Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Evaluation of Structured Parallelization of k-Means-Based Clustering Methods

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von:	Maximilian Mackeprang
geboren am:	23.01.1990
geboren in:	Kiel
Gutachter/innen:	Prof. Dr. rer. nat. Jens-Peter Redlich
	Prof. Dr. sc. nat. Joachim Fischer
eingereicht am:	
verteidigt am:	

Abstract

At the present time, data size and complexity of clustering tasks (e.g. machine learning) demand parallel computing on multiple cores. However, designing parallel applications using threads and blocking is prone to non-trivial errors. This thesis uses an structured parallelization approach called **algorithmic skeletons** to build a model to express variants of one of the most used algorithms in data analysis: **k-Means**. We compare different variations of the k-Means algorithm to establish a common ground between them. The similarities are used to build a **general k-Means skeleton** that can be used to implement different k-Means variations. Based on the interface, different parallelization possibilities are shown. Furthermore the runtime properties of the original k-Means algorithm, a kd-tree based version, k-Median and fuzzy c-Means are evaluated in an experimental setup, using the algorithmic skeleton framework **Skandium** to provide the implementations.

Abstract(german)

Die Komplexität und Datenmenge heutiger Computersysteme zur Clusteranalyse (z.B. im maschinellen Lernen) erfordert die parallele Verarbeitung der erforderlichen Berechnungen, um die Laufzeit in annehmbaren Grenzen zu halten. Ein Problem dabei ist, dass das Erstellen paralleler Applikationen unter Verwendung von Threads und locking Mechanismen anfällig für nicht-triviale Fehlerfälle ist. Ansätze strukturierter Parallelisierung versuchen dieses Problem zu lösen, in dem sie Anwendungsentwicklern Modelle der nebenläufigen Programmierung bereitstellen. Diese Arbeit benutzt einen „**algorithmic skeletons**“genannten Ansatz zur strukturierten Parallelisierung, um ein Modell für verschiedene Varianten von einem der meistgenutzten Algorithmen in der Datenanalyse auszudrücken: **k-Means**. In der Arbeit werden verschiedene Varianten des Algorithmus verglichen und auf Gemeinsamkeiten untersucht. Des Weiteren wird ein Überblick über verschiedene Ansätze zur Parallelisierung des Algorithmus gegeben. Ausgehend von diesen Erkenntnissen wird eine Schnittstelle für ein **allgemeines Modell des k-Means Algorithmus** entworfen und als algorithmic skeleton umgesetzt. Mithilfe der algorithmic skeleton Bibliothek **Skandium** wird das entworfene Modell umgesetzt und in einem experimentellen Umfeld evaluiert. Die Evaluation untersucht die Eigenschaften der Parallelisierungsmöglichkeiten für vier verschiedene Variationen: der originale k-Means Algorithmus, eine kd-tree basierte Version, der k-median sowie der fuzzy c-means Algorithmus.

Contents

List of Figures

List of Algorithms

1. Introduction

1.1. Motivation

In the past decades the computing industry was driven by fast improvements of the underlying hardware technology. This regular growth, well known as *Moore's law* was traditionally accompanied by a doubling of the CPUs clock frequency [18]. For application developers this meant a reliable improvement of sequential code runtime over time. In 2004 however clock speed stalled due to physical constraints and the manufacturers instead began building CPUs with multiple execution cores [93]. Researchers expect this trend to continue leading to hardware systems comprising tenths to hundreds of integrated cores [6]. To utilize these cores efficiently, programs that are able to perform concurrent computations in parallel are needed. Another recent development in the field of information technology are ever increasing amounts of data, both stored and processed [98]. To deal with this development and to keep systems using these data amount in acceptable time frames, parallel processing is necessary.

1.1.1. Algorithmic Skeletons as a Model of Structured Parallelism

There are many models available to enable the programming of concurrent systems, providing a wide range of levels of abstraction. The subsequently featured models are representatives for the low and the high end of the scale.

The Thread Model (from *thread of execution*) is a widely adopted concept for modeling of concurrent execution. Threads provide an interface to create and manage lightweight sub-processes in a program. In contrast to operating system processes, the user handles creation and management of the threads [13]. Thread based approaches can be divided into two categories: in the model of Hoare/Dijkstra [47, 29] threads are seen as **Communicating Sequential Processes**, e.g. communication between them has to be explicitly modeled. The advantage of this model is that *race-conditions* (e.g. concurrent accesses of shared data) cannot occur. The disadvantage is that modeling communications between threads tends to grow very fast in complexity especially when greater amounts of threads are involved [40].
The other model (developed by Wyllie/Vishkin [89, 38]) bases the communication of threads on **shared memory** and explicitly allows concurrent access. Both approaches share a low level of abstraction: the programmer is explicitly tasked with modeling concurrency. On the one hand this allows fine grained access and tuning of potential systems, on the other hand communication and coordination has to be managed manually. This fact bears great error potential and often results in considerable complexity especially for large systems [69, 54]. For this reason the thread-model is often compared to the *GOTO*-Instruction in assembly languages: the amount of potential interactions between threads and with globally shared memory segments make it difficult to isolate error sources or reason about side effects in a parallel system [77, 40].

Functional Programming On the other side of the spectrum, providing a high level of abstraction, there are languages based on the **lambda calculus**. Programs are expressed as a set of functions, with no notion of explicit state or concurrency. The absence of state is fulfilled via the concept of *pure functions*, meaning functions must not have side-effects on any components of the program and always return the same corresponding result value when invoked with a specific argument. A functional program is expressed in nested function calls, which can be represented as a graph. The process of evaluating this graph and executing the correct functions is called *graph reduction.* For a parallel execution of the program, execution graphs can be partitioned and executions assigned to different processors. The advantages of functional programming in respect to parallelization are that data races cannot occur in a programming model which provides function application on immutable data. Furthermore, due to the absence of side-effects, race conditions or different results due to multiple invocations of a function cannot occur. The disadvantages of this model are strongly linked to the implicit parallelism: The following example taken from [22, p. 10] shows that the degree of parallelism inherent to a functional program depends on the function structure: In the naive implementation the $factorial$ is implemented as a recursive function, leading to a sequential execution graph

$$\begin{aligned} factorial\ 0 =& 1 \\ factorial\ n =& n * factorial(n-1) \end{aligned}$$

By reformulating the problem to:

$$\begin{aligned} factorial\ 0 =& 1 \\ factorial\ n =& product\ 1\ n \\ product\ a\ a =& a \\ product\ a\ b =& (product\ a \lfloor \frac{a+b}{2} \rfloor) * (product(\lfloor \frac{a+b}{2} \rfloor + 1) b) \end{aligned}$$

We get an execution graph with multiple branches which can possibly be executed in parallel. This shows the disadvantage that the problem formulation language has no concept of explicit parallelism. In order to write programs that efficiently use parallel systems, the programmer still has to know about the runtime parallelization techniques used. Another problem of parallelism in functional languages is the distribution of work to available nodes [22].

Structured Parallelism To find a trade-off between these abstraction levels, researchers developed the method of structured parallelism. Similar to the principles of *structured programming*, structured parallelism tries to find *patterns* that can be used to formulate well defined parallel execution [78]. The idea of structured parallelism is to divide the problem space into two layers: In the **productivity layer** the patterns are combined by programmers to achieve concurrency and parallelism in domain specific application areas in reasonable amounts of time. The patterns are implemented in what is called the **efficiency layer**: experts for parallel programming create the synchronization and communication needed for a specific pattern preferably optimized to a specific hardware layout [6]. **Algorithmic skeletons** are a model for structured parallelism that tries to abstract patterns in the form of parameterizable higher order functions. These functions capture the communication and synchronization needed by a specific pattern (and thereby providing interfaces for the productivity layer). These skeletons provide abstract generic views on algorithms, which are specialized by the supply of *computation functions*, that are inserted into the skeletal outline. The computation functions used by a skeleton can be expressed in an imperative or object-oriented matter, thus enabling programmers with a background in these programming paradigms to build parallel programs while still relying on their expertise.

1.1.2. Application: Cluster Analysis

Both computing power and data volume have reached a point where its possible for enterprise users to apply theoretic insights and algorithms in the field of *machine learning* productively and economically feasible. This fact has lead to a rise in applications using machine learning techniques and applying formerly theoretic algorithms in practice. To process the amount of data currently used in enterprise applications we need parallel applications that are able to process data quickly. Clustering is an essential task for applications in fields like machine learning or data mining. One of the most used clustering algorithms is k-Means [97]. By providing an algorithmic skeleton for this algorithm we can reduce the runtime, leveraging the benefits of skeletal programming (namely the abstraction of parallelism and synchronization) while simultaneously enabling users with domain specific knowledge to adapt implementation details quick and easy.

1.2. Research Goals

The goal of this thesis is to provide an algorithmic skeleton for the general k-Means algorithm. The skeleton then will be evaluated on shared-memory multi-core systems in regard to the following aspects:

parallelism: different parallelization possibilities extracted from the related work in the area.

drop-in improvements: the *kd-tree* based improvement of k-Means will be implemented and compared to the original version.

adaptability to show the application of the skeleton to different k-Means variations, the k-median and fuzzy c-means algorithm will be implemented using the skeleton and evaluated regarding their parallelization efficiency.

1.3. Outline

This thesis is structured as follows: The section **Background** (section 2) provides fundamentals and theory in the two fields used: **algorithmic skeletons** and **cluster analysis**. For each field terminology, background and features are explained. Next the chosen algorithm **k-Means** is introduced. Furthermore some variations are presented (namely the k-median, k-medoid and fuzzy c-means variants) In section 3 **Related Work** is discussed: different models and approaches for handling parallelism and concurrency are presented briefly. Afterwards an overview of optimizations and parallelization approaches for the k-Means-algorithm is given and the different methods are compared to each other. Based on this survey, the different parallelization possibilities are listed and used for the **Definition of the General k-Means Skeleton** (section 4) based on which different parallelization schemes are derived. To evaluate the skeleton design it is realized using an algorithmic skeleton library called *Skandium*. The section **Realization** 5 explains characteristics and execution semantics of the library. An empirical analysis of the realization of the different schemes is given in the Section **Evaluation** (6) along with a discussion of the obtained results.

2. Background

In the subsequent section an overview is given over the theoretical foundations of both **cluster analysis** and **algorithmic skeletons**. Key concepts are listed and terminology is defined.

2.1. Cluster Analysis

The field of cluster analysis concerns itself with the classification of data into different partitions. These partitions are called **cluster**. Informally, data elements belonging to a cluster share some kind of distinct feature. There is no widespread formal definition, but for the context of this thesis a **cluster** is defined as a set of objects which share an internal homogeneity (similarity) and an external separation (dissimilarity). This corresponds to definitions found in [99]. Ideally, the degree of dissimilarity between the subsets should be larger than the degree of similarity within each cluster [8].

Cluster Analysis originally developed in the field of anthropology and was then adopted by other fields like psychology or biology [88]. The task of organizing data into different groups is a fundamental human approach to comprehending all kinds of phenomena [99]. Taking this fact into account, it seems natural that cluster analysis is applied in many different areas, ranging from natural sciences (genotyping in biology, prediction of molecule properties in chemistry or classification of materials in geology) to sociology (analysis of connections in social networks), computer science (computer vision, multimedia processing, machine learning, data retrieval) and many more [99, 88, 31].

In cluster analysis the input data is normally represented as a set of **data elements**. Each of these elements (also called data points or observations) consists of multiple **features** defining the element. These features are normally represented as a multidimensional vector. They can be quantitative or qualitative, continuous or binary, nominal or ordinal, which determines the usable measures to compare them [99].

Due to the wide range of application areas, many approaches to the automation of cluster analysis with the help of *clustering algorithms* exist today. The coarse-grained classification of the approaches is based on the amount of prior knowledge about the data:

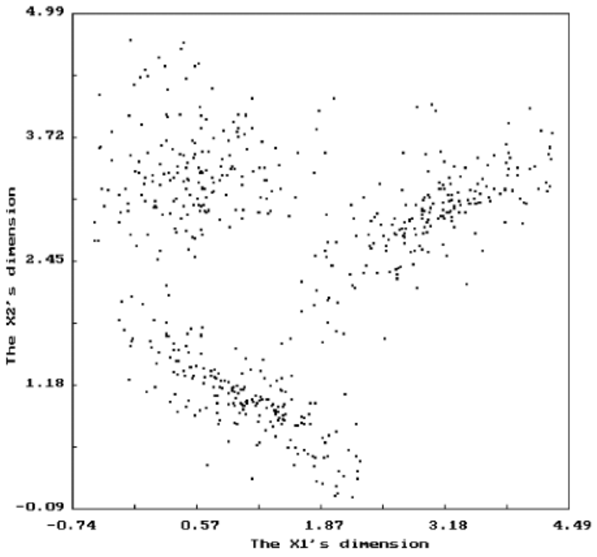
Statistical Classification If the task is to assign new data elements to a set of already known clusters, it is called statistical classification. The goal is to take a set of *labeled data* (meaning the clusters are known a priori and every data element is assigned to one of the clusters) and, for a new element, to decide to which cluster to assign it to. Every element should be assigned to a cluster so that both the similarity within this cluster (intra-cluster-similarity) and the dissimilarity between clusters (inter-cluster-dissimilarity) remain high. In machine learning terminology this is called *supervised learning* [99].

Clustering In contrast to the supervised approach the starting situation for unsupervised clustering is that there is no prior knowledge about the data set. The overall goal is to gain knowledge about the data (as a first step of information retrieval) [99]. The goal for the algorithms is to divide the data into subsets that share some similarity. As clustering is an unsupervised machine learning technique it provides a tool for autonomous systems to extract information for further decision making [99]. Similarly, in the field of data-mining, researchers presented with a visualization of found clusters can detect patterns or make further assumptions about the data more easy. Unsupervised clustering can be further divided into the *hierarchical* and *partitional* approaches:

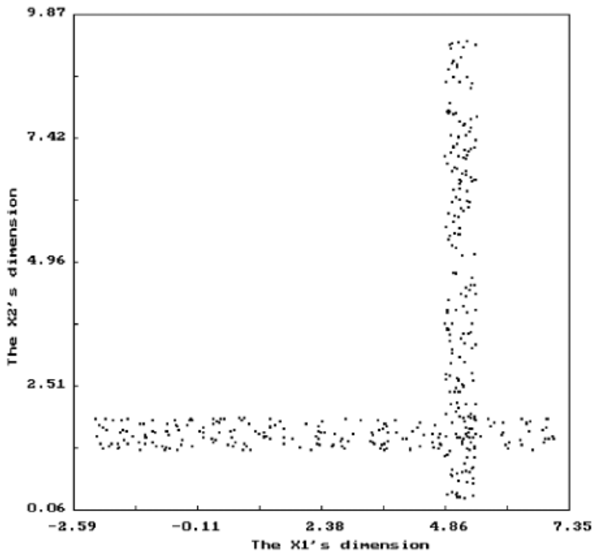
Hierarchical Clustering Hierarchical clustering is an approach similar to biological systematics: starting at one end (top-down: the first cluster is the whole dataset or bottom-up: the starting points are individual elements) the whole data set is divided into a hierarchical structure (like trees or dendrograms) with unions of clusters aggregating the contents of their children [53, 88].

Partitional Clustering In contrast the goal of partitional clustering is to find a fixed number of clusters (usually denoted k) without hierarchical structure in a dataset. The clusters are defined as disjoint subsets of the original set. Formally, given a set of input data $X = x_1, \ldots, x_j, \ldots, x_n$ partitional clustering seeks k partitions of $X, C = C_1, \ldots, C_k (k \leq n)$ such that

1. $C_i \neq \emptyset, i = 1, \ldots, k$
2. $\bigcup_{i=1}^{k} C_i = X$
3. $C_i \cap C_j = \emptyset, i, j = 1, \ldots, k$ and $i \neq j$



(a) data that fits to centroid based clustering



(b) data that fits to density based clustering

Figure 1: Examples for different data groupings (taken from [53])

Cluster Representation The similarity criterion used in the preceding cluster definition can be interpreted in many ways. This leads to various ways to represent clusters. The simplest way to represent a cluster is the raw grouping information: each element is assigned to a cluster. To extract concepts, generalizations or data aggregations, the clusters have to be represented by some kind of model [34]. One feature of similarity is the density of a region of point. Cluster models based on this feature are called **density-based** [82]. Figure 1b depicts a sample for clusters depending on density. Another way to represent clusters is to create a model expressed as functions and to interpret the given data as the output of one or multiple statistical distribution (an example for this cluster representation is the Expectation Maximization Algorithm which interprets data as output of multiple Gaussian normal distributions). In prototype or **center-based** algorithms the clusters are objects which are more similar to a prototype that defines the cluster than to the other existent prototypes. If the input data is in the form of a graph, clusters can be represented by **connected components** that are differentiated from the rest of the graph.

