edge whose origin and destination colors are distinct. For each such edge the corresponding positions of the origin and destination super-vertices in a new *outdegree* array are incremented with an atomic operation. The new *first_edge* array is obtained from performing an exclusive scan over *outdegree*. The next step is to assign and insert edges in the contracted graph. Once again, the algorithm will determine which edges will be in the contracted graph by checking the origin and destination colors. A copy *top_edge* of the new *first_edge* array is done to keep track of where to insert the new edges. When an edge is assigned to a super-vertex it is inserted (in the new *weight* and *destination* arrays) in the position specified in *top_edge* and the algorithm increments *top_edge* so the next edge will not overwrite the previous one. The increment is done with an atomic operation since multiple edges can be assigned to the same super-vertex and the insertion of all edges is being done in parallel. It should be noted that duplicated edges are note removed, i.e. several distinct connections between two super-vertices can be kept, since the author considers that the benefit of doing so does not outweigh the computational overhead.

It should be noted, however, that this algorithm doe not support unconnected graphs, i.e., it is not able to output a forest of MSTs. Upon contact, the author reported that a solution to that problem is, on the step of building the flag array, only mark a vertex if it is both the representative of its supervertex and has at least one neighbour.

Exclusive scan

The *scan* operation is one of the fundamental building block of parallel computing. Furthermore, two of the steps of the Borůvka variant of [**?**] are performed with an exclusive scan where the operation is a sum. To illustrate the functioning of the exclusive scan, let us consider the particular case where the operation of the scan is the sum and the identity (the value for which the operation produces the same output as the input) is naturally 0. Let us further consider the input array to be the sequence $[0, 1, 2, 3, 4, 5, 6, 7]$. Then the output will be $[0, 0, 1, 3, 5, 8, 13, 19]$. The first element of the output will be the identity (if it were an inclusive scan, it would be the first element itself). The second element is the sum between the first element of the input array and the first element of the output array, the third element

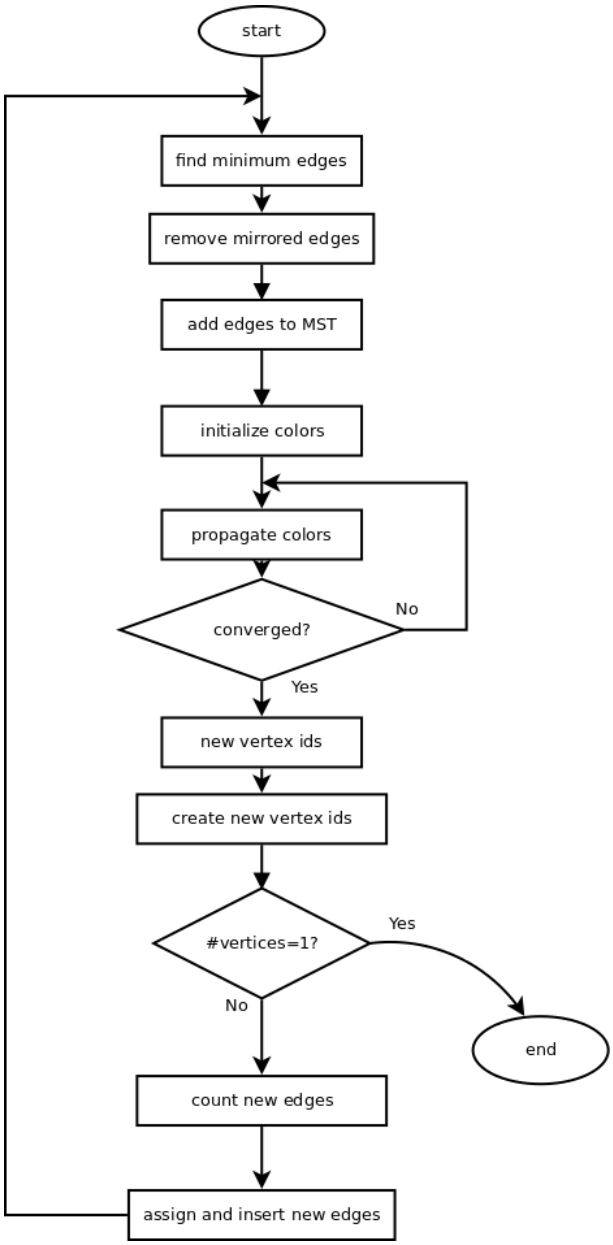


Figure 3.7: Flow execution of Sousa2015.

is the sum between the second element of the input array with second element of the output array, and so on. This algorithm seems highly sequential in nature each element of the output array depends on the previous one, but two main parallel versions can be found in literature: Hillis and Steele's [**?**] and Blelloch's [**?**]. The two approaches focus on distinct efforts: the former focus on optimizing the number of steps [**?**] while the latter focus on optimizing the amount of work done [**?**]. The focus will be on the Blelloch's algorithm.

Blelloch's algorithm is comprised by two main phases: the *reduce phase* (or *up-sweep*) and the *down-sweep phase*. The algorithm is based on the concept of *balanced binary trees* [**?**], but it should be noted that no such data structure is actually used. An in-depth explanation of this concept and how it relates to the algorithm being reviewed falls outside the scope of the present work and it is recommended that the reader consult [**?**] or [**?**] for such details.

During the reduce phase (see Figure **??**), the algorithm traverses the tree and computes partial sums at the internal nodes of the tree. This operation is also known as a parallel reduction due to the fact that the total sum is stored in the root node (last node) of the tree [**?**].

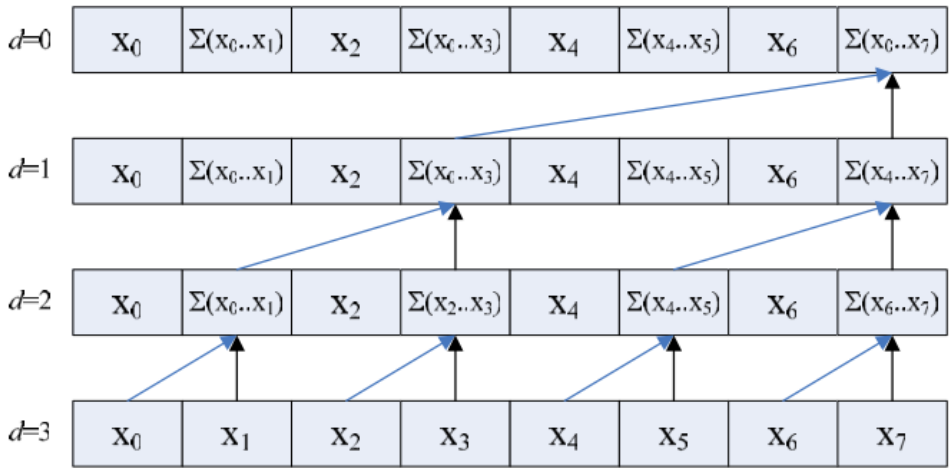


Figure 3.8: Representation of the reduce phase of Blelloch's algorithm [**?**]. *d* is the level of the tree and the input array can be observed at $d = 0$.

In the down-sweep phase (see Figure **??**) the algorithm traverses back the tree. During the traversal, and using the partial sums calculated in the reduce phase, it builds the scan in place (overwriting the result of the reduce phase).

This the computational complexity of this algorithm is higher than that of its sequential counterpart. The sequential version has a $O(n)$ computational complexity since it only goes through the input array once and performs exactly n additions. Blelloch's algorithm, in the other hand, performs $2(n-1)$ additions in the reduce phase and $n-1$ swaps in the down-sweep phase. However, since it is a parallel algorithm and the computation will be distributed across several processing units, it still performs better. It should also be noted that, as described, the algorithm supports input arrays of a size that is a power of 2. However, [**?**] explains how to overcome this limitation and also offers an implementation for CUDA and hardware specific optimizations.