Similarity Measures To determine clusters which share internal similarity, a notion of a similarity measure between data elements is needed. The chosen concept for similarity determines the clustering results and often depends on the features of the dataset. One natural way to express both similarity and dissimilarity is to determine the **distance** of two points. A function that expresses a distance between two elements is called a metric (examples for possible distance metrics are shown in section 2.2.2). The requirements for proximity metrics/measures are [99]:

- Symmetry: $D(x_i, x_j) = D(x_j, x_i)$
- Reflexivity: $D(x_i, x_j) = 0$ if and only if $x_i = x_j$
- Positivity: $D(x_i, x_j) \geq 0 \forall xi, xj$
- Triangle inequality: $D(x_i, x_j) \leq D(x_i, x_k) + D(x_k, x_j)$ for all x_i, x_j and x_k

Cluster Configuration Quality To determine the result **quality** of partitional clustering, a function is needed to express the quality for a given clustering configuration. This function is often called **objective function** or criterion function. Like the cluster model and the similarity function the objective function influences the result clusterings and relies on knowledge about the data: clustering results considered good in respect to one criterion are not necessarily valid in respect to the (unknown) source groupings.

2.2. k-Means

The k-means problem uses a center-based approach to partitional clustering: each cluster in the data is represented by a **centroid**. The data elements are assigned to a cluster if they are “near” to it (in respect to a chosen proximity measure). The number of clusters k is normally a parameter. Options to determine a suitable value for this number can be found in section 3.2.4.

Problem Definition The problem is to determine a set of k points (called centroids) in the data space so as to minimize the mean distance from each data point to its nearest centroid. The goal is to partition data into k clusters so that the similarity between the points within a cluster is maximized. This means the quality of one cluster S_i (intra-cluster similarity) can be measured by summing up the distances from all points in the cluster to the centroid (c_i):

$$\sum_{x \in S_i} d(x, c_i)$$

To measure the quality of the overall cluster configuration (given k clusters) the distances of all clusters are summed up as well:

$$\sum_{i=1}^{k} \sum_{x \in S_i} d(x, c_i)$$

this leads to the subsequent target function (also called **objective function**) [31]:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} d(x, c_i)$$

For the input space $\mathbb{R}^d$ the most commonly used distance measure is the **squared euclidean distance** which leads to the following objective function:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{c}_i\|^2$$

To find the global optimal clustering, the naive way is to iterate through all possible cluster configurations and calculate the objective function value. There are k^n ways to partition a set of n points into k possibly-empty clusters. If we only consider non-empty cluster the number of possibilities is [31]:

$$\frac{1}{k!} * \sum_{j=0}^{k} (-1^{k-j} \binom{k}{j}) j^n$$

This results in configuration amounts not feasible for simple enumeration of all possibilities: "for example, if 25 objects are to be grouped into 4 clusters, there are approximately 4.69×10^{13} different partitions" [90].

Furthermore, research has shown that the problem of finding the cluster configuration that minimizes the global squared error is NP-complete [3, 27, 76, 64]. This leads to a heuristic, iterative *hill climbing* approach, which means starting out with one partition configuration and an objective function to determine the quality of the current clustering. Subsequently, the cluster configuration is adapted iteratively (by moving data elements between the subsets), so that the objective function value is reduced with each step. This leads to a local optimum clustering in acceptable time [53].

2.2.1. Lloyds Algorithm

The most widespread algorithm solving the k-Means problem heuristically is the version described in algorithm 1 which was proposed by Lloyd [74] as a vector quantization method and therefore is often called Lloyds algorithm. A very similar algorithm was proposed by Forgy [37] which is why the algorithm is sometimes referred to as Lloyd-Forgy algorithm. The term *k-Means* itself was first used by MacQueen [75] to describe a model representation for the partitions of a dataset having high within-cluster similarity.

The algorithm solves the problem of unsupervised clustering in the $\mathbb{R}^d$ space. Due to the simplicity and efficiency k-Means is one of the most used algorithms in the field of data mining, being used in a wide area of applications such as image processing, pattern classification, computer vision or machine learning [31, 97].

Algorithm 1: Lloyd-Forgy k-Means

```
  Data: data set: $x_1, \dots, x_n$ with $x_i \in \mathbb{R}^d$
  Data: Initial set of means: $c_1, \dots, c_k$ with $c_i \in \mathbb{R}^d$ and $c_i$ chosen randomly from the
        data set
  Result: local optimum for the cluster centers: $c_1, \dots, c_k$
1 while assignments changed between the iterations do
2     Assignment Step: $S_i^{(t)} = \left\{x_p : \left\|x_p - c_i^{(t)}\right\|^2 \le \left\|x_p - c_j^{(t)}\right\|^2 \ \forall j, 1 \le j \le k\right\}$,
3     Update Step: $c_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$
4 end
```

Informally the algorithm iterates between two steps: In the **Assignment Step** the *squared euclidean distance* to all cluster centers is calculated for every point in the given data set. The squared distance is used to avoid the computational intensive square root. The point is then assigned to the cluster center m_i that has the minimal distance to the point. After assigning all points to a center the **Update Step** recalculates the cluster centers: The mean of all points assigned to the i^{th} cluster is calculated (in every dimension) and the center for the next iteration m_i is set to the result value. These two steps are repeated until the assignments of the data points does not change between the iterations. Figure 2 shows a visualization of the algorithm.

The version presented by MacQueen [75] differs slightly: cluster-centroids are moved after each assignment of a data element instead of being recalculated in batches after each assignment iteration. This variation is sometimes called *h*-means [91] and was further investigated by Hartigan et al. [46].

Complexity The algorithm has to keep track of the cluster centroids and the data elements used. Therefore the **space complexity** of the algorithm is bound by the

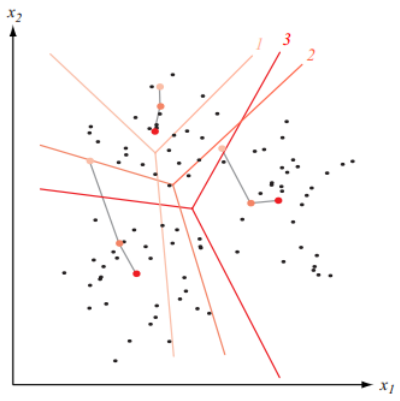


Figure 2: visualization of the Llyod/Forgy algorithm (d=2,k=3,i=3) [32, p. 617]

number of data elements n and the number of cluster centroids k each with a given dimension d [103]:

$$O((n+k)d)$$

The theoretical worst case **runtime complexity** is determined by the number of possible clustering configurations (the number of possible voronoi cells): Inaba et al. [51] found that this is in $O(n^k d)$. Arthur et al. [4] proved that k-Means has polynomial smoothed complexity, which means that although the worst-case behavior is in $O(n^k d)$ the practical runtime is normally polynomial [31]. The average runtime of the algorithm depends on the number of elements in the input data n, the dimensionality d, the number of cluster centers k and the number of iterations until convergence i, therefore:

$$O(nkdi)$$

Kucukyilmaz [64] divides the algorithm into the following steps:

1. Centroid Initialization
2. Distance Calculation: nk distance calculations are needed, each of which needs $2d$ floating point operations (flops).
3. Minimum: to get the cluster centroid with the minimal distance, nk flops are needed.
4. Recalculation of the *mean squared error* (used to determine convergence): kd flops
5. Recalculation of the cluster centroids: nd flops

Given T iterations and a duration of T_{flop} per floating point operation this leads to the following k-Means runtime:

$$(2nkd + nk + nd + kd) \times T \times T_{flop} \tag{1}$$

Drawbacks The k-Means algorithm has several drawbacks. The variations presented in the next section try to overcome some of them. First of all, the value of k needed to execute the algorithm has to be determined. Choosing the right number of clusters remains a difficult task (although there are various proposed method of deriving this value, see 3.2.4). Furthermore the result of the algorithm is a **local** minimum dependent on the initial centroid configuration (seeding) and has no theoretical guarantees concerning the global optimum (result clusters may be arbitrarily far from the optimal clustering). Due to the centroid representation as the mean of all cluster members and the clustering of all datapoints the algorithm is **sensible to outliers** and noise. Due to the choice of similarity measure (euclidean distance) and the recalculation method (means) the input data is **limited to numerical values**. The last point is that the resulting clusters are in the forms of **hyperspheres** (due to the centroid representation). This means natural clusters in the input data with another shape will not be found by the algorithm [99].

2.2.2. Variations

Due to its popularity the k-Means algorithm is subject to many proposed variations aiming at different stages. In the next section, different variations are summarized and analyzed in respect to their shared features, in order to establish a general k-Means approach, used for the algorithmic skeleton. The biggest group of variations is the choice of proximity metrics other than the euclidean distance used by Lloyds algorithm.

Distance Metric As illustrated in the cluster analysis background (section 2.1) the chosen distance metric determines the clustering results substantially. Due to the representation of data elements as d-dimensional vectors, metrics valid for a d-dimensional space can be used. For example the **Manhattan distance** (used in k-median, see 2.2.3) can be used as distance measure. The generalization of these metrics is called **Minkowski distance** which is defined between two Elements X and Y as:

$$(\sum_{i=1}^{n}|x_i - y_i|^p)^{1/p}$$

where the before mentioned metrics are the special case for $p = 1$ (Manhattan) and $p = 2$ (Euclidean).The advantage of using the generalized Minkowski distance is that it allows for other cluster shapes depending on the value of p [31]. Taking into account correction for scale or distribution of data, the **Mahalanobis distance** could be used, which augments the euclidean distance with a covariance matrix.

There are multiple ways to account for data not in $\mathbb{R}^d$: for composed binary features (e.g. bit patterns) a distance measure based on the equality of the features' parts could be applied (for example the **Hamming distance**), for nominal features a mean matching criterion. Furthermore the distance of two elements can be expressed in a *similarity matrix*, where the element m_{ij} represents the distance between the i^{th} and the j^{th} element of the input data [99].

To account for symmetry features in the data instead of locality (as with Minkowski distance) a distance measure that treats symmetric elements as similar can be used. This approach is used by Su et al. for a symmetry based k-Means method [92] which uses point symmetry distance (between a point x_j and a centroid c_j):

$$d_s(x_j, c) = \min_{j=1,\dots,N, j\neq i} \frac{||(x_i - x_j) + (x_j - x_i)||}{||(x_i - x_r)|| + ||(x_j - x_i)||}$$

With this measure, clusters with symmetric features are found. The point symmetry distance is non-metric, meaning the limitations listed in section 2.1 do not apply.

Algorithmic Variations The **k-medoid** method addresses the problem of spaces where the notion of a mean of several points is not defined (for example for nominal features). Instead of using the centroids defined by the mean of all points in a cluster the k-medoid method uses the point with the minimal distance to all cluster members. The distances can be expressed by a nominal metric (e.g. an equality metric on every dimension of the compared points) or as a given distance table (with a fixed distance value for every point combination). The most popular algorithm addressing the k-medoid problem is **partitioning around medoids** [62]. Park et al. [83] show that the k-medoids problem can be solved using a k-means-like approach of iteratively reassigning medoids for each cluster.

The **fuzzy c-means** algorithm [86] transfers the k-Means approach into the realm of *fuzzy clustering*. A detailed description is given in 2.2.4.

In [95] Wagstaff et al. show an algorithmic variation that incorporates domain knowledge about relations between data elements in the k-Means algorithm. To achieve the integration, the assignment step is adapted so that during the assignment of a point the constraints are checked for the proposed cluster (an example for a constraint is that two points cannot exist in the same cluster). If the assignment fails the second nearest cluster is chosen continuing through the clusters.

Empty Cluster Handling A feature not specified by the original k-Means algorithm is the handling of centroids with no data-element assigned during the *assignment step*. This case can be handled in various ways: The easiest is to abort the algorithm execution and report an **error**. The center without assignments can be **removed** from the list of centroids for the next iteration. In this case final results contains only $k - 1$ centroids. Alternatively it can be set to a **new randomly chosen location** and passed to the next iteration. Furthermore, Muhr and Granitzer proposed an algorithm that **merges** multiple clusters with to few assigned points into a new cluster [81] (see also section 3.2.4) Another way to circumvent the existence of empty clusters is a method called *constrained k-means* proposed by Bradley et al. in [16]: The constraint that clusters should not contain fewer than τ points is integrated in the assignment step. After the integration the problem is solved by showing it's equivalence to the Minimum Cost Flow (MCF) linear network optimization problem.

Convergence Criteria The convergence criterion depends on the chosen distance metric and recalculation method: It defines a threshold for the optimizations carried out by assignment and update steps. In Lloyds algorithm the criterion was defined as the stabilization of clusters: when no points are assigned to two different clusters between two iterations, the algorithm has converged to a local optimum. This criterion can be converted to the comparison of the **sum of squared errors** (if $SSE_i = SSE_{i+1}$) because it is the objective function derived from the assignment/update combination of the algorithm. In [32] Duda et al. showed how to derive other criterion functions from multiclass discriminant analysis. For fuzzy variations often a **threshold** ε is chosen to indicate convergence. To limit runtime of the algorithm another method is to limit the **iterations** of the algorithm to a fixed value.

2.2.3. k-Median

The k-Median algorithm tries to improve the quality of the results by changing the distance measure and the recalculation method: Instead of the squared euclidean error the taxicab-geometry is used. During the **Update Step** the clusters are recalculated by using the median instead of the mean [53].

2.2.4. Fuzzy c-Means

Fuzzy c-Means is an enhancement of Lloyds algorithm: Instead of assigning every data point to exactly one cluster center, the algorithm uses assignment probabilities which are calculated for every point and all cluster centroids [33, 12]. This is often referred to as "soft k-Means" to differentiate from the "hard" assignment of every point to exactly one cluster. The objective function used in fuzzy c-means is

$$J = \sum_{i=1}^{n} \sum_{j=1}^{c} (u_{ij})^m d$$

where d is a distance measure (normally squared euclidean distance). $m \in \mathbb{R}$ with $m \geq 1$ is the *fuzziness index*: for $m = 1$ the results converge towards the results of Lloyds algorithm. u_{ij} represents the membership degree between the i^{th} data element and the j^{th} cluster (with $u_{ij} \in [0,1] \forall i,j$). To iteratively optimize the clustering configuration, the steps of algorithm 2 are repeated until the maximum delta of the centroids between two iterations falls below a predefined threshold: $\max_{ij}\{|u_{ij}^{(k+1)} - u_{ij}^{(k)}|\} < \epsilon$.

The fuzzy c-Means algorithm can be interpreted as an intermediate step toward a **expectation maximization** algorithm using gaussian mixture models to describe the clusters [86, 842 ff].

In [106] Zhi et al. show that the **k-harmonic Means** proposed by [103] is a special case of fuzzy k-means clustering, with the *fuzziness index* set to $m = 2$. The fuzzy c-Means algorithm is subject to variations as well. A generalized model for fuzzy c-Means as well as a discussion of variations can be found in the work of Yu et al. [100].

Algorithm 2: The fuzzy c-Means algorithm

```
  Data: data set: $x_1, \dots, x_n$ with $x_i \in \mathbb{R}^d$
  Data: Membership matrix $[u_{ij}]$ initialized randomly
  Result: local optimum for the cluster centers: $c_1, \dots, c_C$
1 while error greater than $\varepsilon$ do
2     recalculate centroids: $c_j = \frac{\sum_{i=1}^{n} u_{ij}^m \cdot x_i}{\sum_{i=1}^{n} u_{ij}^m}$
3     update membership matrix: $u_{ij} = \frac{1}{\sum_{k=1}^{C} \left(\frac{\|x_i - c_j\|}{\|x_i - c_k\|}\right)^{\frac{2}{m-1}}}$
4 end
```

2.3. Generalized k-Means

This generalized approach is formalized in algorithm 3. The two abstract steps can be instantiated by choosing a proximity measure and a objective function (for an example see 2.2.1). Due to the fact that the k-Means approach was proposed in Lloyds algorithm the version depicted in algorithm 3 is sometimes called **generalized Lloyd algorithm** [67].

Algorithm 3: Generalized k-Means approach [82]

```
1 Select $k$ points as initial centroids.
2 repeat
3     Form $k$ clusters by assigning each point to the centroid minimizing the distance
      measure.
4     Recompute the centroid of each cluster.
5 until convergence Criterion is reached;
```

2.4. Algorithmic Skeletons

Algorithmic skeletons provide a means to express **structured parallelism** in programs. The subsequent chapter provides an overview over the history, the original objectives, the basic principles and the advantages of this technology. In addition common algorithmic skeletons are listed and explained.

2.4.1. History

The idea of algorithmic skeletons goes back to the work of Muray Cole [24] in 1989 who proposed it as a programming model that enables the user to express parallelism in a structured, concise way. Cole evaluated the trade-off between the clarity and portability of a program and the performance gains by leveraging specific hardware and combining techniques from functional and imperative programming to express the *skeletal structure* of parallel algorithms (see 2.4.2). In the following years the field was further developed and enhanced by various universities. Implementations of the idea for most of the common programming languages were provided (an overview of algorithmic skeleton libraries is provided in [40]). During the last two decades the algorithmic skeleton model was enhanced by providing a way to nest skeletons [44], proposing sensible restrictions for the structure and implementation of the model [23] and the development of *common algorithmic skeletons*: general use patterns for parallel programming [40]. Another research area is the optimization of skeleton configurations via automatic tuning [19]. Despite the improvements and implementations, the idea of algorithmic skeletons was never adopted in commercial settings, although various techniques bare resemblance to the concept (for example the map/reduce approach [40]).