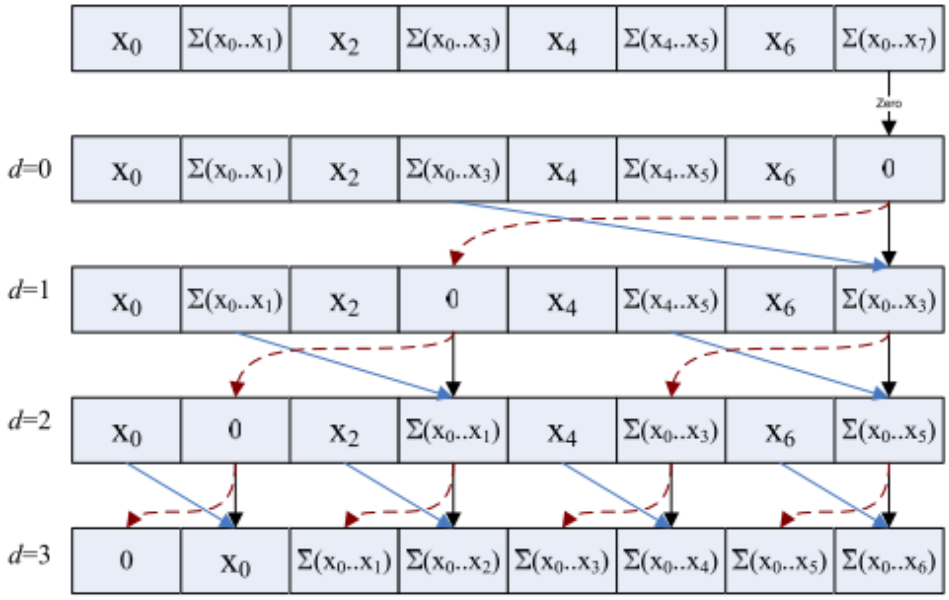


Figure 3.9: Representation of the down-sweep phase of Blelloch's algorithm [**?**]. *d* is the level of the tree.

3.3 Quantum clustering

The field of quantum clustering has shown promising results regarding potential speed-ups in several tasks over their classical counterparts. Currently, two major approaches for the concept of quantum clustering were found in the literature: quantization quantization of clustering methods to work in quantum computers or algorithms inspired by quantum physics.

The former approach translates in converting algorithms to work partially or totally on a different computing paradigm, with support of quantum circuits or quantum computers. Both **?**] and **?**] show how the quantum paradigm can be used for speed-ups in the machine learning algorithms, with the possibility of obtaining exponential speed-ups. Many of these quantizations make use of Groover's database search algorithm [**?**], or a variant of it, e.g. **?**]. Most literature on this approach is also mostly theoretical, since the physical requirements still do not exist for testing these methods. This approach can be seen as part of the bigger problem of quantum computing and quantum information processing. An alternative to using real quantum systems would be to simulate them. However, simulating quantum systems in classical computers is a very hard task by itself and literature suggest that it is not feasible [**?**]. The quantization approach falls outside the scope of this dissertation, but **?**] offers a thorough review on the state of the art of machine learning in the quantum paradigm.

The second approach is part of the wider field of Computational Intelligence. A study of the literature reveals that it typically further divides itself into two categories [**?**].One comprises the algorithms based on the concept of the qubit, the quantum analogue of a classical bit with interesting properties found in quantum objects. Several algorithms have been modeled after this concept, often also gathering inspiration from evolutionary genetic algorithms, which were successfully applied in several areas:

- tackling the Knapsack problem as described in **?**] and its improved version [**?**];

- solving the traveling salesman problem [**?**];
- two implementations of a Quantum K-Means algorithm [**? ?**];
- a Quantum Artificial Bee Colony algorithm [**? ?**];
- two approaches for Fuzzy C-Means (FCM), namely Quantum New Weighted Fuzzy C-Means (QNW-FCM) [**?**] and Quantum Fuzzy C-Means (QFCM) [**? ?**];
- a novel quantum inspired clustering technique based on manifold distance [**?**];
- a Quantum-inspired immune clonal clustering algorithm based on watershed [**?**].

The other approach models data as a quantum system and uses Schrödinger's equation in some way. Often, the data is modeled as a quantum system, where each pattern is a particle, and Schrödinger's equation is used to evolve the particle system into a solution. This has been applied in an optimization problem for electromagnetics using a quantum particle swarm [**?**] and also on several clustering algorithms:

- the Quantum Clustering [**?**] algorithm treats patterns as quantum particles whose potential is computed with Schrödinger's equation and the system is evolved with the Gradient Descent method;
- Dynamic Quantum Clustering [**?**] is a variation of **?**] that uses Schrödinger's equation to evolve the system;
- a Fuzzy C-Means approach [**?**] based on Quantum Clustering [**?**];
- QPSO+FCM, a Fuzzy C-means [**?**] based on Quantum-behaved Particle Swarm Optimization (QPSO) [**?**].

For more information on quantum inspired algorithms, the reader is referred to **?**] which offers a thorough survey on the state of the art of quantum inspired computation intelligence algorithms. The following two sections contain a brief overview of the concept of the qubit and the description of an algorithm that uses this approach. Afterwards, an algorithm that follows the Schrödinger's equation approach is reviewed.

3.3.1 The quantum bit approach

The quantum bit

To understand the algorithms based on the concept of the qubit, it is useful to cast some insight about its properties and functioning. This section has the purpose to provide a brief introduction to this topic. An extended and in-depth review of this and related topics can be found in **?**]. The qubit is a quantum object with certain quantum properties such as entanglement and superposition. Within the context of the studied algorithm, the only property used is superposition. A qubit can have any value between 0 and 1 (superposition property) until it is observed, which is when the system collapses to either state. However,

the probability with which the system collapses to either state may be different. The superposition property or linear combination of states can be expressed [**?**] as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where ψ is an arbitrary state vector and α, β are the the probability amplitude coefficients of basis states $|0\rangle$ and $|1\rangle$, respectively. The Dirac bra ket notation is employed, where the *ket* $|.\rangle$ corresponds to a column vector. The basis states correspond to the spin of the modeled particle (in this case, a ferminion, e.g. electron). The coefficients are subjected to the following normalization:

$$|\alpha|^2 + |\beta|^2 = 1$$

where $|\alpha|^2$, $|\beta|^2$ are the probabilities of observing states $[0]$ and $[1]$, respectively. α and β are complex quantities and represent a qubit:

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Moreover, a qubit string may be represented by:

$$\left[\begin{array}{c|c|c} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \end{array}\right]$$

The probability of observing the state $|000\rangle$ will be $|\alpha_1|^2 \times |\alpha_2|^2 \times |\alpha_3|^2$. To use this model for computing purposes, black-box objects called *oracles* are used. Oracles are important to understand quantum speed-ups. They can be understood as subroutines that cannot be usefully examined or as unknown physical systems that perform a quantum operation on a qubit string [**?**] and with properties one would like to estimate. Within the context of the present work an oracle is an abstraction for the programmer. It is an object which can be called and changes state (which can be observed) as a consequence. For the purpose of the following sections, the concept of the oracle is more related to *oracles with internal randomness* [**?**] or, more simply, a probabilistic Turing machine, as in the case of [**?**].

Quantum K-Means

The Quantum K-Means (QK-Means) algorithm, as described in [**?**], is based on the classical K-Means algorithm. It extends the basic K-Means with concepts from quantum mechanics (the qubit) and evolutionary genetic algorithms.

Within the context of this algorithm, oracles contain strings of qubits and generate their own input by observing the state of the qubits. After collapsing, the qubit value corresponds to a classical bit, with a binary value.

Ideally, oracles would contain actual quantum systems or simulate them - this would correctly account for the desirable quantum properties. As it stands, oracles are not quantum systems or even simulate

them and can be more appropriately described as random number generators. Each string of qubits represents a number, so the number of qubits in each string will define its precision. The number of strings chosen for the oracles depends on the number of clusters and dimensionality of the problem (e.g. for 3 centroids of 2 dimensions, 6 strings will be used since 6 numbers are required). Each oracle will represent a possible solution.

The algorithm has the following steps:

1. Initialize population of oracles
2. Collapse oracles
3. K-Means
4. Compute cluster fitness
5. Store
6. Quantum Rotation Gate
7. Collapse oracles
8. Quantum cross-over and mutation
9. Repeat 3-7 until generation (iteration) limit is reached

Initialize population of oracles The oracles are created in this step and all qubit coefficients are initialized with $\frac{1}{\sqrt{2}}$, so that the system will observe either state of the qubit with equal probability. This value is chosen taken into account the necessary normalization of the coefficients, as described in the previous section.