2.4.2. Basic Principles

To explain the basic principles of algorithmic skeletons the example of the *map* function is chosen. Informally *map* applies a function to all elements of a collection of data elements (for a graphical representation see 3). This can be expressed by the use of higher order functions, a technique from the functional programming environment, defining functions with functions as a parameter. Given a function in the form of [22]:

$$square\ x = x * x$$

with a type

$$square : int \rightarrow int$$

We can use the aforementioned *map* which applies the function to all elements of a given collection to express the following:

$$(map\ square) : [int] \rightarrow [int]$$

By abstracting the **square** function to an arbitrary function with the type $f : a \to b$ we can obtain a generalized *map* function, like follows:

$$\text{map} : (a \to b) \to ([a] \to [b])$$

Next the notion of a *split* and a *merge* function is introduced to provide arbitrary segmentation of a data collection. The types of these functions are:

$$\text{split } f_s : a \to [a]$$

$$\text{merge } f_m : [b] \to b$$

This leads to the following definition of a **map-skeleton**:

$$\triangle_{map}(f_s, f, f_m) : (([a] \to a) \to (a \to b) \to (b \to [b])) \to ([a] \to [b])$$

A programmer that wants to use this skeleton now has to provide implementations for the 3 functions (split, apply, merge). One key element of the original algorithmic skeleton approach is that these three functions are provided as **sequential** imperative code. This eliminates the need for the programmer to adopt the functional programming style and use already existent skills instead.

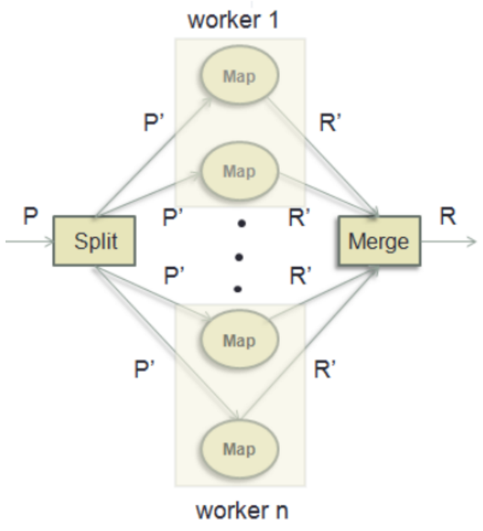


Figure 3: Visualization of the *map*-skeleton [7]

2.4.3. Goals

The goal of algorithmic skeleton is to provide an abstraction level that on the one hand enables **explicit expression of parallelism** (in contrast to functional languages) and on the other hand hides details about the communication and synchronization needed to achieve parallelism (in contrast to basic message passing libraries). By making the skeletons *parameterizable* through the provision of sequential functions, the model provides **flexibility** and encourages **re-usage** of the skeleton patterns. Another goal was to enable **scalability**: The interface of a skeletal system doesn't grow in complexity when adding more processing nodes. Therefore the user is enabled to build applications for many-core systems without error prone manual management of concurrency. Although algorithmic skeletons have been implemented in functional languages, the core idea of Cole was to provide a means to **parallelize imperative languages**.

2.4.4. Advantages

An algorithmic skeleton separates the *computation* (which result is computed) from the *coordination* (how the result is computed). This provides multiple advantages:

- clarity: The separation enables a user interface that can be expressed in sequential functions. This improves reasoning about the actual computations the program executes. Code concerning synchronization and communication between processes is separated from the business logic (separation of concerns).
- portability: When the application is expressed as an interaction of skeletons, underlying hardware changes can be abstracted away. This enables portability: A given skeletal program can be executed on a given system, as long as the used skeletons are implemented for it.
- optimization potential: by abstracting away the communication, the implementation of a skeleton for a specific system can use the communication optimizations provided by this system
- adaptability: changing minor details about the business logic (e.g. variations of k-Means) leaves the overall structure of the program untouched.
- flexibility: Due to the missing order of function application in parallel skeletons, runtime optimizations can be done on different systems: An example would be to weigh parallelization overhead against performance gains and sequentialize execution ad hoc [2].

2.4.5. Common Skeletal Patterns

Since the initial proposal of Cole, many algorithmic skeletons have been proposed. The following chapter provides an overview over common skeletons, which are general purpose and can often be found implemented outside the skeleton research area (taken from [40]). These general purpose skeletons can be classified into three subsets:

Data-parallel skeletons enable parallel processing of multiple data elements concurrently and are normally used when dealing with a large amount of elements. The most used example from this class is the *map* skeleton shown in 2.4.2. It provides a way to execute the same function on multiple data elements. The *fork* skeleton works in a similar manner: a collection of data elements is split and functions are applied in parallel. The difference to the *map* skeleton is that with *fork* a different function can be applied to each data element. The *reduce* skeleton aggregates data into a single result by applying a provided *aggregation function* to element pairs from the input set.

Task-parallel skeletons operate on tasks rather than splitting data elements: they define which operations can be carried out concurrently and how different tasks interact with each other. The most used skeleton in this subset is the *pipe* skeleton: It describes the pipeline approach to parallelism also found in CPU pipelines: The processing of data is split into multiple stages which can be carried out concurrently. The data is passed on from one stage to another. *farm* describes the well known master/slave-worker pattern: a coordination task triggers different subtasks that carry out their assigned work concurrently. The skeleton equivalents of the classic control flow patterns are also part of the task-parallel skeletons: The *if* skeleton runs a skeleton only if a condition function evaluates to true, the *for* skeleton executes a skeleton for a predefined number of times and the *while* skeleton executes another skeleton while a condition function remains true.

Hybrid skeletons are the third class, which represents algorithm-based skeletons working with both data and task parallelism. The most common patterns in this class are description of two algorithms: first *divide-and-conquer* is a skeleton representation of the well known divide and conquer algorithm: An input problem is split by a divide function as long as a condition function remains true for the split input. The handling of the split results is done in parallel. When a point is reached where the condition is no longer true a sub-skeleton representing the conquer logic is applied to each subset. The divide and conquer approach is used in various approaches to structured parallelism (see section 3.1.4). For combinatorial optimization problems, the *branch and bound* algorithm can be expressed as a skeleton, taking a set of problem instances and a set of feasible solutions and applying a given objective function to determine result quality and pruning possibilities.

2.4.6. Limitations

Although algorithmic skeletons provide an elegant way to abstract away communication and synchronization patterns of parallel computation, they do not solve problems inherent to the performance evaluation of parallel algorithm. Implementations of skeletons are still subject to performance affecting problems as contention on synchronization objects, load imbalance, overhead, false sharing, inefficient memory access patterns or I/O being the bottleneck [78].

3. Related Work

Both fields used in this thesis received great attention over the last decades. Since the rise of multi-processor architectures in the common desktop and server computers, parallel programming is subject to a increasingly large number of improvement approaches and research topics. Similarly due to the increasing popularity of machine learning, many papers investigate possibilities of optimizing the k-Means algorithm and adapting it to newly emerging computing environments. This chapter gives an overview over the essential developments in both fields.

3.1. Structured Parallelism

Structured parallelism tries to reduce the possibility for errors and improve the re-usability of parallel solutions by introducing patterns/structures similar to the way **structured programming** did to sequential code. Due to the stalling of processor clock speed, models for structured parallelism have seen great attention in the past decade. Because of this an exhaustive study of approaches to parallel programming is not feasible in the given context. Instead this study shows a selection of programming models representative for trends in the field. The focus of this section is on models in the realm of imperative programming. Approaches not based on this paradigm are listed in section 3.1.5.

Terminology When talking about parallel programming, it is important to differentiate between two concepts related to the field. If a system is able to switch execution context and therefore switch between multiple tasks in progress, it is called **concurrent**. A **parallel** system however can independently execute different tasks *at the same time* [17]. This becomes more clear by looking at the thread abstraction used in many programming languages: Although the programmer is able to express the possibility to execute parts of the program concurrently, for a system with only one processing unit the code is executed sequentially.

3.1.1. HPC Based Approaches

The first class of structured parallel programming originated in the field of high performance computing (HPC) and often provides a high number of fine tuning options. The first approach uses compile directives (**directive based**) to express the parallelization of sequential code blocks. Representative techniques are for example *OpenMP* or *OpenACC* [25]. The aim of this approach is to parallelize iterations over data in a structured manner (for example a *for* loop). The other approach uses an abstract model to enable accelerators (like GPUs or Co-processors). Languages designed to program such highly parallel systems (e.g. *OpenCL* or *CUDA*) use a geometric model to express thread relations.

3.1.2. Task Based Approaches

To abstract the usage of threads, one approach is to define a notion of *work* that is executed. This work is generated dynamically while executing a program (for example using the fork/join model) and then mapped to the available level of parallelism on a given machine (load balancing). The scheduling of the work can be done by an **executor** component which schedules work onto a processing unit, or done in a decentralized manner. One work balancing approach is **work stealing** where every processing unit is assigned a queue of work items to execute. When this queue is empty for a processor, it takes work from the queues of one of the other processors. The work stealing approach is employed in many of the popular languages(e.g. the fork/join framework in java, the TBB scheduler, or the Task Parallel Library (TPL)) [79].

3.1.3. Data Driven Approaches

The data driven approaches are related to the field of *dataflow* programming (see 3.1.5) which focuses on a purely functional approach backed by special purpose hardware. The subsequently presented approaches however model data transformations as libraries for sequential programming languages. The idea is to model the system as a series of computations on a stream of data (stream processing). One way is to model the graph explicitly either **graphically** or **textually** through the representation of nodes and edges as objects (e.g. Dryad [52] or the Thread Building Blocks Graph Flow, an approach to parallelization of C++ Code). The disadvantage of using graphic representation is that a mapping between the graph nodes and available computing capacities of a system is needed.

The second approach is to express the desired results as functions on data elements and then evaluating only the data transformations needed for the result (lazy evaluation). This approach is used in a data center computing context with a concept called **Resilient Distributed Datasets** [101]. The key aspect of data driven approaches is a more **declarative** approach: instead of describing how the results are computed the user of the technique describes what desired transformation should be applied to the data. This abstracts away the specific implementation which leaves room for parallelization. Another way is taking data structures familiar in the programming community (for example a vector) and designing libraries that provide the same interface as the original data structure but internally handle the communication and synchronization needed when a concurrent access occurs. This approach is called **concurrent data structures** [80].

3.1.4. Parallel Patterns

In the last decades, different ways to handle parallelization of programs emerged. The field of **parallel patterns** tries to extract programming design patterns specially suited for parallelization. This shows close resemblance with algorithmic skeletons: They can be viewed as a **formal notation** for parallel patterns. This is the main difference between the two approaches: Parallel patterns are implemented in programming libraries (e.g.

Thread Building Blocks' Parallel Patterns for C++ [78]), while the skeletons express an abstract mathematical way for the pattern interface. Naturally the commonly used design patterns overlap with the common general purpose skeletons shown in section 2.4.5. An example for a parallel design pattern is the *map* pattern taken from [77]:

> "The map parallel computation pattern applies a function to every element of a collection (or set of collections with the same shape), and creates a new collection (or set of collections) with the results from the function invocations. The order of execution of the function invocations is not specified, which allows for parallel execution. If the functions are pure functions with no side effects, then the map operation is deterministic while succinctly allowing the specification of a large amount of parallelism." [77]

The research areas of parallel patterns and algorithmic skeletons overlap: In the subsequently presented ParaPhrase project skeletons are used as the implementation tool for common parallel patterns [45]. Other commonly used parallel patterns include *reduce* (combining elements of a set in parallel), *pipeline* or the *workpile* pattern which corresponds to the *divide and conquer* skeleton. The **fork/join model** is a generalization of the divide and conquer algorithm. Examples for implementations are the Java Fork/Join Pool or CILK's spawn and sync directives.

ParaPhrase: The ParaPhrase Project (started in 2011) tries a holistic approach to the field of parallelization. It aims to provide parallelism through the refactoring of programs using high-level design patterns (implemented with the algorithmic skeleton approach). Furthermore, the parallelization is optimized via mapping of available hardware resources during execution. Its target are heterogeneous many-core architectures. During the project, skeleton libraries were developed for both erlang (Skel) and C++ (FastFlow) [45].

3.1.5. Other Approaches

Dataflow Programming originated as an alternative to the *von Neumann* model: In contrast to the static nature of data in the model, the dataflow model proposes a computational structure, building on local memory and execution of instructions as soon as their operands are available. A program for this model is described as a directed graph where data is send from one computation node to another. The model introduces corresponding hardware (dataflow hardware architecture) [57].

Parallelization of Functional Languages: To harness the implicit parallelism in functional languages given by independently evaluated functions, **distributed execution** analyses the distribution possibilities of execution graphs [50, 26]. For the explicit expression of parallel execution in functional language, **language extensions** like concurrent Haskell or MutiLisp extend the languages with explicit directive for parallelism. These extensions enable the expression of the before mentioned models (task/data based) in a functional context.

Coordination Models and Languages propose programming languages dedicated to process coordination. These languages (such as *Linda* [39]) establish a virtual shared memory system in heterogenous networks and provide means to manage data and processes in this system.

3.2. k-Means optimizations

k-Means has been subject to various optimizations and enhancements. The focal point of this thesis are the different methods of parallelization of the algorithm, therefore sequential optimizations are only analyzed briefly. For further explanation of sequential k-Means optimizations see [31].

3.2.1. Exact Sequential Acceleration

The goal of exact sequential acceleration is to modify the algorithm so that the runtime is improved while the results stay the same. These methods are also called **drop-in optimizations**. The approaches can be divided into two subsets: First the algorithm can be sped up by using **spatial data structures**. The authors of [84] and [61] use a *kd-tree* to store the input data: In both cases a set of candidate centroids is maintained while traversing the data tree which is filtered for each passed level in the tree (representing subdivisions of the data space). A further explanation of the *filtering algorithm* is given subsequently.

The other approach is to use one of the features of distance metrics: the **triangle inequality** (see Section 2.1). Due to the subdivision of the input space and the geometric reasoning, the sequential optimizations are often sensible to the natural clustering of the data. When data uniformity is high, assignment of large amounts of points to one cluster centroid based on their spatial arrangement becomes unlikely. The triangle inequality based approaches are different in respect to the achievable speedup over the original k-Means algorithm given a dataset configuration (number of elements, dimensionality, number of cluster centers). In a comprehensive quantitative study Drake shows an overview of different speedup schemes and their suitability for different dimensionality and number of centers. The reported speedups (for the birch dataset) reach up to 38 times less execution time compared to the original k-Means algorithm [31].

The Filtering Algorithm The algorithm proposed in [61] uses a *kd-tree* [11] as the data structure for the data elements. The input space is subdivided by the nodes of the tree: The root node represents the whole input set. Each node represents a split of the input space into two subspaces. The split dimension is determined by iterating the available dimensions and splitting the space at the **median** coordinate of the input set for the current dimension. The node's children represent the points partitioned above and below the split coordinate. The leaf nodes represent each point in the input data set. The *filtering algorithm* leverages this partitioning to reason about cluster membership of the containing points: if a subspace of a node (denoted $u.cell$) is in its entirety further away from a centroid than from another, the further centroid can be *filtered* from the list of centroids the points in the subspace have to be compared to. The algorithm traverses the whole tree and for each node removes centroids from a candidate set using the logic outlined before. Algorithm 4 shows the logic used, given a Node u and a set of candidate centroids Z. The function $z.isFarther(z*, C)$ is used to remove centroids which are farther away from $z*$ than the whole cell described by C. The performance gains of the

Algorithm 4: The filtering algorithm [61]

```
Function Filter(kdNode u, CandidateSet Z)
    $C \leftarrow u.cell$ ;
    if u is a leaf then
        $z* \leftarrow$ the closest point in $Z$ to $u.point$ ;
        $z * .wgtCent \leftarrow z * .wgtCent + u.point$;
        $z * .count \leftarrow z * .count + 1$ ;
    else
        $z* \leftarrow$ the closest point in $Z$ to $C$'s midpoint ;
        foreach $z \in Z \setminus \{z*\}$ do
            if $z.isFarther(z*, C)$ then
                $Z \leftarrow Z \setminus \{z\}$;
            end
            if $\|Z\| = 1$ then
                $z * .wgtCent \leftarrow z * .wgtCent + u.wgtCent$ ;
                $z * .count \leftarrow z * .count + u.count$ ;
            else
                Filter($u.left, Z$);
                Filter($u.right, Z$);
            end
        end
    end
```

filtering algorithm are sensitive to the distribution of the input data elements. When a natural clustering is present in the input data, more centroids are filtered earlier in the tree traversal (because of a higher probability that the spatial decompositions of the input space match a natural clustering). Furthermore, it is shown that for higher dimensions the performance degrades [31].

3.2.2. Approximative Approaches

The idea of approximative approaches is to provide better runtime while deviating from the results of the original algorithm (preferably by a theroretical bound). For k-Means this often means choosing subsets of the input data. This can be done either by picking random data elements from the input set (sampling, as shown by the APKM algorithm in [63]) or based on knowledge about the data (core-sets) [60, 10]. A subcategory of the approximative approaches are *streaming* algorithm which try to cluster the data in a single pass. Furthermore, they do not require loading the complete set of data input elements into the main memory. Proposed algorithms are *single pass k-Means* [1] or ScaleKM [14].

Yinyang k-Means proposed by Ding et al. [30] combines the usage of the triangle inequality with a grouping approach. For each element-group upper and lower bounds to the centers are calculated to eliminate unnecessary distance calculations.

3.2.3. Centroid Initialization

Due to the iterative hill climbing approach of Lloyds algorithm, it's result is dependent on the initial clustering configuration (it finds a local optimum dependent on it). There are many different ways to pick the first set of centroids (this step is also referred to as **seeding**), with the easiest being a random pick either from the input data set or from the data space. This initialization however provides no guarantee about the final result. Furthermore, the choice of initial centroids has impact on the number of iterations until convergence is reached and on the quality of the final result [5].

k-Means++ The idea of k-Means++ is to choose the first centroid randomly and subsequent centroids so that they are far away from each other (the probability of choosing a point as a centroid rises proportionally to its distance to the existing centroids). With Lloyds algorithm no theoretically motivated statements can be made on the quality of the found minimum. k-Means++ can show that its $\Theta(\log k)$-competetive to the global optimum [31].

Scalable k-Means++ The k-Means++ algorithm has a time complexity of $\mathcal{O}(nkd)$ (equivalent of one sequential k-Means iteration). The process of k-Means++ can be parallelized as well. One technique to do so is described in [9].

Competitive Passes Another method of improving the quality of the result clustering is to run the algorithm with different seedings in parallel. **Mux-k-Means** uses a technique that connects multiple runs and compares their final error-values [72].

3.2.4. Number of Clusters (k)

A traditional shortcoming of k-Means is the assumption of a known number of clusters (k) in the dataset. The objective function optimized by Lloyds algorithm (within-cluster sum of squares) decreases monotonically with increasing k (down to the trivial clustering of $k = n$ where each point has its own cluster). Therefore other measures are needed to determine the validity of a chosen k [34].

To determine the number of clusters, some methods have been proposed: The first one is the manual review of validity through **visualization** of the data set. However this method only works if the given dataset can be projected into a 2-dimensional space.

Another method is the definition of a **stopping rule**: Xu et al. [99] mention that there are over 30 indexes proposed to determine the quality of a given k.

A different approach to this problem is proposed by Muhr and Granitzer [81]: During the execution of the algorithm defined thresholds are introduced, that provide logic when to **split and merge clusters** [81]. As pointed out by Estivill-Castro [34] the validity criterion functions could for themselves be used as clustering objectives.

x-Means tries to solve the problem of finding an appropriate number of clusters by starting multiple runs of k-Means with increasing values of k and examines the result quality by a given secondary criterion. There are multiple proposed information criteria, most notably **Akaike's information criterion (AIC)** or the **Bayesian inference criterion** [85]

3.3. Parallelization of k-Means

Due to the popularity of the k-Means algorithm, various approaches to parallelize the process exist. Subsequently some selected techniques are presented. The given approaches are analyzed on problem decomposition and distribution features. All examined sources assume that the number of input elements is sufficiently larger than the number of clusters ($n >> k$). Furthermore, most of the examined papers provide algorithms for a specific target platform (shared memory, NUMA, MapReduce). The goal of this thesis is to unify these approaches in a skeletal model of the algorithm. A majority of the analyzed sources leverages the inherent data-parallelism in k-Means: The input data elements are not dependent on each other, therefore distance measurement and assignment to the next centroid can be done in parallel. The common approach is to divide the input data into preferably uniform subsets and to perform the distance calculation on one chunk per processor. This approach will be referred to as **p-chunks** and is sketched out in algorithm 5. Subsequently the analyzed approaches to k-Means are classified in relation to their target system architecture.

3.3.1. Datacenter/Cluster Environments

The first category of targeted systems is the datacenter/cluster environment. The nodes in this environment are assumed to be under the operators' control and able to communicate with each other. This target system architecture can be broadly categorized into two fields: First the networked systems, where computing nodes are explicitly addressed and communication is modeled using a variation of message passing. The second category describes clusters of nodes participating in MapReduce based algorithms: The communication between the nodes is not modeled explicitly anymore, instead the parallelization semantics of the MapReduce model are applied.

Networked Systems In the *networked systems* environment, computing nodes have no shared memory. Communication between processes is done using a network protocol like TCP/IP or Infiniband. The most used communication model is the *Message Passing Interface*(MPI) (which is used by the most high performance computing systems).

The earliest proposition of a parallel version of k-mean found is by Stoffel and Belkoniene [91]. The authors propose parallelization of the assignment step (similar to algorithm 5). The paper emphasizes the difference between cluster recalculation after every assignment (which can't be done in parallel effectively) and global recalculation after each assignment step (see 2.2.1). The performance was tested on a 32 PC network connected by 10MBit Ethernet. The authors claim 90% efficiency for this setup and a data set of 20 attributes and 100 000 objects (with 20 clusters). The time needed to distribute the data is not measured.

Dhillon and Modha [28] propose a parallel k-means algorithm for distributed memory multiprocessors using the MPI-communication model. The paper uses the approach of splitting the data into p chunks and furthermore computes partial results for the new centroids: For each processes k "weighted centroids" are calculated by adding up all vectors belonging to each cluster. Also, the amounts of added data elements are tracked for each cluster. This enables the algorithm to recalculate the centroids by summing up each local weighted centroid and dividing them by the total summed up amount of data elements for each cluster (see algorithm 6). This method works because the mean calculation can be splitted into local operations without changing the result (see section 4.1.2). The empirical analysis shows that, while speedup degrades with smaller n, it's near linear for a data size greater than 2^21 (for d=8 and k=8). Furthermore, the measurements show linear scaleup for varying n,d and k. The approach by Dhillon and Modha is enhanced by Tian [94] by adding a subset clustering seeding and by Joshi [58] by adding bisecting k-means based seeding.

The approach of Kantabutra et al. [59] is to partition input data into k subsets. These subsets are sent to the computation nodes in a master/slave approach: every slave computes the mean of its own subset and broadcasts it. Afterwards the slaves calculate all distances and broadcast the resulting subsets to all other slaves. From this broadcast every slave receives the data elements assigned to its own managed cluster and repeats the process. At the end, the master process collects the resulting subsets and returns them. The used communication model is MPI over TCP/IP. Experiments show that no speedup is achieved until $n > 600K$ (with d=2 and k=4 on random data). For greater data size a speedup between one and two is measured (for four nodes). The disadvantage of this approach is that the number of possible processing elements is determined by the number of cluster centers.

MapReduce Based Approaches The first approach to adapt k-Means to the MapReduce paradigm is proposed by Chu et al. [21] as part of a broadly applicable programming method for algorithms that fit the statistical query model. The used algorithm resembles the local centroids: the data is split into p subgroups and in the map-function assigned in parallel. Afterwards the local weighted centroids along with the amounts are summed up by the reduce-function. Measurements conducted on a sixteen node cluster show a speedup between eight and twelve for sixteen used nodes. The approach is essentially repeated by Zhao [105] explicitly for k-means. Furthermore, empirical results for sizeup, speedup and scaleup are given. Results show that speedup is between 2.5 and 3.5 for 4 nodes; the speedup degrades with a smaller dataset (the datasets tested were $1, 2, 4$ and 8 GB in size).

Another approach proposed Li et al. [73] uses locality sensitive hashing (LSH) to implement various optimization approaches: First, the initialization phase is executed on prototype points extracted from the hash buckets created by the LSH. This provides efficient creation of a initial centroid configuration. Furthermore, the prototypes are used to prune unnecessary distance calculation during the assignment step: Due to the locality feature of the hash function, for a prototype point p distances are calculated only for the centroids placed in buckets near the hash bucket of p. Furthermore, all points represented by p can be assigned at once.

3.3.2. Shared Memory Multicore Systems

The first contribution found explicitly targeting shared memory systems is by Hohlt [48]: The author proposes a pthread based parallelization of k-Means. The parallelization follows the scheme depicted in algorithm 6. First, the data is partitioned into p chunks and then, the assignment is done in parallel. The feature data, membership of the data and the global centroids are shared between the used threads. Empirical analysis shows that the recalculation step of the mean takes up only a negligible fraction of the overall runtime. The experiments conducted (on image segmentation data) show that speedup is near linear for one to four threads. Another contribution for shared memory systems was made by Kucukyilmaz [64]. The approach uses threads with a fixed capacity to execute the assignment step in parallel. Centroid recalculation is done following the *localcentroid* (algorithm 6) scheme. Kucukyilmaz adds a parallel seeding variation, where each processing task initializes a fraction of the initial centroids.

kd-Tree Based Parallelization Parallelization approaches for the kd-tree based k-means were proposed by Gursoy in [42] for shared memory using pthreads and [41] for distributed memory environments. The first approach is to parallelize the distance calculations occurring while the tree is traversed: A thread takes nodes out of a shared work pool and applies the calculations needed to filter the set of centroid candidates. It then proceeds with the left child node while storing the right child node in the pool. The right child node is then processed by other available threads.

The other proposed approach works similar to the **p-chunks** scheme: The input data is divided into equal sized subsets and for each subset a kd-tree is generated, which is then traversed using the *filtering algorithm* (see section 3.2.1). This approach is called **random decomposition**. To further improve this approach, the data can be partitioned by dividing the input set into subspaces using a geographical partitioning. This variation is called **spatial decomposition**.

3.3.3. Grid Computing/Distributed Systems

Another type of target systems are *distributed systems.* In contrast to the system architecture used in the previous target environments, distributed systems are not assumed to be under the control of one individual. This leads to several implications regarding the algorithm design: First, the data distribution cannot be assumed to be evenly across the involved computing nodes. Second, communication time between nodes can be non-uniform, making moving data difficult. This leads to a series of approximative k-Means algorithms, trying to calculate clusterings using as few data reads as possible.

Jin et al. [56] propose an *drop-in* approach to k-Means combined with a distributed version that targets loosely coupled computing nodes in a setting where the input data is unevenly distributed over the machines. The authors show that in the presence of load imbalance the proposed algorithm outperforms a parallel k-Means version using p-chunks.

3.3.4. Others

Another approach to speed up runtime of k-means is to use special purpose hardware like Graphics Processing Units (GPUs), Manycore Solutions or Field Programmable Gate Arrays which provide ample parallelism capabilities and can be programmed using Frameworks such as Compute Unified Device Architecture (CUDA) or the Open Computing Language (OpenCL) [102].

Graphics processing unit (GPU) Farivar et al. [36] propose a parallel version of k-means using the CUDA programming interface to leverage GPU processing power. The parallelization follows the p-chunks algorithm, with a sequential cluster recalculation step. The centroids of each iteration are stored in the *constant memory* which is an 8 KB cache per thread. This limits the dimensionality of the data. Experiments were conducted using 1 million one dimensional data elements and 4.000 cluster centers. The conducted experiments showed an achieved speedup factor of 13, compared to a general purpose 2Ghz CPU.

Manycore Solutions (MIC) Another specialized hardware example is the Intel Many Integrated Core (MIC) co-processor, which offers a high number of hardware level threads combined with integrated units for vectorized execution of computation (SIMD). In [96] Wu et al. propose a vectorized implementation of Lloyds algorithm tailored for the use on the MIC co-processor (and using OpenMP as the threading model). The approach taken by the authors is to invert the distance calculation sequence from iterating the centroids for each point to iterating the data points for each centroid. The distance calculations are executed in a vectorized settings and the results are send to a CPU acting as the master. The distance comparison and assignment is done by this CPU. The evaluation done shows that speedup scales near linear for a number of threads lesser then 32, when clustering a dataset consisting of 5 million elements, each with a dimensionality of 10, into 50 clusters.

Algorithm 5: p-chunks problem decomposition of the k-Means cluster assigment step

```
Data: vectors: $x_1, \dots, x_n$ with $x_i \in \mathbb{R}^d$
Data: centroids: $c_1, \dots, c_k$ with $c_i \in \mathbb{R}^d$
Data: processing units: P
for processing node $p \in P$ do in parallel
    for $i \leftarrow (i-1) * (n/P), i * (n/P)$ do
        for $j \leftarrow 1, k$ do
            $\text{dist}_p, j \leftarrow \text{distance}(x_i, c_j)$;
        end
        clusterIndex $\leftarrow$ index(min($dist_{j,0<j<k}$);
    end
end
Cluster recalculation;
```

Algorithm 6: local centroid calculation and global aggregation parallelization scheme

```
Data: vectors: $x_1, \dots, x_n$ with $x_i \in \mathbb{R}^d$
Data: centroids: $c_1, \dots, c_k$ with $c_i \in \mathbb{R}^d$
Data: processing units: P
for processing node $p \in P$ do in parallel
    for $i \leftarrow (i-1) * (n/P), i * (n/P)$ do
        for $j \leftarrow 1, k$ do
            $\text{dist}_p, j \leftarrow \text{distanz}(x_i, c_j)$;
        end
        localVector[j]$+ = x_i : i = $ index(min($dist_{j,0<j<k}$));
        localAmount[j]++;
    end
    send $localAmount_p$ to master;
    send $localVector_p$ to master;
end
for $i \leftarrow 1, k$ do
    for processing node $p \in P$ do
        $globalVector[i] + = localVector_p[i]$;
        $globalAmount[i] + = localAmount_p[i]$;
    end
    $c[i] \leftarrow (globalVector[i]/globalAmount[i])$;
end
```

4. Definition of the General k-Means Skeleton

In order to find a suitable abstraction for the skeleton, multiple aspects have to be considered: The abstraction level must be high enough to allow different variations of the algorithm to be expressed. On the other side, the abstraction level must allow implementations to leverage possible parallelization schemes. In section 4.1 distinctive features of the selected k-Means variants are analyzed. Based on the insights the type of the higher order function is motivated in section 4.2 and the possible parallelization schemes are presented in section 4.2.1.

4.1. Features of k-Means-Algorithms

Starting out with the abstract steps depicted in algorithm 3, the following chapter compares realization variants and features of k-Means in respect to parallelization possibilities.

4.1.1. Assignment Step

The first feature used in the assignment step is the choice of a distance metric. The main difference of the metrics looked at in section 2.2.2 is the **amount of shared data** used to determine the distance: While Lloyds algorithm uses only the point in question and the set of cluster centroids, other algorithms use secondary data structures (for example the membership matrix used by fuzzy c-means) which are shared between distance calculations. Another example for this is the *point symmetry distance* which depends on the value of multiple input points. These dependencies could lead to difficulties when executing distance calculations in parallel because they represent a shared resource and could therefore be subject of contention. However as long as there are no direct dependencies between calculations of different input data elements it is still possible to parallelize this step (as seen in section 3.3) by using the *p-chunks* decomposition scheme (see algorithm 5). The chosen distance metric also determines the applicability of drop-in improvements using geometric reasoning (see section 3.2.1): When the distance metric doesn't conform to the **metric constraints** (especially the triangle inequality), drop-in improvements cannot be applied. Therefore we need a general k-Means skeleton definition that is agnostic to these drop-in variants. Furthermore, The distance metric can be relaxed to distance functions operating on **nominal or binary features** (for example the k-Medoids algorithm). To account for this variations both the cluster assignment and the centroid update step have to operate on abstract data spaces accounting for both nominal and continuous features. Another task that theoretically could be done in parallel is the distance calculation for each dimension (assuming that the metric allows for independently calculation of dimension-subsets and aggregation of the results).

4.1.2. Update Step

For the recalculation of the cluster model in the update step even greater differences between the algorithmic variations exist. They range from the sequential recalculation of every new cluster centroid (e.g. k-median) to the simple aggregation of parallel computed partial results. As shown by Kantabutra et al. [59] the recalculation logic of a centroid for one cluster has no effects on the recalculation of other clusters and therefore can be done in parallel. This holds up for the other selected variations as well which leads to a possible parallel execution of the update step. Another important aspect is the distribution semantic of the recalculation function:

Aggregation Function Properties When considering parallelization of the recalculation step, it is necessary to reason about the distribution semantics of the aggregation functions. In general, aggregation functions can be classified in respect to their distribution possibilities as follows [107, p. 47]:

distributive functions: Distributive functions can be evaluated on different subsets of the data and merged, without the result depending on the subset partition. Formally we call a function F distributive when an operator Θ exists so that $F(u \Theta v) = F(u) \Theta F(v) \forall u, v$.

algebraic functions: An aggregation function with m arguments is called algebraic if every argument of the final result is a distributive function. An example is the mean: Sum and Count of the Elements can be independently collected.

holistic function: Every aggregation function that needs full knowledge of the input and therefore can not be partially calculated (e.g. the median).

When using this classification we see that for *distributive* or *algebraic* functions we can build partial results of the recalculation during the parallel execution of the assignment step (as shown in algorithm 6).

4.1.3. Convergence Criterion

The convergence criteria proposed by the different variations can be classified into two subsets: Lloyds algorithm defines convergence as the state where no data elements are assigned to different centroids between two cluster iterations. Other variants define the convergence as the stability of a secondary criterion between the two iterations (either by defining equality or in the context of fuzzy clustering defining a threshold). This can be abstracted to a function operating on two representations of the clustering configuration (either the clustering itself or a secondary value), that determines the convergence of the algorithm. The k-Means algorithm is inherently stateful: because we have to compare iteration results (for the convergence criterion), we have to provide informations about the model before and after one specific iteration. From this, pathological cases can be constructed, defining the model so that its data size makes storing the model impossible. Unfortunately, algorithmic skeletons provide no means to syntactically prohibit this case. Therefore using secondary criteria like the mean squared error is encouraged to keep memory overhead low.

Similar to the recalculation of the cluster centroids, it is possible to calculate partial results of the convergence criteria if the aggregation function for the criterion is *distributive* or *algebraic* (for example the mean squared error can be calculated in parallel by summing up the error of each distance calculation, see [28]).