Collapse oracles Collapsing the oracles implies making an observation of every qubit of each string in all oracles. This is done by first choosing a coefficient to use (either can be used), e.g. α. Then, a random value r between 0 and 1 is generated. If $\alpha \geq r$ then the system collapses to $[0]$, otherwise to $[1]$.

K-Means In this step we convert the binary representation of the qubit strings to base 10 and use those values as initial centroids for K-Means. For each oracle, classical K-Means is then executed until it stabilizes or reaches the iteration limit. The solution centroids are returned to the oracles in binary representation.

Compute cluster fitness Cluster fitness is computed using the Davies-Bouldin index for each oracle. The score of each oracle is stored in the oracle itself.

Store The best scoring oracle is stored.

Quantum Rotation Gate So far, the algorithm consisted of the classical K-Means with a complex random number generation for the centroids and complicated data structures, namely the oracles. This is the step that fundamentally differs from the classical version. In this step, a quantum gate (in this case a rotation gate) is applied to all oracles except the best one. The basic idea is to shift the qubit coefficients of the least scoring oracles in the direction of the best scoring one. These oracles will have a higher probability of collapsing into initial centroid values closer to the best solution so far. This way, in future generations, the oracles do not initiate with the best centroids so far (which would not converge into a better solution) but we they are close while still ensuring diversity (which is also a desired property in the genetic computing paradigm). In other words, we look for better solutions than the one we got before in each oracle while moving in the direction of the best we found so far.

The genetic operations of cross-over and mutation are both part of the genetic algorithms toolbox. Literature suggests that this operations may not be required to produce variability in the population of qubit strings [**?**]. The reason for this is that enough variability is produced with the use of the angle-distance rotation method [**?**] in the quantum rotation operation, with a careful choice of the rotation angle. Still, when used, the goal of these operations is to produce further variability into the population of qubit strings.

3.3.2 Schrödinger's equation approach

The other approach to clustering that gathers inspiration from quantum mechanical concepts is to use the Schrödinger equation. The algorithm under study was created by Horn and Gottlieb [**?**] and was later extended by Weinstein and Horn [**?**].

The first step in this methodology is to compute a probability density function of the input data. This is done with a Parzen-window estimator in [**? ?**]. The Parzen-window density estimation of the input data is done by associating a Gaussian with each point, such that

$$\psi(\mathbf{x}) = \sum_{i=1}^{N} e^{-\frac{\|\mathbf{x}-\mathbf{x}_i\|^2}{2\sigma^2}}$$

where N is the total number of points in the dataset, σ is the variance and ψ is the probability density estimation. ψ is chosen to be the wave function in Schrödinger's equation. Further detals can be found in **? ? ?**].

With this information, one will compute the potential function $V(x)$ that corresponds to the state of minimum energy (ground state = eigenstate with minimum eigenvalue) [**?**], by solving the Schrödinger's equation in order of $V(x)$:

$$\text{V(x)} = \text{E} + \sigma^2 \frac{}{2\nabla^2\psi} \frac{}{\psi = E - \frac{d}{2} + \frac{1}{2\sigma^2\psi}\sum_{i=1}^{N}\|\mathbf{x}-\mathbf{x}_i\|^2 e^{-\frac{\|\mathbf{x}-\mathbf{x}_i\|^2}{2\sigma^2}}}$$

And since the energy should be chosen such that ψ is the groundstate (i.e. eigenstate corresponding to minimum eigenvalue) of the Hamiltonian operator associated with Schrödinger's equation (not represented above), the following is true

$$E = -min\frac{\frac{\sigma^2}{2}\nabla^2\psi}{\psi} \tag{3.1}$$

From equations **??** and **??**, $V(x)$ can be computed. This potential function is related to the inverse of a probability density function. Minima of the potential correspond to intervals in space where points are together. So minima will naturally correspond to cluster centers [**?**]. However, it is very computationally intensive to compute $V(x)$ to the whole space, so the compututation of the potential function is only done at the data points. This should not be problematic since clusters' centers are generally close to the data points themselves. Even so, the minima may not lie on the data points themselves. One method to address this problem is to compute the potential on the input data and converge this points toward some minima of the potential function. This is done with the gradient descent method in [**?**].

Another method [**?**] is to think of the input data as particles and use the Hamiltonian operator to evolve the quantum system in the time-dependent Schrödinger's equation. Given enough time steps, the particles will converge to and oscillate around potential minima. This method makes the Dynamic Quantum Clustering algorithm. The nature of the computations involved in this algorithm make it a good candidate for parallelization techniques. **?**] parallelized this algorithm to the GPU obtaining speed-ups of up to two magnitudes relative to an optimized multicore CPU implementation.

Chapter 4

Methodology

The aim of this thesis is the optimization and scalability of EAC, with a focus for large datasets. EAC is divided in three steps and each has to be considered for optimization.

The first step is the accumulation of evidence, i.e. generating an ensemble of partitions. The main objective for the optimization of this step is speed. Using fast clustering methods for generating partitions is an obvious solution, as is the optimization of particular algorithms aiming for the same objective. Since each partition is independent from every other partition, parallel computing over a cluster of computing units would result in a fast ensemble generation. Using either or any combination of these strategies will guarantee a speed-up.

Initial research was under the field of quantum clustering. After this pursuit proved fruitless regarding one of the main requirements (speed), the focus of researched shifted to parallel computing, more specifically a K-Means parallel version within the GPGPU paradigm.

The second step is mostly bound by memory. The complete co-association matrix has a space complexity of $\mathcal{O}(n^2)$. Such complexity becomes prohibitive for big data, e.g. a dataset of 2×10^6 samples will result in a complete co-association matrix of $14901\ GB$ if values are stored in single floating-point precision. This problem is addressed by exploring the inherent sparse nature of the co-association matrix.

The last step has to take into account both memory and speed requirements. The final clustering must be able to produce good results and be fast while not exploding the already big space complexity from the co-association matrix. The work in this last step was initially directed towards parallelization and afterwards out-of-core processing using a disk stored co-association matrix.

The methodology for optimizing for speed is relevant and permeates the approach taken to every problem faced in the present work. For optimizing the speed of an algorithm, one starts by first profiling said algorithm and analyze which parts take the longest to compute. These parts are the focus of optimization as any change on them will have the greatest effect on the overall algorithm. To further illustrate this point, let us consider an algorithm that spends 75% on a section of the code to which a speed-up of 2 is possible and 25% on a part to which an infinite speed-up is possible (this translates in its execution time being negligible). Let us further assume that only one part of the algorithm may be

optimized. If one optimizes the first part, the local speed-up is 2 and the overall speed-up is 1.6. On the other hand, if one optimizes the second part, the local speed-up is infinite but the overall speed-up is only $1.333(3)$.

All the algorithms were implemented in Python with high reliance on numerical and scientific modules, namely NumPy [**?**] and SciPy [**?** **?** **?**]. Important modules for visualizing and processing statistical results were MatplotLib [**?**] and Pandas [**?**], respectively. The SciKit-Learn [**?**] machine learning library has a plethora of implemented algorithms which were used for benchmarking as well as integration in the devised solutions. The iPython [**?**] module was used for interactive computing which allowed for accelerated prototyping and testing of algorithms. Python is a interpreted language which translates in its performance being worse than a compiled language such as C or Java. To address this problem, the open-source Numba module from Continuum Analytics was used to allow for writing code in native Python and have it converted to fast compiled code. Furthermore, this module provides a pure Python interface for the CUDA API. The proprietary NumbaPro module was used for two high level CUDA functions, namely *argsort* and *max*.

4.1 Quantum Clustering

Research under this paradigm aimed to find a candidate for the first phase of EAC. Two candidates were considered: Quantum K-Means (QK-Means) and Horn and Gottlieb's quantum clustering (QC)algorithm. These algorithms were chosen so as to have a representative from both approaches identified in chapter **??**.