4.2. Higher Order Function

The type system used for the following chapter is taken from the work of Leyton [70] who provided a typed skeleton system along with correctness proof and a skeleton library implementation. When designing the higher order function representing the k-Means skeleton, the starting point is the definition of the input data space. We define an input space $\mathbb{V}$ that is deliberately underspecified. With this we can account for elements having continuous features as well as binary or nominal ones (e.g. the **k-Medoids** algorithm). Another advantage of leaving the input data organization underspecified is that drop-in improvements like **kd-tree based k-Means** can be applied to the general skeleton without changing its type signature. An example of this is the parallel execution of kd-tree based k-Means where $\mathbb{V}$ represents a set of kd-trees and the split for the assignment step simply assigns one kd-tree to each parallel execution in the assignment step.

The input data is defined as a set $V \in \mathbb{V}$ that has a cardinality of n. To enable parallel processing of input elements we need a way to split the data in subsets. This is modeled as a function:

$$f_{split} : \mathbb{V} \to [\mathbb{V}]$$

The next consideration to take into account is the abstract generalized input and output type of the skeleton. The abstract goal of k-Means is the iterative refinement of a clustering configuration. In Lloyds algorithm this cluster configuration is represented by a set of centroids. The algorithm takes an input set and returns a set of updated centroids representing a local minimum. Furthermore, we need the input data for the algorithm to work on. This can be modeled as the abstract algorithm having a type of

$$f_{kmeans} : \langle \mathbb{V} \times \mathbb{C} \rangle \to \mathbb{C}$$

where C represents a set of elements from the input space $\mathbb{C} \subset \mathbb{V}$ (note that this does not mean that the centroids are part of the input elements). To furthermore abstract the concept of the centroid cluster representation we define a model space $\mathbb{M}$ that represents different ways to express the cluster configuration (for example the set of centroids $C \in \mathbb{M}$). This leads to the following type of the general k-means skeleton

$$f_{kmeans} : \langle \mathbb{V} \times \mathbb{M} \rangle \to \mathbb{M}$$

Modeling the Convergence Criterion As shown in section 4.1.3 the convergence criterion can be modeled as a function comparing model representation between the iterations. Given the model type introduced in the last paragraph we can define the convergence criterion as:

$$f_c : \langle \mathbb{M} \times \mathbb{M} \rangle \to \mathbb{B}$$

Another advantage of the abstract model space is that secondary criteria can be defined as part of the model (e.g. $M := \langle \mathbb{N} \times \mathbb{C} \rangle$) where the first part of the tuple represents the squared error and a convergence function for Lloyds algorithm could be modeled as $f_c(a, b) \mapsto true$ if $a_1 < b_1$.

Skeleton Definition The simplest way to model the abstract k-Means skeleton would be to define a skeleton that takes as input a function for the assignment/expectation step f_e, a function for the update/maximization step f_m and the convergence criterion function f_c. However this approach has several drawbacks: The aggregation semantics of the maximization function (see section 4.1.2) provide different parallelization possibilities for the update/maximization step, that cannot be expressed by one skeleton. Furthermore, defining the skeleton using only the 3 functions disables further nesting capabilities for the skeleton. According to Leyton [70], nesting is an important factor for skeletons in order to achieve scalability on heterogeneous systems. Therefore, a better model for the k-Means Skeletons is an interleaving of two sub-skeletons for each step of the algorithm, called **expectation** and **maximization** skeleton. This leads to a higher order function consisting of the following components:

- The input functions: The convergence criterion f_c which has a type definition of $f_c : \langle \mathbb{M} \times \mathbb{M} \rangle \rightarrow \mathbb{B}$, an expectation skeleton and a maximization skeleton.
- The output of the higher order function is the abstract algorithm definition f_{kmeans}, having a type of $f_{kmeans} : \langle \mathbb{V} \times \mathbb{M} \rangle \rightarrow \mathbb{M}$

This leads to the following definition of a general k-Means skeleton:

$$\begin{aligned} \triangle_{kmeans}(f_c, \triangle_e, \triangle_m) :& \langle \mathbb{V} \times \mathbb{M} \rangle \rightarrow \mathbb{M} \\ \text{input} \mapsto & \text{output if } f_c(\text{input}, \text{output}) \\ \mapsto & \triangle_e \circ \triangle_m(\text{input}) \text{otherwise} \end{aligned}$$

Where the sub-skeletons are defined as follows:

Expectation Skeleton For the different proposed distance metrics a general function f_e can be established, which assigns every data element to a cluster. The type of this function is therefore $f_e : \mathbb{V} \rightarrow \langle \mathbb{N} \times \mathbb{V} \rangle$. Note that this assignment is not necessarily the fixed assignment. Fuzzy c-means can be simulated with this behavior by defining the probabilities as part of the data element and choosing the cluster with the highest probability as assignment result. The overall skeleton takes the model and input data elements as input and produces an intermediate representation of the labeled data elements as output, which is then processed by the maximization skeleton.

Maximization Skeleton The maximization skeleton calculates the new model for the algorithm from the labeled data output produced by the expectation skeleton. The formal restrictions on this skeleton depend on the intermediate data from the expectation step. Furthermore, the output has to be of type $\mathbb{M}$ so that it can be used as the model for the next iteration. This leads to a corresponding function type of: $f_m : \langle \mathbb{N} \times \mathbb{V} \rangle \to \mathbb{M}$

4.2.1. Parallelization Schemes

In the original work of Cole [24], for every proposed skeleton there is an exemplary implementation on a theoretical computer-model (a processor grid) provided. The drawback of the used hardware model is that lacks applicability to currently used systems, both in the home computing sector and in data-center environments. Instead of providing a model for the processor grid this thesis proposes four different **parallelization schemes** which are evaluated empirically (see section 6). Subsequently the different schemes are presented and their features explained. For each parallelization scheme the corresponding configuration of the sub-skeletons is shown and the parallelization semantics are visualized by providing an UML activity diagram describing the parallel execution and synchronization points for each scheme.

Assignment Step All further parallelization schemes make use of the data parallelism inherent to the k-Means algorithm, by using data decomposition following the *p-chunks* scheme explained in section 4.1.1. The differences between the parallelization schemes therefore lie in the handling of the update/maximization step:

Sequential Maximization The first parallelization scheme derives from the fact that the update/maximization step contributes only to a minor portion of the total runtime (as seen in [48]): While the assignment step is executed in parallel, the update is carried out sequentially for all results by the master.

$$\begin{aligned}
&\triangle_{kmeans}(\triangle_e, \triangle_m, f_c) \\
&\qquad\qquad \text{with } \triangle_e = map(f_{split}, f_e, f_{merge}) \\
&\qquad\qquad\qquad \triangle_m = seq(f_m)
\end{aligned}$$

Where f_{split} and f_{merge} split the input according to the *p-chunks* scheme. The expectation step is carried out by a function $f_e : \langle \mathbb{V} \times \mathbb{M} \rangle \to \langle \mathbb{N} \times \mathbb{V} \rangle$ that computes the distance of every data element in the input subset to the corresponding cluster centroids and outputs the index of the centroid with the lowest distance. f_m takes the assigned data elements and inputs and uses them to calculate the new cluster centers given a function type of $f_m : \langle \mathbb{N} \times \mathbb{V} \rangle \to \mathbb{M}$. The parallelization semantics can be described by an UML activity diagram as shown in figure 4. After the parallel assignment step, the threads are synchronized, the results merged and the master executed the maximization step.

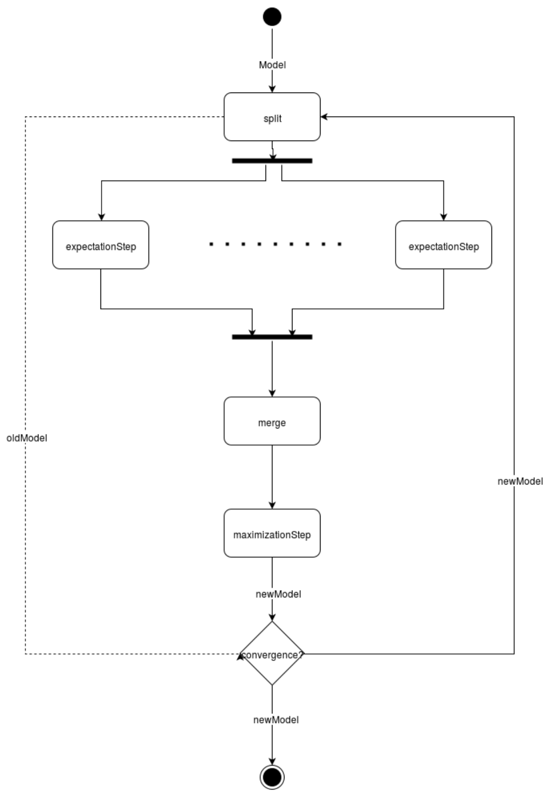


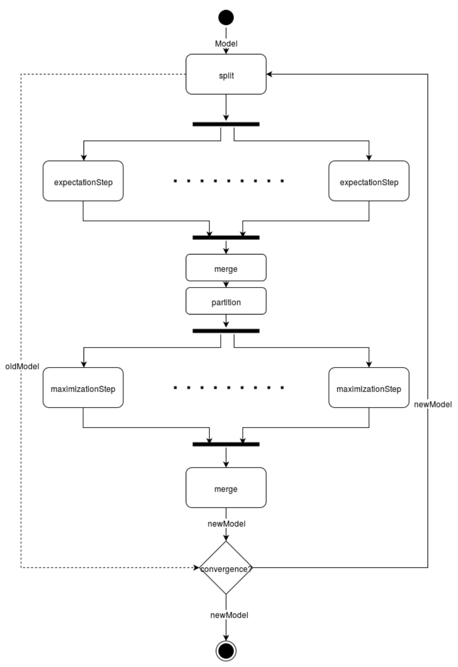
Figure 4: Activity diagram for the *sequential maximization* parallelization scheme

Parallel Maximization leverages the fact that in various k-Means based algorithms centroid recalculation depends only on the data points assigned to the corresponding cluster (for example calculating the means of all points in a cluster in Lloyds algorithm). Therefore, the centroid update can be executed in parallel by splitting the input data along the cluster assignments (k). The parallelization of the maximization step is limited: When the number of computing nodes exceeds the number of desired clusters, computing nodes can't be used (similar to the approach taken by Kantabura et al. in [59]. The parallel maximization scheme can be expressed by the following configuration for the general k-Means skeleton:

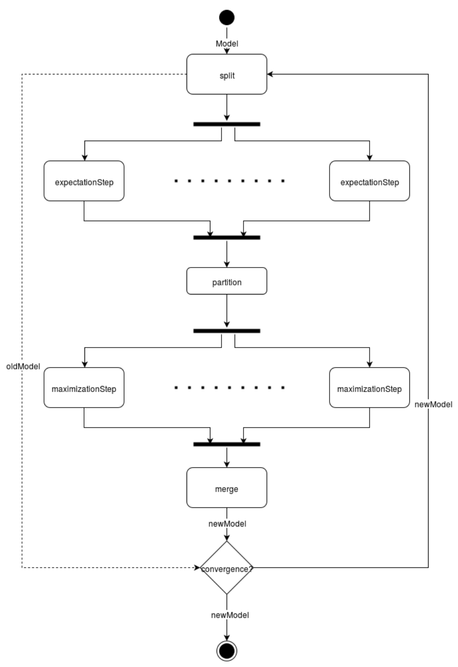
$$\begin{aligned} &\triangle_{kmeans}(\triangle_e, \triangle_m, f_c) \\ &\qquad\qquad \text{with } \triangle_e = map(f_{split}, f_e, f_{merge}) \\ &\qquad\qquad\qquad \triangle_m = map(f_{partition}, f_m, f_{merge2}) \end{aligned}$$

where the expectation skeleton is defined in the same way as it was in the *sequential maximization*: The input is split using the *p-chunks* scheme and assignment is carried out in parallel. For the maximization skeleton, we need a function that splits the labeled data on the lines of the clusters, leading to $f_{partition} : \langle \mathbb{N} \times \mathbb{V} \rangle \rightarrow [\mathbb{V}]$ The resulting clusters can now be used to recalculate the new centroids in parallel. This is done by the maximization function $f_m : \mathbb{V} \rightarrow \mathbb{M}$ (an example for Lloyds algorithm would be f_m calculating the mean of the input elements). The new cluster representations now have to be merged to form a new global model. This is done by the f_{merge2} function that merges the results $f_{merge2} : [\mathbb{M}] \rightarrow \mathbb{M}$. With this model, the first map skeleton merges its partial outputs to an intermediate data representation which is then re-split by the partition function of the maximization skeleton. We can condense this to a function $f_p : [\mathbb{V}] \rightarrow [\mathbb{M}]$. The condensed scheme is subsequently called **Hybrid Partition**. The UML activity diagrams for both schemata can be seen in figure 5.

Partial Aggregation The partial aggregation scheme is a method which only works for model-update functions that are either *distributive* or *algebraic* (see 4.1.2). It uses the parallelization of the maximization step proposed by map-reduce based approaches (see 3.3.1): The expectation step is carried out in parallel and the intermediate results are aggregated by a *partial merge* function (like shown in figure 6). There is no static configuration of the general k-Means skeleton for this parallelization scheme due to the fact that the parallelism degree of the partial merge step has to be chosen by the application developer. In the following evaluation the partial merge is implemented by



(a) Activity diagram for the *sequential maximization* parallelization scheme



(b) Activity diagram for the *sequential maximization* parallelization scheme

Figure 5: Activity diagrams for the parallelization schemes based on a parallel execution of the maximization/update step

defining the skeleton as:

$$\begin{aligned}\triangle_{kmeans}(\triangle_e, \triangle_m, f_c)& \\ \text{with } \triangle_e &= map(f_e) \\ \triangle_m &= seq(f_{merge})\end{aligned}$$

and defining the output of the expectation skeleton to be the partial computation results. In the example of Llloyds algorithm, this means defining the output of the expectation skeleton as a tuple $\langle \text{sum}_p, \text{count}_p \rangle$, where sum_p defines a vector with the sum of all elements assigned to a cluster and count_p the count of elements assigned to a cluster. This means that the maximization skeleton is reduced to a simple f_{merge} that computes $\text{sum}_p/\text{count}_p$ for each cluster. This scheme can only be applied to *distributive* and *algebraic* aggregation functions.

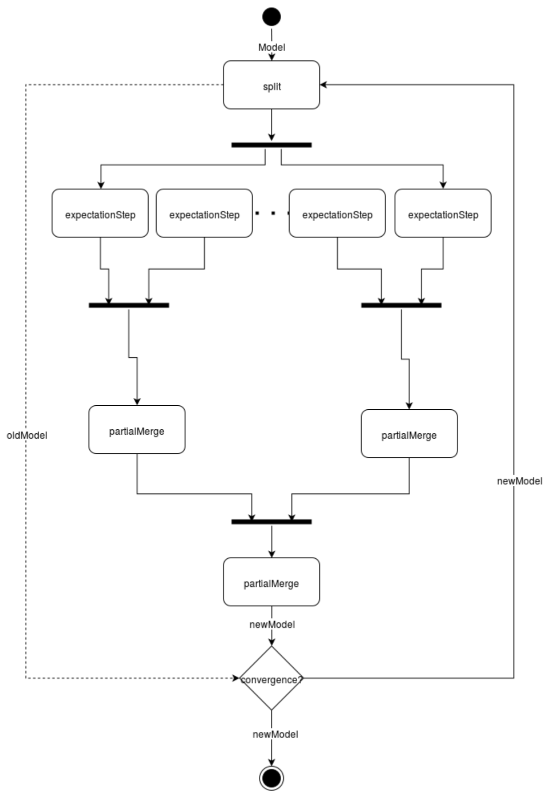


Figure 6: Activity diagram for the *partial merge* parallelization scheme

5. Realization

To implement the parallelization schemes an algorithmic skeleton library called **Skandium** (proposed by [71]) was chosen. Skandium is an algorithmic skeleton library for Java, based on the semantics of the Calcium library [70] and implemented on top of the Java Executor Framework, therefore targeting shared memory multi-core systems. The next section (5.1) lists the advantages and drawbacks of the chosen approach. The conversion semantics defined by Calcium are applied to the general k-Means skeleton in section 5.2.

5.1. Skandium

Choosing Java as the implementation language has the advantage of using a widespread language known by many people that could possibly benefit from the proposed solutions (especially in the setting of data mining and machine learning). It provides ease of use and multiple aspects (such as first class functions, concurrent data structures or asynchronous execution abstractions) useful for the implementation of algorithmic skeletons. In addition the execution environment provides portability and interoperability with other JVM-based languages (this enables usage in polyglot programs based on the JVM [35]). A disadvantage is the indirect memory handling, which provides no means to directly reason about memory layout of implemented data structures or perform manual cache optimization, complicating the analysis and discussion of runtime deviations. Furthermore, the *garbage collection* system of the JVM introduces performance side effects (see [7]) Using an open source library as the algorithmic skeleton abstraction has the advantage of easy customization and expandability while providing concise execution semantics (see 5.2) and build in optimizations (such as a *notify*-system to communicate between tasks and their children). The library provides the abstractions needed by algorithmic skeletons without introducing the need to learn a new language. A disadvantage is that the Java programming language provides no method of defining pure functions, therefore giving no language level protection against parallel programming pitfalls: It is still possible to construct data races or deadlocks when using the Skandium library.

Assumptions To provide the execution model used in Skandium, multiple assumptions about the given skeletons are made. These are subsequently outlined and analyzed in respect to the given problem. Further explanations can be found in [70, Chapter 3].

Single input/output: Skeletons can only receive/produce scalar inputs/outputs. This assumption can be fulfilled by defining a model class that represents data, centroids and assignments. The trade-off here is between the expressiveness of the algorithm and extensive data copying/sharing

Passive skeletons: Each skeleton output is directly related to a previously received input. This has little impact on the proposed implementation, because the process is triggered by an input model and produces exactly one output model corresponding to the convergent state.

Stateless skeletons: Skeletons are stateless and therefore their sequential blocks are also stateless. This imposes a restriction on the convergence criterion: the state of the last iteration has to be compared with the current state to determine convergence. This fact is accounted for by the type definition of the criterion function f_c as $\langle \mathbb{M} \times \mathbb{M} \rangle \to \mathbb{B}$: The state is kept outside the function (old and new Model are passed to the function as parameters). The old model to compare with is kept in memory by the corresponding iteration-instruction (see section 5.2). The drawbacks of this approach are outlined in section 4.2.

5.2. Instruction Generation Semantics

The Skandium library uses the instruction approach defined by Leyton [70] to convert the abstract skeletons into an executable system. The outermost instruction mapping converts the sub-skeletons for expectation and maximization step to instruction sets using their corresponding rules. Furthermore, a $kmeans_I$ instruction is generated that represents the skeleton as a whole:

$$\frac{\triangle \twoheadrightarrow S}{\triangle_{kmeans}(f_c, \triangle_e, \triangle_m) \twoheadrightarrow kmeans_I(f_c, S_e, S_m)}$$

In order to transform the k-Means skeleton to instructions, fitting the Skandium model, we have to enable transparent handling of splitting the given model parameter (p) into a pair (m_{old}, m_{new}) that can be handled by the criterion function f_c. Furthermore, we need to model the ongoing iteration depending on the result of f_c. To achieve this using Skandium instruction reduction rules the following instruction types are introduced:

- $loop_I$: This instruction represents one loop iteration of the k-Means algorithm. It applies the criterion function to the model pair (given to it as the parameter) and returns the result model if convergence has been reached. Otherwise it pushes the sub-skeleton and a copy of itself to the instruction stack to continue the loop.

- es_I is a utility function to enable transparent handling of the statefulness of k-Means: although the loop instruction accepts a tuple m_{old}, m_{new}, the expectation skeleton accepts a single model as parameter. To convert the stateful representation back to the new model needed for the next iteration, the es_I instruction passes only the current model to the sub-skeleton.

- ms_I is the counterpart of es_I. It is created by the loop function and parameterized with the model of the current iteration. After the evaluation of the expectation and maximization steps the ms_I instruction takes the output of the maximization skeleton and joins it with the model parameter to gain the tuple m_{old}, m_{new} which is passed as input to the next loop iteration.

Reduction Rules Evaluating the instruction $kmeans_I$ splits the given model for further processing in the loop instruction:

$$\frac{p : M}{kmeans_I(f_c, S_e, S_m)(p) \rightarrow loop_I(f_c, S_e, S_m)(p_{(1)}, null)}$$

$$\frac{p : M \times M}{kmeans_I(f_c, S_e, S_m)(p) \rightarrow p_{(2)}}$$

The loop instruction pushes further instances of itself on the stack, depending on the convergence criterion. Furthermore, it generates the maximization instruction that saves the previous value of the model used in the criterion function.

$$\frac{f_c(p) = true}{loop_I(f_c, S_c, S_m)(p) \rightarrow p}$$

$$\frac{f_c(p) = false}{loop_I(f_c, S_c, S_m)(p) \rightarrow es_I(p) \cdot S_e \cdot S_m \cdot ms_I(p_{(2)}) \cdot loop_I}$$

Lastly, the support instructions handle supplying the sub-skeletons and the convergence criterion with the needed model parameters:

$$es_I(p) \rightarrow p_{(2)} \quad ms_I(p_{old})(p) \rightarrow (p_{old}, p)$$

The generated instruction stacks are processed by tasks as depicted in figure 7: While tasks exist in the ready queue, the system assigns them to the thread pool. Every task consumed possibly generates new sub-tasks, which are added to the queue as well. When all tasks are marked as finished, the result is put into the output stream.

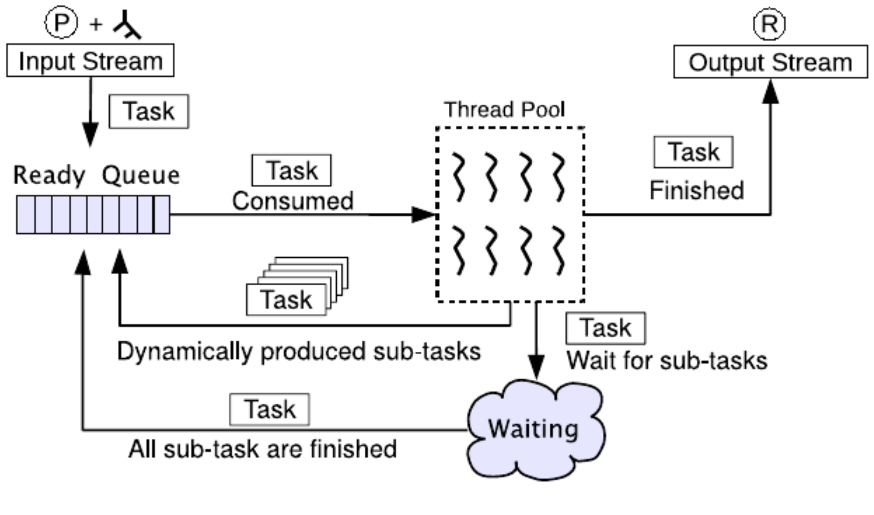


Figure 7: Principles of Skandium task processing [71]

6. Evaluation

Due to the abstract general nature of the proposed skeleton, possible usages have many degrees of freedom, leading to various usage scenarios. One scenario can be categorized by:

- The algorithm used
- The parallelization scheme used
- The parameters of the input data (number of input elements n, and dimensionality d of each input element)
- The desired number of clusters k
- The natural clustering of the data (random or well separated)
- The initial clustering model

In the next chapter the, proposed parallelization schemes are evaluated in respect to different properties: First we show that when using the original k-Means algorithm (Lloyds algorithm), the runtime results of the parallelization schemes match the theoretical model. To achieve this goal, the measurements conducted by Kucukyilmaz [64] are replicated using the given implementation. Second we evaluate the efficiency of the different parallelization schemes and compare them with the results achieved by related work in the field. Dhillon and Modha [28] show that the efficiency of the parallel algorithms depends on the input data size n. To validate this, various experiments with different input data sizes are conducted. To determine the overhead induced by using Skandium as the skeleton library, a manual parallelization is implemented and compared to the skeleton-based implementation in respect to their runtime.

A main aspect of the skeleton is the adaptability to different usage scenarios. To show that it is possible to use drop-in improvements of k-Means, a kd-tree based version is implemented using the skeleton and its parallelization possibilities are shown. Next we show that different variations of k-Means can be implemented using the skeleton. We show how to realize implementations of the k-median and the fuzzy c-means algorithm. For both algorithms we show the applicability of the parallelization schemes and influence of the different data access patterns and computation shares. In the last section we compare achievable speedup for the different implemented variations of k-Means to determine shared properties and influence of different expectation/maximization skeletons on achievable speedup.

6.1. Methodology

To evaluate runtime behavior and parallelism, the proposed parallelization schemes were implemented as skeletons with the *Skandium* skeleton framework. The implementations were integrated into a Java application, which was deployed to the target system using a single jar file. Measurements were taken on a multi-tenant system provided by the Computer Science Department of the Humboldt University. To mitigate influences caused by system utilization, the measurements were repeated multiple times, through different times of the day and days of the week. The hardware specifications, along with used software versions can be found in appendix A.3.3.

Data Sets The data set used to compare changes in the input element parameters (number of values n, attributes of a data element d) is synthetically generated. It consists of random double precision float values (8 bytes) between zero and one. Furthermore, multiple real world application datasets are used (for an overview see A.3.1). The datasets are completely loaded into memory and made accessible to the skeletons via immutable shared memory (static data).

Convergence Criterion For the conducted tests a convergence criterion is implemented that sets the number of iterations to a fixed amount (given as a parameter). This enables the data independent variation of the number of iterations.

Model Initialization All implemented variants use the **random from dataset** centroid initialization method (see section 3.2.3). To ensure comparability of the algorithm results a pseudo-random number generator is used which is initialized with a given seed. The clustering results of the different parallelization schemes are compared to their sequential counterparts to ensure validity.

Time Measurements The time measurements are performed by using the Java *System.currentTimeMillis()* method call. For every run two measurements are recorded: the **total time** measurement represents the time difference between the first and the last line of the main method, the **algorithm time** represents the runtime of the implementation excluding the time to set up the skeletons and read the input data from disk. The time measurements are stored in a SQLite Database, along with the input parameters of the execution. Until otherwise noted the subsequently presented runtime show results averaged over ten runs.

Parallelization Scheme Abbreviations The parallelization schemes illustrated in section 4.2.1 are abbreviated in the subsequent figures as follows: the sequential maximization scheme is denoted **sd-sm**, the parallel maximization scheme **sd-mm**, the hybrid partition scheme **sd-hp** and the partial aggregation scheme **sd-pm**.

Parallelization Metrics When examining parallel algorithms, it's not sufficient to show the validity of the results. Instead we need metrics to describe the quality of the parallelization. A number of metrics are proposed by literature, for measuring the efficiency and scalability of parallel algorithms [65]. For this thesis we measure the following metrics, which are widely used in parallel algorithm research [77]: Informally, **speedup** measures how the addition of further computing units reduces the runtime for a fixed problem. Formally, if a single computing unit needs T_1 time to process a given problem, and T_P denotes the time needed to process the same problem on P processing elements, then speedup is defined as:

$$\frac{T_1}{T_P}$$

The ideal speedup is linear, which means that the runtime directly depends on the amount of computing unit that are added to the system (e.g the runtime is half the sequential runtime for two computing units, etc) . Ideal speedup is hard to achieve, due to synchronization and communication overhead of parallel systems.

Scaleup defines the ability of a parallel system with p processing nodes to perform a p-times larger job on p in the same runtime as the original system. If $T(n)_p$ = time to process a problem of size n using p processing nodes this is defined as:

$$\frac{T(n)_1}{T(p * n)_p}$$

The computing cores available to the implementation is externally set via the use of the *taskset* unix utility. For the full measurement script see A.3.2.

All measurements assume that the input data fits into main memory (for data not fitting in main we expect performance drops, since the k-Means algorithm requires multiple passes over the data set. For further analysis of k-Means in a environment with disk-resident datasets the reader is referred to the work of Hadian and Sharivari [43]).

6.2. Lloyd/Forgy k-Means

The first set of experiments conducted uses the implementation logic of Lloyds algorithm (see algorithm 1). To evaluate if the given implementation conforms to the theoretical model in respect to runtime behavior, the following experiment conducted by Kucukyilmaz [64] is repeated. To examine the efficiency of the parallelization, we measure the speedup and scaleup metrics introduced in the last chapter for each of the parallelization schemes.

6.2.1. Cost Model Comparison

The first experiment conducted serves as a validity check for the Skandium based implementation provided. We expect the runtime of the implementation to be similar to both the results of Kucukyilmaz and the theoretical runtime complexity shown in section 1. The experiment sets fixed values for the dimensionality, number of clusters and iterations while varying the input element count. For a set with random input data and a dimensionality of 30, every parallelization scheme of the skeleton is run for three clusters ($k = 3$) and for a fixed number of five iterations $i = 5$. The runtime is measured for an input size ranging from $10,000$ to $1,000,000$ data elements (n). For the sequential version we expect a linear increase of runtime duration, as predicted by the cost model shown in section 2.2.1. For the parallel versions we expect a better runtime than the sequential version, bound by the optimal linear speedup achievable for the parallelization (in this case $^1/_8$ of the sequential runtime). Figure 8 shows the result of the experiment: We see that the runtime results conform with our theoretical model: with increasing input size the runtime deteriorates. For $n = 1,000,000$ the runtime rises higher than the predicted linear model while still staying inside the $O(nkdi)$ bounds. Possible explanations for this phenomenon include non-linear memory access on the target system or suboptimal cache accesses. The parallel versions show the expected decreased runtime compared to the sequential version. The efficiency of the different schemes is compared in the next section.

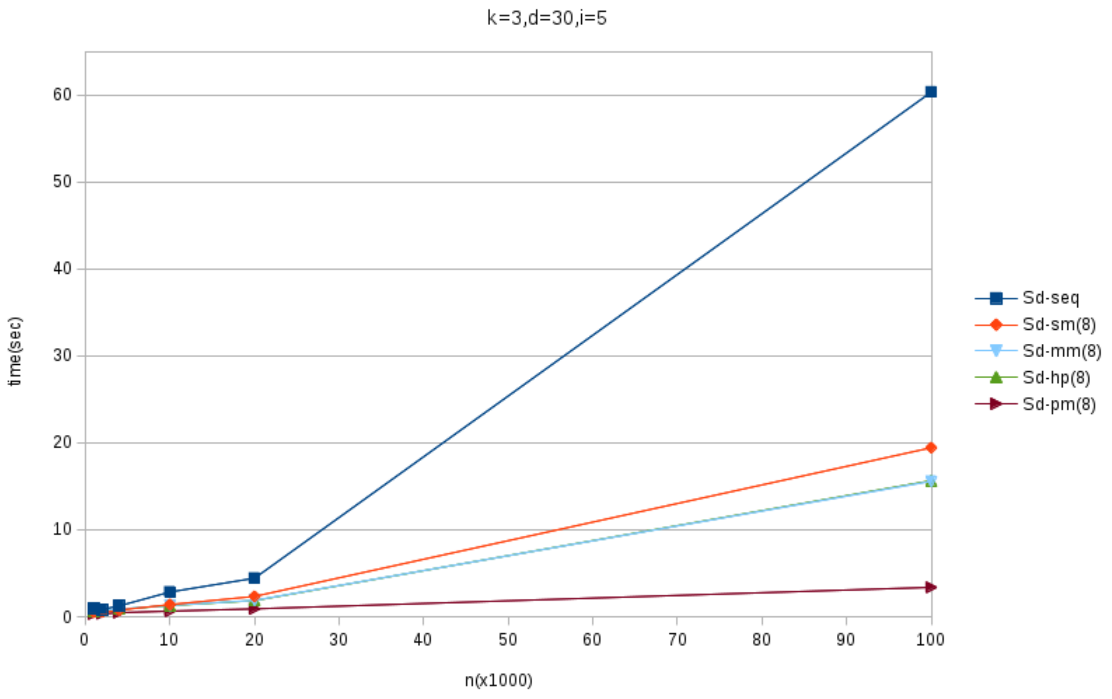


Figure 8: Runtime behavior of k-means implementations with respect to data size

6.2.2. Parallelization Schemes

After verifying the runtime of the skeleton and it's parallelization schemes in the last section we now investigate the features of the parallel versions further by using the common metrics of speedup and scaleup. The first experiment in the next section uses the *random* dataset and fixed values for the data size and k-Means-parameters ($d = 3, k = 10, i = 10, n = 5,000,000$) while gradually increasing the number of cores available to the program. For this configuration we measure the speedup and scaleup for the different provided parallelization schemes. For the *partial-merge* scheme we expect speedup measurements similar to the experiments conducted by Dhillon and Moda [28], because the skeleton configuration corresponds to the *local centroids* scheme as depicted in algorithm 6. For the *parallel maximization* and the *hybrid partition* scheme we expect similar speedup results, as the parallel maximization limits the sequential proportion of the executed code. Furthermore, we expect better speedup results for the *hybrid partition* scheme as the overhead induced by the execution of the merge and partition functions is reduced by unifying them into the hybrid-partition function. For the sequential maximization scheme we expect lower results, because the amount of sequential executed program is higher than for the other schemes.

The speedup results in figure 9 show the following insights: Although the speedup of the *partial-merge* scheme increases for a higher number of cores, it shows sub-linear behavior: The speedup reaches four for eight Cores which corresponds to 50% efficiency. There are various possible reasons for this behavior: Overall the skeleton library produces runtime overhead that varies with the input parameters: intermediate results of parallel execution paths are stored for synchronization and work is not executed directly but through a TaskExecutor given by the framework. To further investigate the overhead induced by the Skandium library, the experiments shown in section 6.2.5 were conducted.