The implementation QK-Means followed the description presented in section **??**, with the exception of the genetic operations *cross-over* and *mutation*. This change was made because literature [**?**] suggested that enough variability is produced in the quantum rotation step with the angle-distance rotation method. Furthermore, the main goal for the first phase of EAC is speed but these operations aim at improving the quality of the solution. As such, it was an implementation decision to cut overhead where possible since literature supported that no significant changes would affect the final result.

An implementation of QC was already available in Matlab. Accordingly, implementing this algorithm translated into porting the code from Matlab to Python.

4.2 Speeding up Ensemble Generation with Parallel K-Means

K-Means is an obvious candidate for the generation of partitions since it is simple, fast and partitions do not need to be accurate - variability in the partitions is a desirable property. This relaxation on the accuracy of the partitions already allows for a faster generation of partitions since fewer iterations of the algorithm need to be executed for a partition to be generated - typically 3 iterations suffice. Further gains are possible with a parallel GPU version of K-Means. The overview of the algorithm used is presented in section **??**. This section will offer implementation details.

The implementation gives as much freedom as possible to the user regarding CUDA parameter choices. By default, each thread will compute the label of one pattern and the blocks are unidimensional and composed by 512 threads. The number of blocks, then, is the number of patterns divided by the the number of threads per block. If the number of blocks exceeds the maximum allowed for one dimension (65535), the grid configuration automatically uses other dimensions. The other parameter that the user can choose deals with how data is transfered back and forward between host and device. The user can choose to allow the CUDA API to handle all the memory transfers or to make the implementation optimize those. The latter will minimize the amount of data transfers and memory allocations and is the default option. Furthermore, it also allows for the algorithm to be executed multiple times without redundant transfer of the pattern set. This permits for the production of an entire ensemble with a single transfer of the pattern set. These parameters (number of patterns per thread, number of threads per block and memory transfer mode) may be configured at runtime. Still, a typical user does not need to be knowledgeable about CUDA to be able to use this implementation, since the tuning of these parameters is optional.

The CUDA implementation of the label computation part of the algorithm starts by transfering the data to the GPU (pattern set and initial centroids) and allocates space for the labels and corresponding distances. 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 centroids. This procedure is available as a CUDA kernel or as three different sequential versions: pure Python, based on NumPy or compiled code with Numba. These same sequential versions exist for the recomputation of the 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, although empty clusters may be desirable in some situations, the target use expects that the output number of clusters to 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.

4.3 Dealing with the space complexity of the co-association matrix

In chapter **??**, two approaches to the space complexity of the co-association matrix in the second phase of EAC are presented: one dealing with p prototypes and the other with the inherent sparsity of the matrix.

The results presented in the sparsity study of EAC [**?**] show that it is possible to obtain a high sparsity in the co-association matrix. In some comes the density of the matrix was as low as 1%. This motivated the focus of the work on exploiting the inherent sparsity of the matrix. A discussion on using

sparse matrices and a novel solution for building the co-association matrix is presented **??**.

As stated before, the focus of the work is on sparse matrices. Still, for the sake of completeness, the different strategies considered for the prototypes approach are discussed here. One strategy is to use the **p-Nearest Neighbors** as prototypes, as already presented in chapter **??**. Before building the co-association matrix, the p closest samples to each sample are computed and stored in the $n \times p$ neighbor matrix. During the voting mechanism of EAC only the neighbors of each pattern are considered. This strategy has a space complexity of $O(2nk)$ and requires the computation of the k-Nearest Neighbors. A second strategy is to use **p random prototypes**, which will a set of unique p patterns. This will be the same for every sample. Here the voting mechanism is altered so that if a sample is clustered with any of the prototypes, the correspondent element in the co-association matrix is incremented. This has the advantage that only a $n \times p$ matrix needs to be stored along with a p array for the chosen prototypes. Furthermore, if p if high enough to provide a representative sample of the dataset the results can be as good as the full matrix. The third, and final, possible strategy identified is similar to the random prototypes. The difference is that instead of choosing p random samples from the dataset, the prototypes will be the representatives of the dataset from another algorithm, e.g. K-Medoids, K-Means.

It would be ideal to combine both approaches (sparsity and neighbors) to further reduce space complexity, but they are not necessarily compatible. This is specially true for the p-Nearest Neighbors strategy since it is unlikely that a sample will never be clustered with its closest p neighbors. This means that the $n \times p$ co-association matrix may not have many zeros which translates in little advantage for using the sparsity augmentation approach the matrix of co-associations.

4.4 Building the sparse matrix

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 overhead.

Documentation of the SciPy library recommends different sparse formats for either building a matrix or operating over it. For building a matrix, the COO (COOrdinate), DOK (Dictionary of Keys) and LIL (List of Lists) formats are recommended. All of these formats rely on extra data structures such as lists as dictionaries to efficiently build the matrices incrementally, i.e. allocating memory as it is required. 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).

As observed before, the sparsity study of EAC [**?**] did not account for the overhead of using the data structures supporting sparse matrices. However, scaling to very large data sets must take this into account. Exploratory testing revealed that using one of the recommended matrices allowed for building the matrix took around two orders of magnitude as long as using a full allocated matrix. Furthermore,

because of the extra data structures, the implementations from SciPy would saturate the main memory from data sets with less than 1 million patterns. Using the CSR format did not saturate memory but it took a completely unacceptable amount of time, many orders of magnitude above any of the formats recommended for building the matrix.

After a search in the literature for efficient and scalable ways to build a sparse matrix proved fruitless, this issues motivated the design of a new method for building the matrix specialized for the characteristics of EAC. The solution devised is based on the CSR format, which has the desired properties of having a low memory trace and allows for fast computations.

4.4.1 EAC CSR

The first step is making an initial assumption on the maximum number of associations *max_assocs* that each sample 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 a sample will have since it is the limit of how many associations each sample will have in any partition. Furthermore, one would expect that the neighbors of each sample will not vary significantly, i.e. the same neighbors will be clustered together with a sample repeatedly in many partitions. This scheme for building the matrix uses 4 supporting arrays:

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

Each sample (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 sample. Throughout this section, the interval of the *indices* array corresponding to a specific pattern (row) is referred to as the *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. Furthermore, it should be noted that this method assumes that the clusters received come sorted in a increasing order.

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* array in the positions of each sample, with the exclusion of the sample itself. The data array (containing the co-association score) is set to 1 on the relevant positions. This process can be observed clearly in Fig. **??**. Because it is the fastest partition to be inserted, it is

picked to be the one with the least amount of clusters (more samples per cluster) so that each sample gets the most amount of associations in the beginning (on average). This is only applicable if the whole ensemble is provided, otherwise there is no way to know what is the biggest cluster of the whole ensemble. This increases the probability that any new cluster will have more samples that correspond to already established associations. Because the clusters are sorted and only a copy of each cluster was made, it is not necessary to sort each row of the matrix by column index.

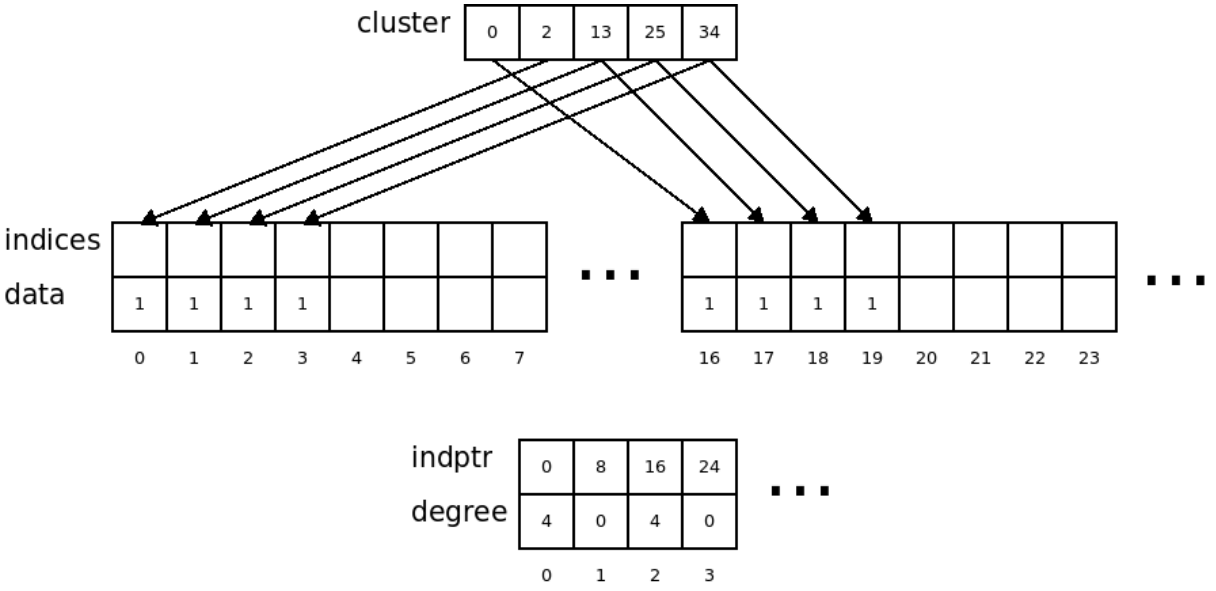


Figure 4.1: Inserting a cluster of the first partition in the co-association matrix.

For the remaining partitions, the process is different. Now, it is necessary to check if an association already exists. For each sample in a cluster it is necessary to add or increment the association to every other sample in the cluster. This is described in Algorithm **??**.

Algorithm 1 Update matrix with cluster.

```
1: procedure UPDATE_CLUSTER(indices, data, indptr, degree, cluster, max_assocs)
2:     for sample n in cluster do
3:         for every other sample na in cluster do
4:             ind = binary_search(na, indices,interval of n in indices)
5:             if ind ≥ 0 then
6:                 increment data[ind]
7:             else
8:                 if maximum assocs not reached then
9:                     add new assoc. with weight 1
```

The binary search is used to search for the association in the indices array in the interval corresponding to a specific row (or pattern). 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 samples in any given cluster, the indices of each sample must be in a sorted state at the beginning of each partition insertion. Two strategies for sorting were devised. The first strategy is to just insert the new associations at the end of the *indices* array (in the correct interval for each sample) and then, at the end of processing each partition, use a sorting algorithm to sort all the

samples' intervals in the *indices* array. This was easily implemented with existing code, namely using *NumPy*'s *quicksort* implementation, which has an average time complexity of $O(n \log n)$

The second strategy is more complex. It 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. Furthermore, this would be done 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 $-ind - 1$, where ind is the position for a sorted insertion of the key in the array, if it is not found. This means that the position where each new association should be inserted is now available. Instead of just adding the new associations after the old ones of each sample, 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. **??**, detailed in Algorithm **??** and explained in the following paragraph.

After each pass on a cluster (adding or incrementing all associations to a sample in the cluster), 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. An element is added to the end of the *new_assoc_idx* array with the value of *degree[i]* so that the last new association can be included in the general cycle. 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 sample). 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 copied to the end of the *indices* interval, i.e. they 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.

4.4.2 EAC CSR Condensed

The co-association matrix is symmetric, which means that only half is necessary to describe the whole matrix. That fact has consequences on both memory usage and computational effort on building the matrix. In the case of the fully allocated matrix, this translates in a reduction to 49.5% of the memory required. Furthermore, only half the operations are made since only half the matrix is accessed, which also accelerates the building of the matrix. There is, however, a small overhead for translating the two dimensional index to the single dimensional index of the upper triangle matrix. When using a sparse matrix according to the EAC CSR scheme described above, there is no direct memory usage reduction. However, the number of binary searches performed by inserting new associations is half which has a significant impact on the computation time relative to the complete matrix.

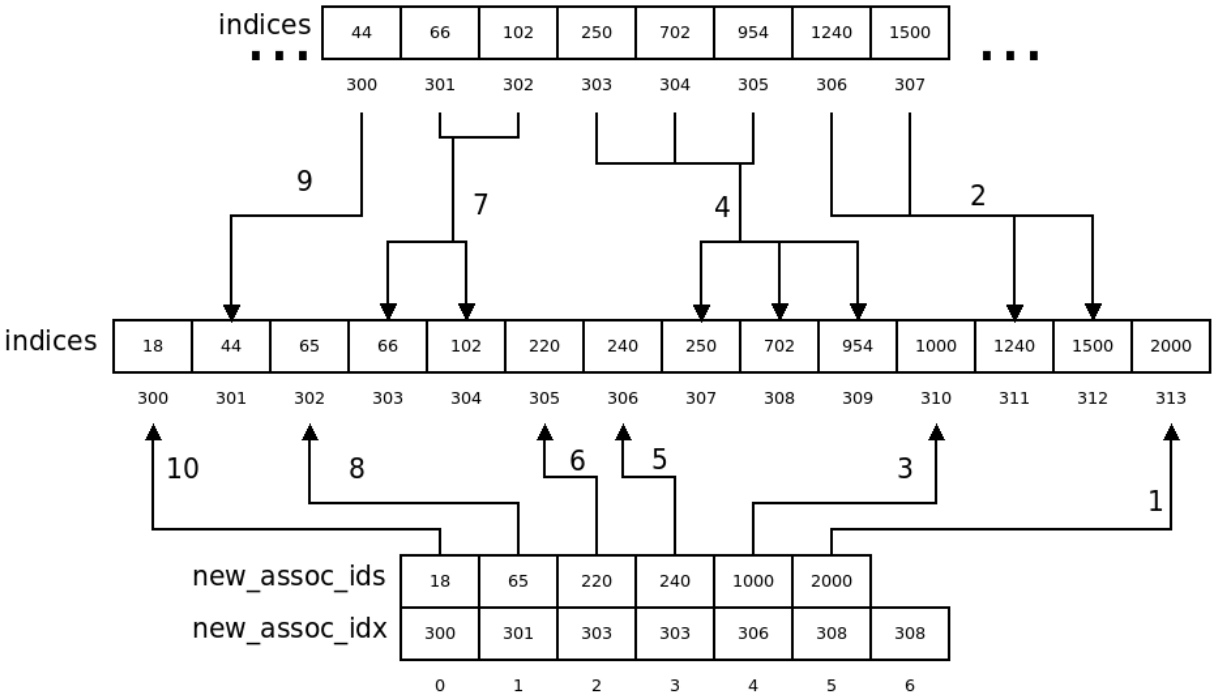


Figure 4.2: Inserting a cluster from a partition in the co-association matrix. The arrows indicate to where the indices are moved. The numbers indicate the order of the operation.

Algorithm 2 Sort the *indices* array in the interval of a sample n.

procedure SORT_INDICES($indices, data, indptr, degree, n, new\_assocs\_ptr, new\_assocs\_ids, new\_assocs\_idx$)
    $new\_assocs\_idx[new\_assocs\_ptr] = indptr[n] + degree[n]$
    $i = new\_assocs\_ptr$
    $o\_ptr = indptr[n] + new\_assocs\_ptr + degree[n] - 1$
    while $i \geq 1$ **do**
        $start\_idx = new\_assocs\_idx[i-1]$
        $end\_idx = new\_assocs\_idx[i] - 1$
        while $end\_idx >= start\_idx$ **do**
            $indices[o\_ptr] = indices[end\_idx]$
            $data[o\_ptr] = data[end\_idx]$
            $end\_idx- = 1$
            $o\_ptr- = 1$
        $indices[o\_ptr] = new\_assocs\_ids[i-1]$
        $data[o\_ptr] = 1$
        $o\_ptr- = 1$
        $i- = 1$

Even though there is no direct memory reduction for switching to the condensed matrix form, this scheme does open possibilities for it. Previously, the number of associations for each sample (number of non zero values in each row) remained constant throughout the whole set of patterns. However, using a condensed scheme means that the number of associations decreases as one steps further to the end of the set, i.e. closer to the bottom the co-association matrix. An example of this can be seen in the plot of the number of associations per sample in a complete and condensed matrix of Fig. **??**. It is clear that, since the number of associations that is actually stored in the matrix decreases throughout the matrix, the pre-allocated memory for each sample can decrease accordingly. One possible strategy, illustrated in Fig. **??**, is to decrease the maximum number of associations linearly. In the example, the first 5% of samples have the 100% of the maximum number of associations and it decreases linearly until 5% of the maximum number of associations for the last sample.