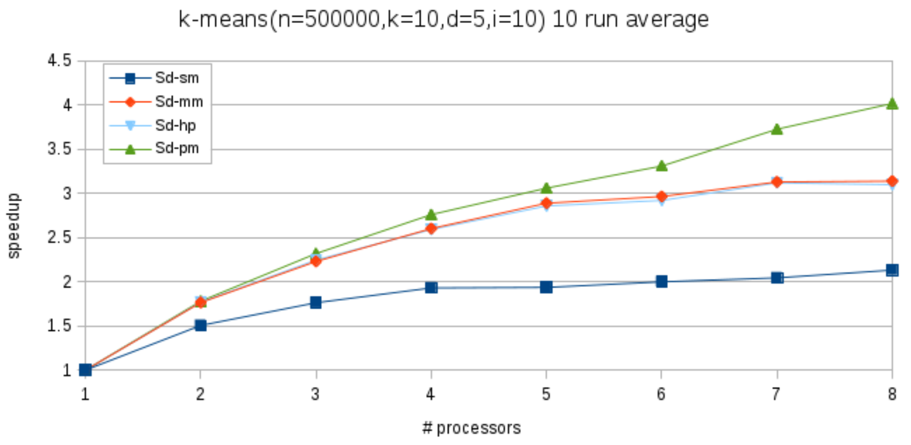


Figure 9: Speedup measurements for Lloyds algorithm on a random input data set with varying number of cores

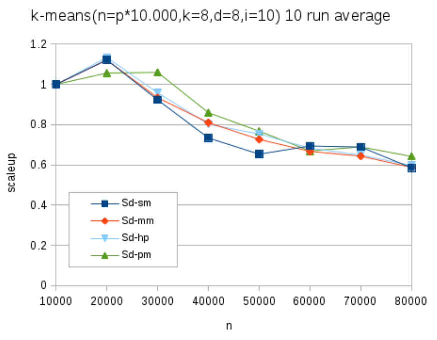
The scaleup property shows how the algorithm behaves when scaling the input problem size proportional to the available computing resources. For k-Means, this gives us three variables for the problem size: the number of input elements (n), the dimensionality of each element (d) and the expected number of clusters (k). For each of the variables the scaleup was measured:

Scaling the Number of Input Elements (n) The expected results for the scaleup property when increasing n are that the parallelization schemes behave similar as in the speedup measurements: due to the parallel nature of the assignment step which accounts for the majority of the algorithms runtime, increasing the problem size (in this case scaling the number of input elements) yields the same results as adding new cores. We therefore expect near linear scaleup for n with slight deterioration for higher processor count, due to the overhead of communication and the sequential recalculation of the cluster centroids. We expect that the scaleup penalty induced by the larger sequential parts has a higher impact on the sequential maximization scheme. In figure 10 the scaleup measurements for an input size of $n = p * 10.000$ are shown (where p denotes the number of processors available for the system). We see that after a phase of superlinear scaleup the values for all parallelization schemes deteriorate for higher processing unit count. This conflicts with the linear scaleup measured by Dhillon and Modha. There are multiple possible reasons for the deterioration: For greater problem size, the size of the input data increases. This leads to a greater amount of time spent in garbage collection increases. Furthermore, allocating and freeing the data structures for both the master and the execution in the different threads takes more time.

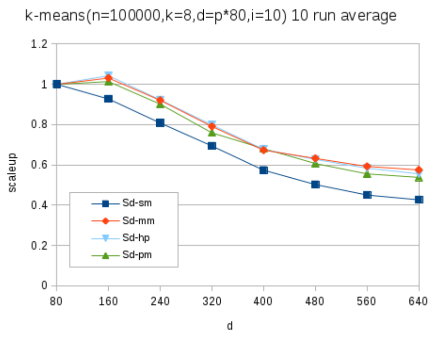
Scaling the Dimensionality (d) Based on the measurements done by Dhillon and Modha [28] we expect better than linear scaleup behavior for the partial merge scheme when increasing the dimensionality of the dataset. For the other parallelization schemes we expect near linear scaleup behavior: The increased dimensionality primarily increases the runtime spent on the assignment step, which is executed in parallel. Similar to scaling the number of input elements we expect the map-maximization and hybrid-partition scheme to outperform the sequential maximization scheme. The results in figure 10b show scaleup properties similar to the results of the number of input elements: scaleup deteriorates linearly after a brief superlinear speedup (for 2 cores). The possible reasons for the drop are similar the ones discussed for the input elements: Memory management takes up a greater share of the runtime (a problem that cannot be mitigated by adding more computing units). Another problem with increased dimensionality is that the data size of the centroid representation grows, leading to higher consumption of caching capabilities in the cores. Another observation is that the achieved speedup of the sequential maximization scheme is smaller than that of the other schemes. This could be explained by the increased computational share of the maximization step (having a runtime complexity depending on d), which can be mitigated by the map-maximization and hybrid partition scheme by using the available cores ($k > p$).

Scaling the Number of Expected Clusters (k) The last measurement conducted was varying the input parameter k, representing the number of assumed clusters in the dataset. The effects on the runtime of the map-maximization and hybrid-partition schemes are harder to predict than for the dimensionality or number of input elements: The degree of parallelization depends on the number of clusters k and therefore we expect runtime to increase when the number of cluster isn't divisible by the number of processing elements available (because this leads to work-imbalance). For the partial merge scheme we expect linear scaleup behavior for k (similar to the findings of Dhillon and Modha). Figure 10c shows the scaleup results for an input parameter of $k = p * 80$. We see that for all 4 parallelization schemes, scaleup is nearly linear. This conforms with the measurements of the related work.

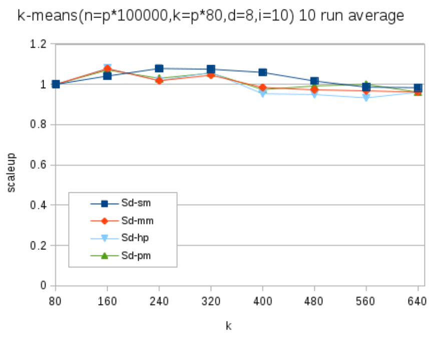
For all three speedup curves we see slightly superlinear scaleup between the single core version and the second measurement using multiple cores. One possible explanation is the runtime overhead generated by garbage collection: When using a single core, garbage collection blocks the algorithm. Using two cores, the system can process input on one core, while the other does garbage collection, leading to slightly better throughput for parallel versions. This hypothesis can be backed by comparing runtime results of the sequential algorithm while using one or multiple cores: For the sequential algorithm, a version that was executed on multiple cores showed better runtime results then a version constrained to one core, albeit no parallel execution was defined in the program (see figure 25 in appendix A.4).



(a) Scaleup results for varying n (running the algorithm with $p*100.000$ elements)



(b) Scaleup results for varying d (running the algorithm with input dimensionality of $p*80$)



(c) Scaleup results for varying k (using $p*80$ clusters)

Figure 10: Scaleup measurements for Lloyds algorithm on a random input data set with varying number of cores

Another property measured by these experiments is the runtime difference between the *hybrid partition* scheme and the *map-maximization* scheme. For speedup as well as scaleup the results show that the runtime difference between the schemes is small. This shows that the overhead of splitting the re-mapping of the data into two functions negligible in the context of the given implementation (Java & Skandium). This observation can be confirmed when examining the relative runtime delta (runtime[sd-mm]−runtime[sd-pm]) which doesn't exceed 1 % of the overall runtime (see figure 11).

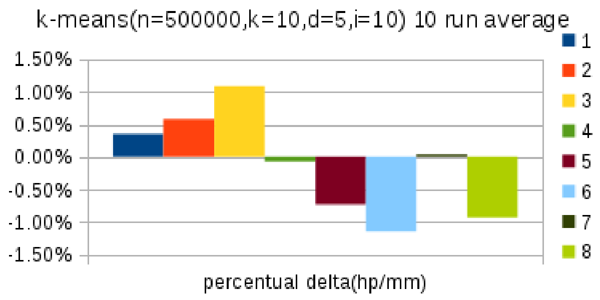
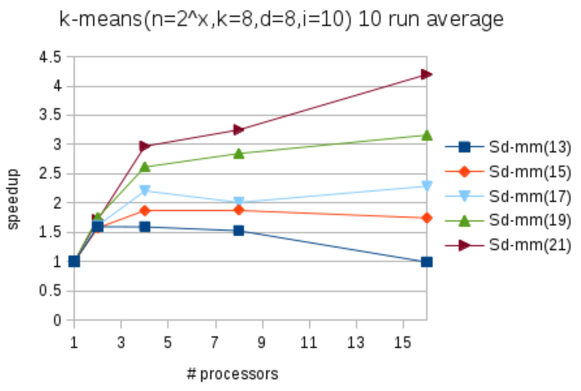


Figure 11: Relative runtime deviation between the *map-maximization* and *hybrid partition* schemes in the speedup measurement runtimes.

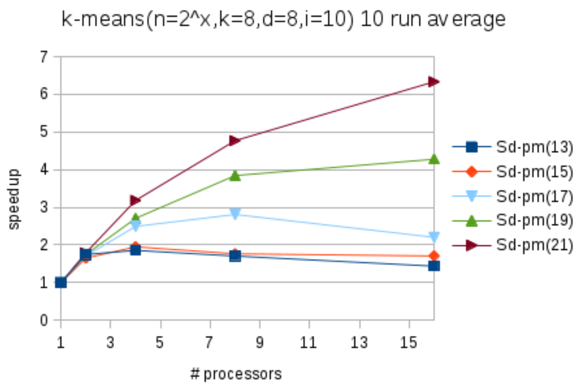
6.2.3. Sizeup Properties

As seen in [28] the speedup properties of k-Means vary with different input data sizes (n). To examine the impact further, multiple runtime measurements, using the random dataset, were conducted for the *map-maximization* and the *partial merge* parallelization schemes: The speedup was measured for $1-16$ processors and for varying amounts of data elements, from $n = 2^{13}$ up to $n = 2^{21}$ elements, while keeping the dimensionality and number of clusters fixed. We expect that for a larger number of elements the achieved speedup is better: the communication and synchronization overhead share of the total runtime is less, if the absolute total runtime is higher.

The results in figure 12 show the impact of the data size on the speedup property: While for a data size greater than 2^{21} speedup is good in both cases, for a data size of 2^{13} and the map maximization scheme, speedup degrades even below the sequential runtime for 16 processors. This shows that the overhead of parallel execution grows with increasing number of processing elements. The communication and synchronization runtime share becomes greater than the computing time on each CPU, leading to a decrease in speedup. This insight shows that due to the overhead, parallel execution should only be considered for larger amount of data. A potential skeleton system for k-Means could use *autotuning* to determine the ideal number of processing elements for a given input problem size and hardware configuration. A possible approach could be the definition of heuristics for the points (as described in the outlook of this thesis in section 7.2).



(a) Map-Maximization Scheme

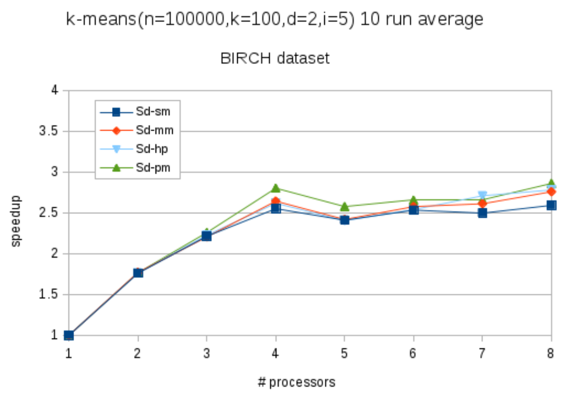


(b) Partial Aggregation Scheme

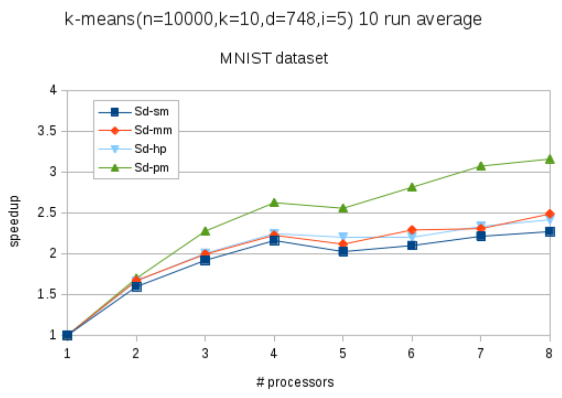
Figure 12: Effects of varying input data size (n) on Speedup factor for Lloyds algorithm

6.2.4. Performance on Real World Datasets

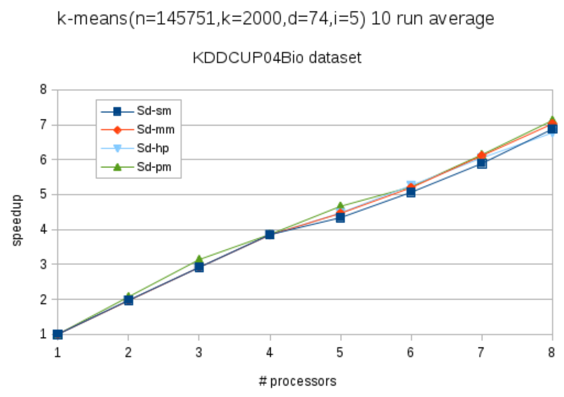
In the next experiment conducted, instead of using random data input and arbitrary numbers of clusters, real world datasets were used (see appendix A.3.1 for dataset-properties). The goal of the experiment was to show influences of dataset properties on the speedup of the algorithms. The datasets used were birch1, MNIST and KDDCUP04Bio. Figure 13 shows the speedup measurements for the given datasets.



(a) Speedup results for Lloyds algorithm (BIRCH dataset)



(b) Speedup results for Lloyds algorithm (MNIST dataset)



(c) Speedup results for Lloyds algorithm (KDD-CUP04Bio dataset)

Figure 13: Speedup measurements of Lloyds algorithm for various datasets

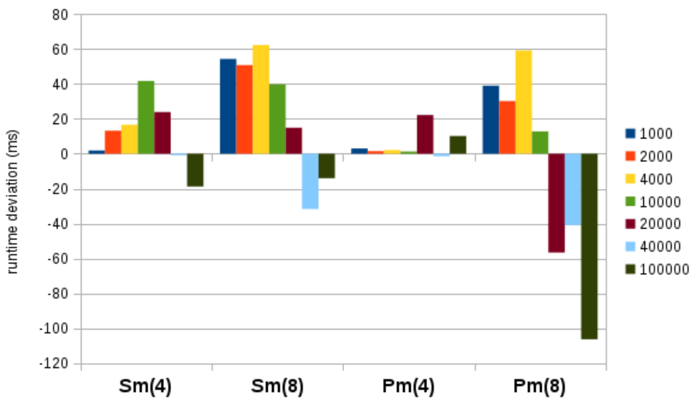
For all three datasets we see a decreased slope when increasing the number of available processors from 4 to 5 (for the BIRCH dataset the impact is higher than for the other two datasets). This performance impact is explainable by the features of the underlying test system (see appendix A.3.3). In the test systems, the computing units are organized in four cores per socket. This leads to higher communication and synchronization times when more than four processors are involved in the system.

Furthermore, we see that the impact of the chosen dataset on the speedup is high, especially for a higher number of available processing cores. This matches the properties shown in the last chapter: for greater input problem size the linear increase of the speedup lasts for a higher number of processors. If we define the problem size S_p as the result of

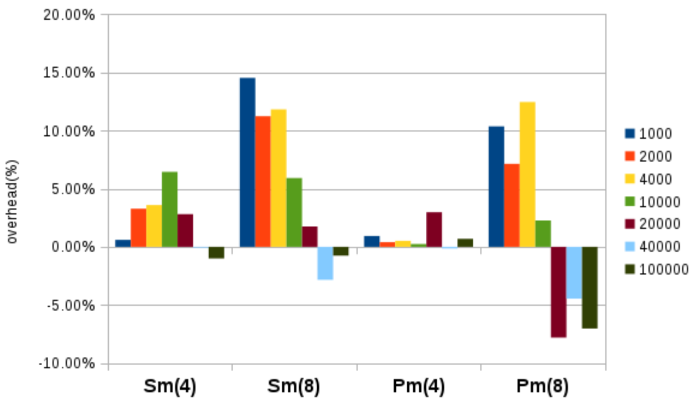
$n*k*d$ we see that for the BIRCH ($S_p = 20*10^6$) and the MNIST ($S_p = 74*10^6$) dataset the speedup deteriorates after 4 processing elements, while the KDDCUP04Bio dataset with a problem size of $21*10^7$ has linear speedup up to 8 processors (the complexity differences can also be seen in the runtime results for the three datasets (shown in appendix A.4)). The effect of the problem size on the speedup behavior is investigated further in section 6.2.6.

6.2.5. Overhead

To compare the computational overhead of using *Skandium* as the algorithmic skeleton framework, manually parallelized versions of the *sequential maximization* and the *partial merge* scheme were implemented using only thread primitives. Figure 14 shows the overhead induced by the *Skandium* framework: Runtime of the manual parallelized implementation is subtracted from the library-based runtime (for each 4 and 8 threads). The graph therefore shows the overhead induced by using *Skandium* in seconds. Negative overhead means a quicker library-based runtime.



(a) total runtime



(b) overhead percentage

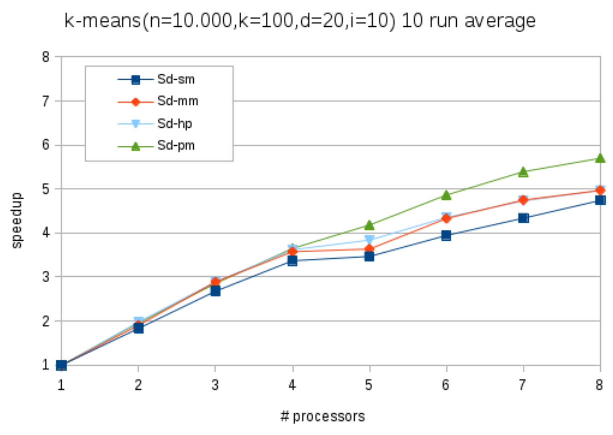
Figure 14: Overhead induced through the use of the skandium library: Deviations between total runtime of manual vs skandium based parallelization (skandium runtime - manual runtime) averaged over 10 runs

The results show that overhead induced is negligible for 4 processors and reaches up to 14.51% (see figure 14b) for 8 processing units and 1.000 input data elements. This shows while the overhead gets greater for a higher number of processing units it doesn't grow with the number of input elements. Instead the results show a negative overhead for input data amount larger than 20.000 elements. This phenomenon remains yet to be explained. Further research has to be conducted in this area: The negative overhead could be in the standard deviation, as sample size of the runs is small (10 runs).