Table **??** shows the memory usage of the different co-association matrix strategies in the general case and for an example of 100000 patterns. The memory consumption for the *sparse condensed linear* type of matrix if given for the parameters presented above for Fig. **??**.

Table 4.1: Memory used for different matrix types for the generic case and a real example of 100000 samples. The relative reduction (R.R.) refers to the memory reduction relative to the type of matrix above, the absolute reduction (A.R.) refers to the reduction relative to the full complete matrix.

Type of matrix	**Generic**	**Memory [MBytes]**	**R.R.**	**A.R.**
Full complete	N^2	9536.7431	-	-
Full condensed	$\frac{N(N-1)}{2}$	4768.3238	0.4999	0.4999
Sparse constant	$0.54 \times N \times max\_assocs$	678.8253	0.1423	0.0711
Sparse condensed linear	$N \times max\_assocs$	372.8497	0.5492	0.0390

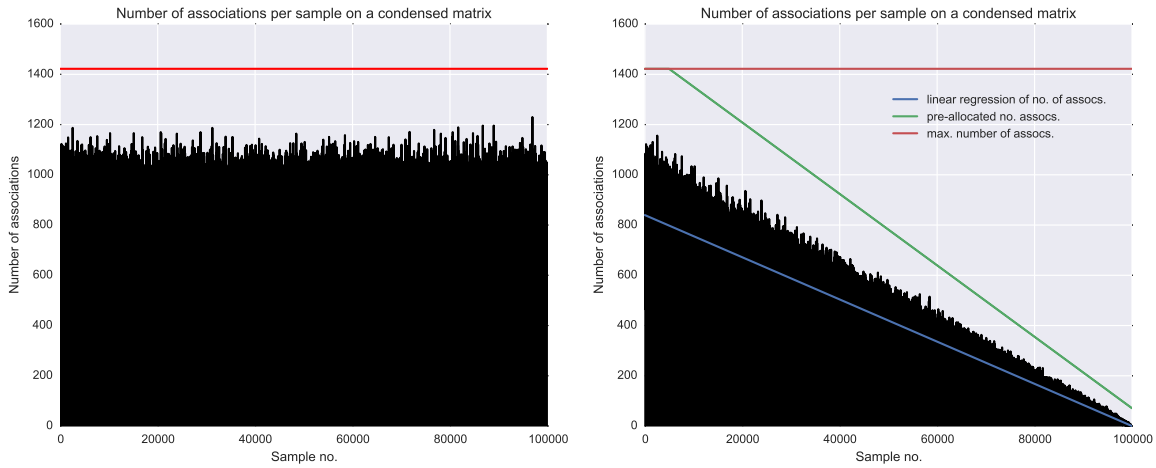


Figure 4.3: The left figure shows the number of associations per sample in a complete matrix; the right figure shows the number of associations per sample in a condensed matrix.

4.5 Final partition recovery

This section will present the considered candidates for the final step of EAC. A GPU version of SL is described in section **??**. External algorithms are a candidate solution and this approach is presented in section **??**. 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.

4.5.1 Single-Link and GPGPU

Single-Link is an inherently sequential algorithm which means an efficient GPU version is hard to implement. However, SL can be performed by computing the MST and cutting edges until we have the desired number of clusters, as described in chapter **??**. Considerable speed-ups are reported in the literature for MST solvers in the GPGPU paradigm. The considered solution uses the efficient parallel variant of Borůvka's algorithm [**?**]. After the necessary pruning of the MST edges has been performed to achieve the desired number of clusters, the output MST is fed to the labeling algorithm which will provide the final clustering.

Generating the MST

The output of the Borůvka's algorithm is a list of the indices of the edges constituting the MST. These indices point to the *destination* of the original graph. The original algorithm [**?**] assumes connected graphs, but this is not guaranteed in EAC's co-association matrix. In graph theory, given a connected graph $G = (V, E)$, there is a path between any $s, t \in V$, where V is the set of vertices in G and E is the set of edges that connects the vertices. In an unconnected graph, this is not the case and unconnected subsets are called components, i.e. a connected component $C \subset V$ ensures that for each $u, v \in C$ there is a path between u and v. In the present implementation, the issue of unconnected graphs was solved in the first step (finding the minimum edge connected to each vertex or supervertex). If a vertex has no edges connected to it (an outdegree of 0 since all edges are duplicated to cover both directions) then it is marked as a mirrored edge. This means that the independent components will be marked as supervertices in all subsequent iterations of the algorithm. The overhead of having these redundant vertices is low since the number of independent components is typically low compared to the number of vertices in the the graph and the processing of such vertices is very fast. As a consequence, the stopping criteria becomes the lack of edges connected to the remaining vertices, which is the same as saying that all elements of the *outdegree* array are zero. This condition can be checked as a secondary result of the computation of the new *first_edge* array. This step is performed by applying an exclusive prefix sum over the *outdegree*. The prefix sum was implemented in such a way that it returns the sum of all elements, which translates in very low overhead to check this condition. The final output is the MST array and the number of edges in the array. The later is necessary because the number of edges will be less than $|V| - 1$ when independent components exist, and also because the MST array is pre-allocated

in the beginning of the algorithm when the number of edges is not yet known.

Number of clusters and pruning the MST

The number of clusters can be automatically computed with the lifetime technique or be supplied. Either way, a list (*mst_weights*) with the weights of each edge of the MST is compiled and ordered. The list of edges is also ordered according to the order of the *mst_weights*. If the number of clusters k was given, the algorithm removes the $k-1$ heaviest edges. If there are independent components inside the MST those are taken into consideration before removing any edges. If the number of clusters k is higher than the number of independent components the final number of clusters will be the number of independent components.

To compute the number of clusters (which results in a truly unsupervised method) the lifetimes are computed. In the traditional EAC, lifetimes are computed on the weights of the clusters by the order they are formed. With the MST, the lifetimes are computed over the ordered *mst_weights* array, which is equivalent. If there are independent components, an extra lifetime is computed from the representative weight connecting independent components. This lifetime will serve as the threshold between choosing the number of independent components as the number of clusters or looking into the lifetimes within the MST. This is because links between independent vertices are not included in the MST. For this reason, the lifetime for going from independent edges to the heaviest link in the MST (where the first cut would be performed) is computed separately. If the maximum lifetime within the MST is bigger than the threshold, than the number of clusters is computed as in traditional EAC plus the number of independent components. Otherwise, the independent components will be the final clusters. Most of this process can be done by the GPU. Library kernels were used to sort the array and compute the $argmax$, and a simple kernel was used to compute the lifetimes.

Building the new graph

The final MST (after performing the pruning) is then converted into a new, reduced graph. The function responsible for this takes the original graph and the final selection of edges that composing the MST and, afterwards, produces a graph in CSR format. The function has to count the number of edges for each vertex, compute the origin vertex of each edge (the original *destination* array only contains the destination vertices) and, with that information, build the final graph. This process can be done by the GPU with simple mapping kernels and a modified binary search for determining the origin vertex.

Final clustering

The last step is computing the final clusters. Any incision in the MST translates in independent components in the constructed graph. This means that the problem of computing the clusters translates into a problem of finding independent connected components in a graph, a problem that usually goes by the name of Connected Component Labeling. To implement this part in the GPU, the aforementioned Borůvka algorithm was modified to output the an array *labels* of length $|V|$ such that the $i-th$ position

contained the label of the $i-th$ vertex. To this effect, the flow of the algorithm was slightly altered as shown in Figure **??**. The kernel dealing with the MST was removed and a kernel to update the labels at each iteration, shown in Algorithm **??**, was implemented. In the first iteration of the algorithm the converged colors are copied to the labels array. In every iteration the kernel to update the labels takes in the propagated colors and the array with the new vertex IDs. For each vertex in the array, the kernel first takes in the the color of the current vertex and maps it to the new color (one should notice that the color is actually a vertex ID and that that vertex has had its color updated). Afterwards, the kernel maps the new color with the new vertex ID that color will take, to keep consistency with future iterations.

Algorithm 3 Update component labels kernel

1: **procedure** UPDATE_LABELS($vertex\_id, labels, colors, new\_vertex$)
2: $curr\_color \leftarrow labels[vertex\_id]$
3: $new\_color \leftarrow colors[curr\_color]$
4: $new\_color\_id \leftarrow new\_vertex[new\_color]$
5: $labels[vertex\_id] \leftarrow new\_color\_id$

Memory transfer with the GPU

The whole algorithm of computing the SL clustering has been implemented in the GPU with minimizing the memory utilization in mind. Transferring the initial graph is the most relevant memory transfer. It has to be transfered twice: first for computing the MST and then to build the processed MST graph. This happens because the initial device arrays used for the graph are deallocated to give space for the arrays of subsequent iterations of the MST algorithm. This implementation design had in mind memory consumption in mind and could easily be avoided with the cost of having to store the bigger initial graph for the entire duration of the MST computation, which might be worthwhile if the GPU memory is abundant. The final labels array is transfered back to the host in the end of computation, but its size is small relative the to the size of the original graph. Furthermore, because the control logic is processed by the host and it is dependent on some values computed by the device, extra memory transfers of single values (e.g. number of vertices and number of edges on each iteration) are necessary. These, however, may be safely dismissed since they are of little consequence in the overall computation time.

4.5.2 Single-Link and external memory algorithms

The co-association matrix can become very large for large data sets. Even using the EAC CSR method to reduce memory usage, the memory used occupy a significant portion of main memory. Furthermore, the algorithm to compute a MST brings additional space complexity that make it infeasible to perform the final partition recovery on a large co-association matrix in main memory. External memory algorithms address this issue. External memory algorithms refers to algorithms that are based on disk (or other large capacity but typically low latency storage technology), usually by loading small batches to main memory, processing them, saving the results and repeat until the whole computation is done. This section describes the process of building the MST with low memory usage. The details of converting the MST in a Single-Link clustering are described in chapter **??** and omitted here.

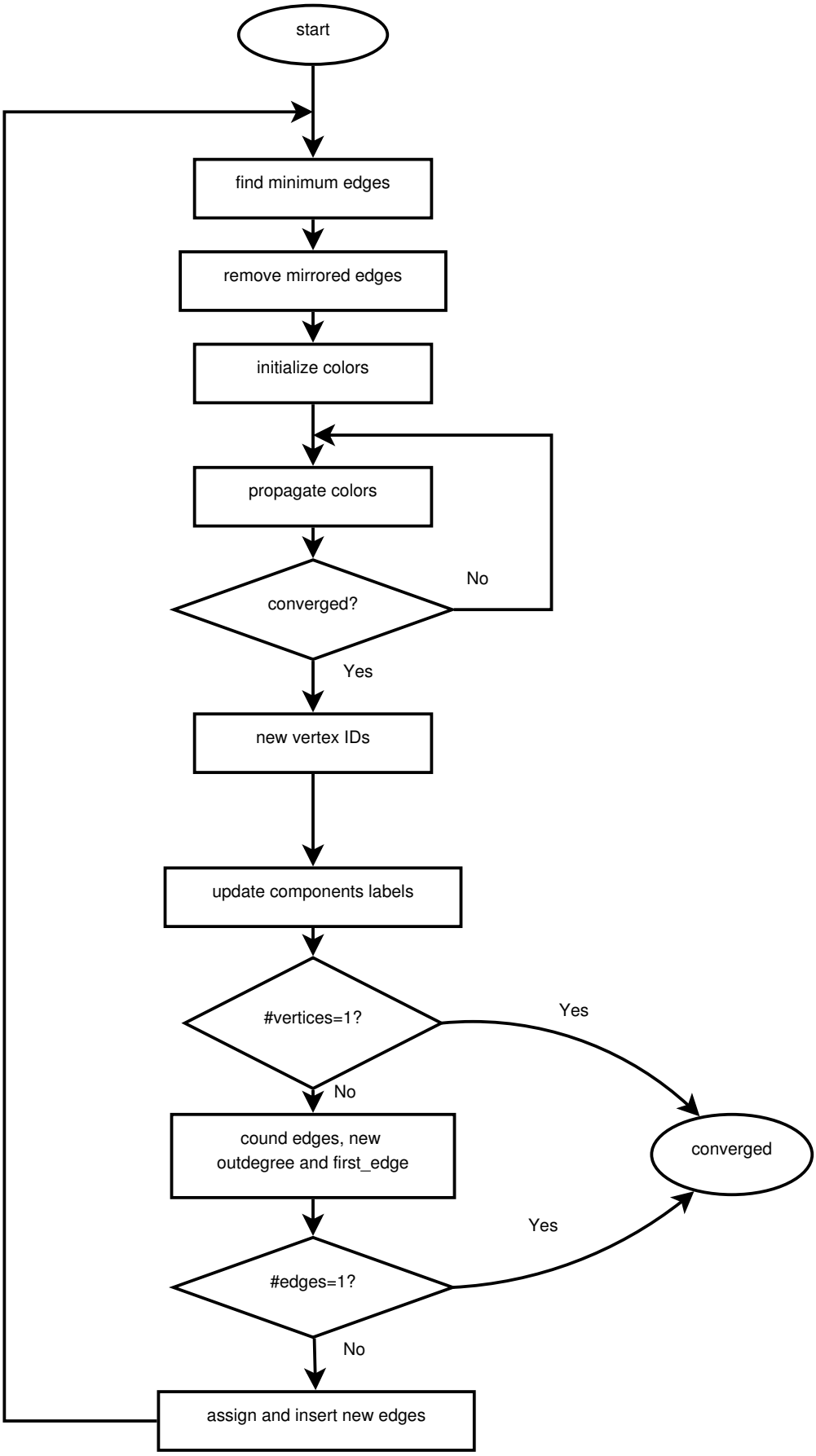


Figure 4.4: Diagram of the connected components labeling algorithm used.

Kruskal's algorithm and implementation

The sequential MST algorithm used was Kruskal. The original paper of Kruskal's [**?**] 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 if connected, the algorithm will stop before processing all edges when $|V| - 1$ edges are added to the MST. Within the EAC context, it is very common for the co-association graph to have independent components which translates in every edge being processed. It should be noted that 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 useful to avoid computing the edge with minimum (*min*) weight in every iteration. Doing so would increase time complexity since a sorting algorithm may have an average time complexity to $O(nlogn)$ (QuickSort algorithm) while a single minimum has a time complexity of $O(n)$. Computing the minimum at every iteration when every edge must be processed means that there will be $|E|$ iterations and so the total time complexity for the minimum operation alone will be $O(E^E)$ if processed edges are only given the maximum value possible or $O(E!)$ if they are removed. These time complexities are much higher than a $O(ElogE)$. The result of this operation is an array of length $|E|$ (i.e. the number of associations in EAC), which means it will occupy at least the same size as the array containing the weights of the edges. However, the total size 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 a the use o 8 bytes for the argsort array. This is the real motivation to store the co-association matrix in disk and use an external sorting algorithm.

Kruskal's implementation with an external sorting algorithm

The *PyTables* library [**?**], which is built on top of the *HDF5* format [**?**], 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, an expanded version is stored instead. The expanded version will be of the same length as the *indices* array and 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) of the *data* array of the CSR matrix. The details of CSI implementation fall outside the scope of this dissertation but are explained in further detail in [**?**]. 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 that can be disregarded.

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 calls of the function. The new implementation loads the graph in batches and in a sorted manner, e.g. first load a batch of the 1000 smallest edges, then a batch of the next 1000 smalled edges, etc. Each batch must be processed sequentially since the edges must be processed in a sorted manner, which means that there is no possibility for parallelism in this process. Typically, the batch size is a very small fraction of the size of the edges, which means that 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 [**?**]. As an example, for a 500000 pattern set the SL-MST approach took 54.9 seconds while the external memory approach took 2613.5 seconds - an order of magnitude higher.

4.6 Resume of the developed work

This chapter exposed a great deal of the actual developed work. It covered solutions and implementations related to the three phases of EAC. For the production of the ensemble, two quantum clustering algorithms and a GPU parallel K-Means variant were implemented. These are algorithms existing in the literature (presented in chapter **??**) and implementation details were presented when relevant. For addressing the computation complexity of the second phase of EAC, a novel method **EAC CSR** for building the co-association matrix was designed and implemented. Finally, three approaches are presented for the last phase of EAC. An implementation of a parallel GPU MST algorithm based on Borůvka's algorithm was presented along with a labeling algorithm for a forest of MSTs. Also, an implementation of SL-MST based on Kruskal's algorithm and an external memory variant of it were presented. The core of the external memory variant is based on using an external algorithm for sorting a very big array stored in disk and, although this is not something new, its application within the SL clustering framework was not found in literature.

Chapter 5

Results

5.1 Experimental environment

All experiments were carried out in one (or more) of three distinct machines what will be referred to as **Alpha**, **Bravo** and **Charlie**. Their CPU and GPU hardware configurations are described in Tables **??**, **??** and **??**, respectively. **Charlie** also has a Seagate ST2000DM001 7200 rpm spinning disk and a Samsung 840 EVO Solid State Drive with sequential read/write speeds of up to 540/410 MB/s

Table 5.1: **Alpha** machine specifications.

	CPU	**GPU**
# Devices	1	1
Manufacturer	Intel	NVIDIA
Model	i3-2310M	GT 520M
Launch date	Q1'11	Q1'2011
Architecture	Sandy Bridge	Fermi
# Cores	2	48
Clock frequency [Mhz]	2100	1480
L1 Cache	64KB IC [a] + 64KB DC [b]	16/48 KB/SM [c]
L2 Cache	512KB	n/a
L3 Cache	3 MB	n/a
Memory [GB]	4	1
Max. memory bandwidth [Gbps]	21.3	12.8

[a]Instruction Cache (IC)
[b]Data Cache (IC)
[c]Each Streaming Multiprocessor has 64 KB of on-chip memory that can be configured as either 16KB of L1 cache and 48 KB of shared memory, or vice versa.

Table 5.2: **Bravo** machine specifications.

	CPU	**GPU**
# Devices	1	1
Manufacturer	Intel	NVIDIA
Model	i7 4770K	K40c
Launch date	Q2'13	Q4'13
Architecture	Haswell	Kepler
# Cores	4	2880
Clock frequency [Mhz]	3500	745
L1 Cache	128 KB IC [a] + 128 KB DC [b]	16/48 KB/SM [c] + 48KB DC [d]
L2 Cache	1 MB	1.5 MB
L3 Cache	8 MB	n/a
Memory [GB]	32	12
Max. memory bandwidth [Gbps]	25,6	288

[a]Instruction Cache (IC)
[b]Data Cache (IC)
[c]Each Streaming Multiprocessor has 64 KB of on-chip memory that can be configured as either 16KB of L1 cache and 48 KB of shared memory, or vice versa.
[d]The Kepler architecture has an extra read-only 48KB of Data Cache at the same level of the L1 cache.

Table 5.3: **Charlie** machine specifications.

	CPU	**GPU**
# Devices	1	1
Manufacturer	Intel	NVIDIA
Model	i7-4930K	Quadro K600
Launch date	Q3'13	Q1'2013
Architecture	Ivy Bridge	
# Cores	6	192
Clock frequency [Mhz]	3400	876
L1 Cache	192 KB IC [a] + 192 KB DC [b]	16/48 KB/SM [c] + 48KB DC [d]
L2 Cache	1,5 MB	1.5 MB
L3 Cache	12 MB	n/a
Memory [GB]	32	1
Max. memory bandwidth [Gbps]	59,6	28,5

[a] Instruction Cache (IC)
[b] Data Cache (IC)
[c] Each Streaming Multiprocessor has 64 KB of on-chip memory that can be configured as either 16KB of L1 cache and 48 KB of shared memory, or vice versa.
[d] The Kepler architecture has an extra read-only 48KB of Data Cache at the same level of the L1 cache.

5.2 Parallel K-Means

5.3 GPU MST

USA graphs from 9th DIMACS Implementation Challenge

5.4 EAC

Chapter 6

Discussion?

6.1 real num assocs compared to samples per cluster

[**?**] reports that, on average, the overall contribution of the clustering ensemble (including unbalanced clusters) duplicates the co-associations produced in a single balanced clustering with Kmin clusters. However, the spectrum of datasets evaluated regarding cardinality was smaller than that evaluated in the present work. The results suggest that the contribution of the ensemble is in fact higher.

6.2 Trade-off speed accuracy memory

When a problem of clustering of big data is at hand, the user should reflect upon what the problem at hand really requires: speed or accuracy. The user should take into consideration the nature of the data and the requirements of the problem (concerning speed and accuracy) before proceeding to the execution of the analysis. The present body of work reflects a method of clustering over big data using a high accuracy, but also high cost, method. Other methods offer the opposite, low cost, low to average accuracy.

6.3 GPU MST

The parallel version of the final step of EAC showed a slowdown relative to its sequential counterpart. This slowdown is related with the performance of the MST algorithm. The implementation of the algorithm was tested in some of the same graphs as those reported in [**?**] and revealed a speedup. However, when using the this MST solver on the target graphs (the co-association matrices) not only there was no speedup, but a significantly slowdown was observed, reaching up to nine times slower. The underlying reason for this is believed to be that the number of edges to node ratio of these graphs is low compared to that typically seen in co-associations matrix, even when using a prototype subset of the original matrix. Since the parallel computation is anchored to nodes, the workload per node is higher from the beginning and can increase significantly as the algorithm progresses. Furthermore, the

workload can become highly unbalanced with some threads having to process hundreds of thousands of edges while others only a few thousands.

6.4 Expanding this work to other scalability paradigms

The present work focused on two main approaches for scalability: GPGPU and out-of-core solution. A current trend in computing of big data is cluster computing, which allows for distributed and parallel computing. For the sake of completeness it is interesting to discuss if the ideas related to the former two paradigms are transferable to the later.

The concept of GPGPU is one of parallelization. It is based on the fact that each computing thread in the GPU will execute instructions on a restricted subset of the entire dataset. This idea is easily transferable to cluster computing, as it is a core concept on both paradigms. For this reason, the generation of the ensemble is a step of EAC that can be very easily transfered to a computing cluster.

Chapter 7

Conclusions

Insert your chapter material here...

7.1 Achievements

The major achievements of the present work...

7.2 Future Work

Much effort was put developing and testing co-association matrix building strategies. The schemes presented here provide a solid framework to work with large data sets in this middle step of EAC. As such, interesting directions for this work to take are testing how state of the art algorithms designed for Big Data would complement EAC in the first and last steps.

The programming model used for the GPU was CUDA, used through a Python library called Numba. This library offers an interface to access most of CUDA's capabilities, but not all. One of those capabilities is Dynamic Parallelism. This offers the ability of having a host kernel call on other host kernel without intervention from the host. This translates in the possibility of moving the control logic in the Borůvka variant (and also its the Connected Components Labeling variant) to the device, effectevely removing the memory transfer of values related with the control logic.

Adaptation of the present implementation to OpenCL. This brings major benefits in respect to portability since OpenCL supports most devices. Moreover, OpenCL's performance is catching in on that of CUDA's and since it's programming model was based on CUDA, it should be straightforward for developers to make the switch.

Application of EAC to the MapReduce framework will further expand the possibilities of application of EAC.

Study the integration of other clustering algorithms within the the EAC toolchain.

A good follow-up of the present work is to study the relationship between several metrics (e.g. sparsity, accuracy, maximum number of co-associations) and the complexity of the dataset. Some metrics

for describing the complexity of the dataset exist (Tin Kam Ho) and it would be interesting to profile several datasets of different complexities and structures and search for the former relationship. On a performance perspective it could prove useful to deduce better rules to set the maximum number of associations in the sparse matrix. On a accuracy perspective it would be interesting to see if there are types of datasets that simply are not a good fit for EAC while other are. It would also be interesting to relate complexity with the threshold cut-off.

The WEAC algorithm is focused on improving accuracy. The underlying concept is to measure the quality of the partitions in the ensemble and allow the better ones to be more influential in the co-association matrix. The concept of measuring the quality of the partitions may prove useful for further decreasing the memory complexity with the EAC CSR scheme without compromising too much accuracy. The basic idea is to choose a max_assocs value that will likely be less than the number of associations many patterns will have, which will result in some associations being discarded. The associations that will be kept are the ones from the first partitions that were processed. With this in mind, one could order the partitions by quality and start the processing from those with better quality.