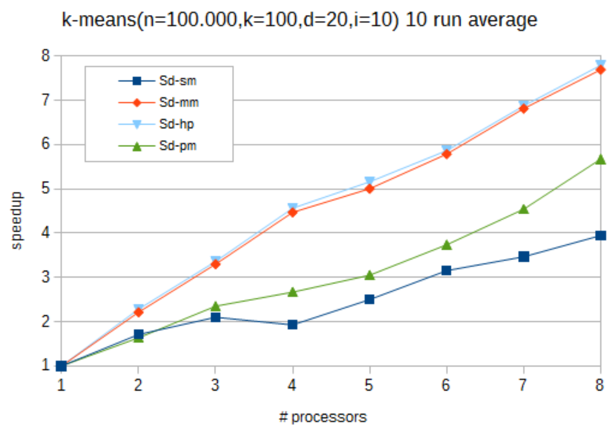
6.2.6. Relation of Speedup and Problem Size

In section 6.2.4 we observed that speedup behavior seems to vary with problem size properties. To confirm this observation, the following experiment using random data was conducted: The speedup of Lloyds algorithm was measured for three datasets: the first represents a smaller problem size of $S_p = 20 * 10^6$, with $n = 10,000$, $d = 20$ and $k = 100$. For this dataset we expect speedup properties similar to the BIRCH and the MNIST datasets. The second and third datasets represent a larger problem size ($S_p = 20 * 10^7$) with the second dataset having a larger number of elements ($n = 100,000$), and the third dataset having a higher dimensionality ($d = 200$). Our hypothesis is that both the second and third dataset show speedup properties similar to the KDDCUP04Bio dataset. In figure 15 we see the speedup results for all three datasets.

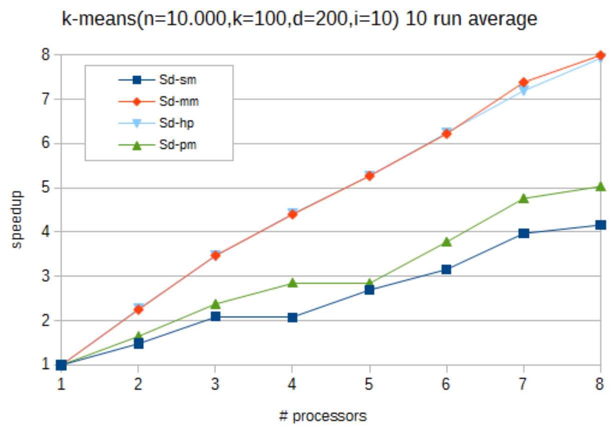
The result show that the speedup is greater for the larger dataset, as assumed. Furthermore, we see that the speedup properties for the second and third dataset are similar, leading to the conclusion that the parts of the problem size can be swapped without major impact on speedup behavior. Furthermore, this suggests using the problem size as a heuristic to determine a trade-off between smaller execution time and needed computing resources.



(a) Speedup results for Lloyds algorithm (smaller costmodel dataset)



(b) Speedup results for Lloyds algorithm (larger costmodel dataset 1)

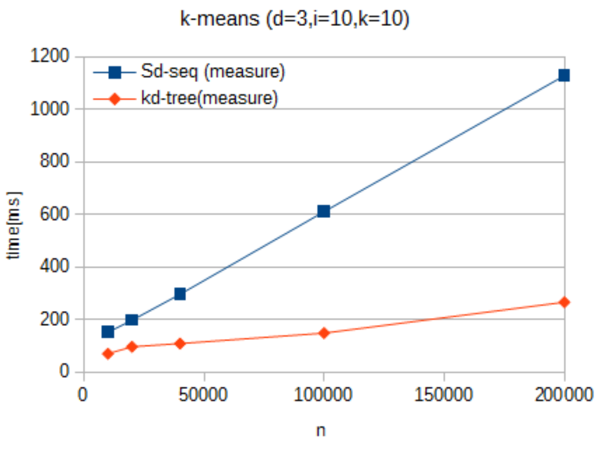


(c) Speedup results for Lloyds algorithm (larger costmodel dataset 2)

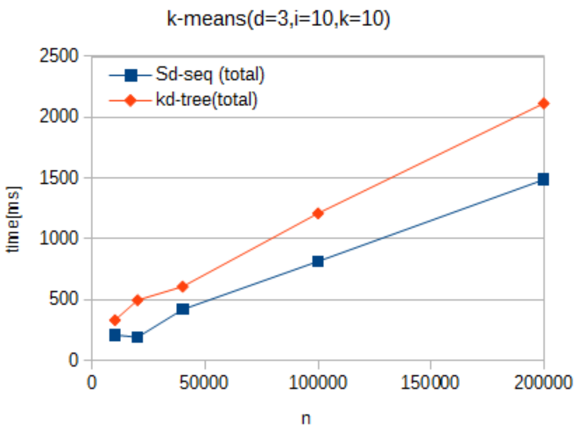
Figure 15: Speedup measurements of Lloyds algorithm for three synthetic datasets with varying problem size

6.3. Algorithmic Variations

After evaluating the properties of the parallelization schemes for a given algorithm (Lloyds algorithm), the next step is to evaluate the influence of using k-Means variations. In the next section we show the applicability of the skeleton and the parallelization schemes to three different variants: The **kd-tree based** k-Means which implements Lloyds algorithm but uses a kd-tree as data layout, the **k-median** algorithm which uses a different distance metric and recalculation logic, and the **fuzzy c-means** algorithm which implements fuzzy clustering. For all versions we measure the achievable speedup for the parallelization schemes usable with the specific algorithm. Furthermore, we compare the speedup results to the findings of the previous section.



(a) Without Preprocessing



(b) With Preprocessing

Figure 16: Runtime Results for the sequential K-Means algorithm in Comparison to the kd-tree based Version (for varying amounts of n)

6.3.1. kd-Tree Based k-Means

To show that the skeleton approach is suitable for exact acceleration optimization, a kd-tree based version of k-Means was realized for the **sequential maximization** scheme. This shows the adaptability of the skeleton approach: by using a *split* function that partitions the input data into p subsets and constructs a kd-tree for each subsets, we can leverage the parallelism provided by the **sequential maximization** scheme without changing the skeleton definition. The corresponding skeleton configuration along with the algorithm used can be seen in algorithm 9 in appendix A.2.3).

To check the validity of the results, a sequential version was implemented first. Figure 16 compares the runtime of the original version and the sequential kd-tree based version. The results show that the kd-tree shows lower runtime for the k-means iteration as shown by Kanungo et al. [61]. When comparing the total runtime of the program, the kd-tree implementation shows higher runtime results than the sequential. This is due to the suboptimal implementation of the kd-tree construction algorithm (which uses a naive median based split logic) which is not the subject of the evaluation.

The speedup results for the random decomposition implemented are shown in figure 17 indicate that the achievable speedup reaches 2.46 when using 7 processing elements. This indicates slightly slower speedup when compared to the *sequential-maximization* scheme applied to the sequential data layout as seen in the previous chapter. One possible explanation for the decreasing speedup could be the memory access pattern generated by the parallel traversal of the generated trees: As the input data is loaded into memory by the master thread before the parallel execution, the data is most likely stored sequentially. The traversal of the kd-tree produces memory access patterns on this data that are harder to predict than the sequential accesses produced by the original version of the algorithm, resulting in a higher cache miss rate.

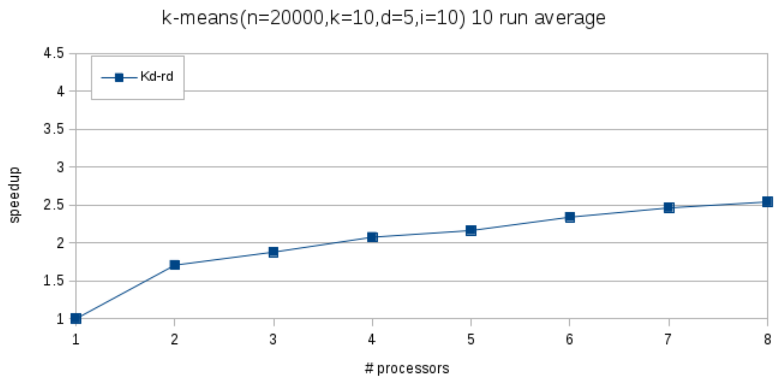


Figure 17: Speedup factor for kd-tree based k-means algorithm, using the *sequential maximization* scheme to provide random decomposition (kd-rd)

6.3.2. k-Median

The changes proposed by the k-median variant (as introduced in section 2.2.3) impose the following variations in the algorithm: In the expectation step the *taxi cab geometry* is used for distance calculations and in the maximization step the *median* for each dimension is used as the new cluster-centroid [15]. The median is function is a holistic aggregation function (see 4.1.2) and therefore parallelization with the partial aggregation scheme is not possible. Instead the parallel *maximization scheme* is chosen (because the median calculation for each cluster is independent). Figure 18 shows the speedup and efficiency of the k-median implementation with the parallel maximization scheme. The results show that for the k-median algorithm achievable speedup for the map-maximization scheme behaves similar to the achieved speedup of Lloyds algorithm (as shown in figure 9). This means while it is not possible to use the partial aggregation scheme, we can still achieve parallelization benefits without changing the skeleton.

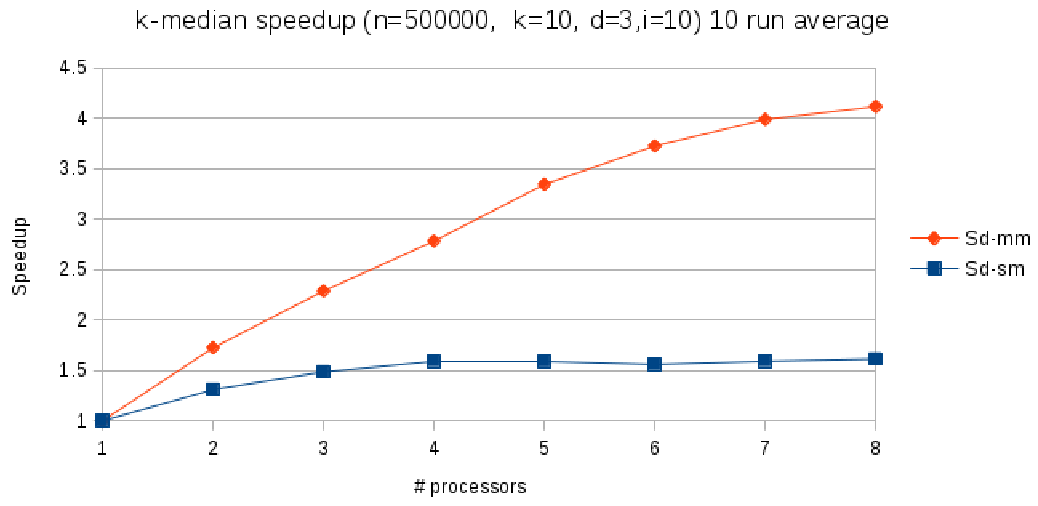


Figure 18: Speedup factor for the k-median algorithm using the parallel maximization parallelization scheme

6.3.3. Fuzzy c-Means

The next examined algorithmic variation is the fuzzy c-means algorithm, as shown in section 2.2.4. Due to the probabilistic nature of the clustering, data dependencies for the distance calculation are higher, making parallelization of the algorithm difficult. Still, the parallel version proposed by Kwok et al. [66] can be expressed using the *sequential maximization* scheme. Due to the higher data dependencies, we expect higher overhead by communication of shared values. The speedup measured, as shown in figure 19 shows that while speedup does not exceed 2.5 the results are similar to the speedup measurements for the *sequential maximization* scheme for Lloyds algorithm. Possible further research could be to search for parallelization approaches enabling the use of the *map maximization* or *partial merge* parallelization schemes.

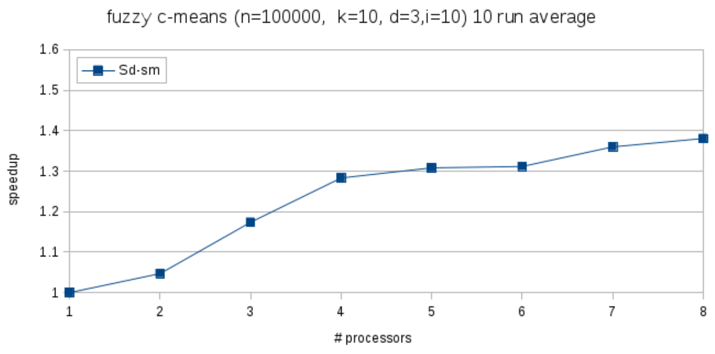
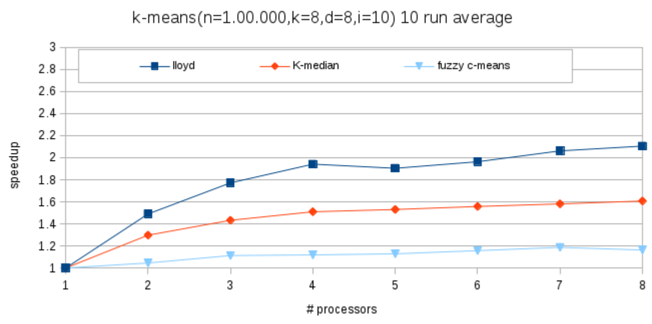


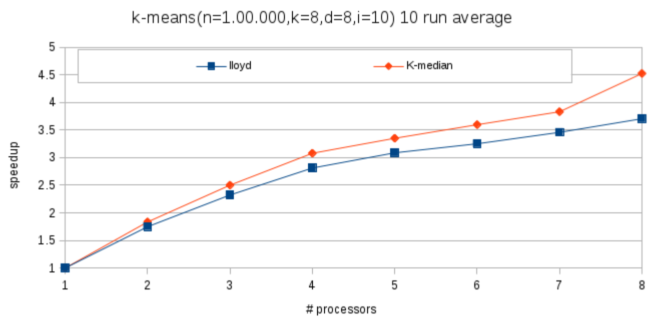
Figure 19: Speedup factors for the fuzzy c-means algorithm using the sequential maximization parallelization scheme

6.4. Speedup Comparisons

In the next experiment, we compare speedup for the 3 algorithms (Lloyd, k-median, fuzzy c-means) for a fixed problem size (random data set with $n = 1.000000, k = 8, d = 8$) with fixed iterations in order to evaluate the behavior of the different parallelization schemes, given different algorithms. Figure 20 shows the results for the sequential and the map maximization scheme. For the sequential maximization scheme we compare the speedup behavior of Lloyds algorithm to k-median and fuccy c-means. The results show that for all algorithms, sequential maximization reaches lower achievable speedup, due to the higher sequential computation amount. Furthermore, we see that the algorithm type has influence on achievable speedup: While the implementation of Lloyds algorithm reaches a speedup of 2.1, the highest speedup of fuzzy c-means is 1.2. This is due to the higher amount of sequential computation time needed by fuzzy c-means: the membership matrix has to be updated for all points, making the maximization step more computationally expensive. For the map maximization scheme we compare behavior of Lloyds algorithm and k-median (because the fuzzy c-means algorithm can not be parallelized in the maximization scheme). We see that the speedup of the k-median performs better, due to the higher runtime share of the maximization step (calculating the median is computationally more expensive than calculating the mean). This shows that when parallelizing the algorithms, the application developer has to take into account sequential amounts of the target algorithm and has to decide which optimizations to use.



(a) Speedup measures for the sequential maximization scheme



(b) Speedup measures for the map maximization scheme

Figure 20: Speedup comparison of the implemented algorithmic variations Lloyd, k-median and fuzzy c-means

6.5. Discussion

In the previous evaluation we have seen that the proposed general k-Means skeleton is applicable to the chosen variations of k-Means both in data layout and algorithmic model. We have validated our assumptions about the theoretical runtime behavior of Lloyds algorithm, recreating the measurements undertaken by Kucukyilmaz in his work

[64]. We have seen that the size of the input data plays a significant role in the speedup achievable by k-Means and that scaling up the processing nodes does not always yield benefits, to the point were runtime exceeds sequential execution (see section 6.2.3). This suggests carefully choosing how and when to parallelize the k-Means algorithm

Various effects on the runtime remain to be explained. Choosing Java as the implementation language for the measurements made reasoning about memory usage and overhead induced by other factors (e.g. garbage collection) difficult. To validate the measurements of this thesis, a goal for further research is the usage of an implementation language with direct memory access (for example using the *Fastflow* skeleton library with the C++ language).

Furthermore, the use of a system not exclusively reserved for the undertaken performance measurements could have an impact on observed phenomenons, although the measurements were spread over different time periods.

Comparing systems with different memory access patterns (different numbers of cores per socket) or memory bandwidth, could show which observed effects can be attributed to memory management and how it affects the overall system behavior.

7. Conclusion

7.1. Summary

This thesis proposed a way to parallelize k-Means based algorithms like Lloyds algorithm, k-Median or fuzzy c-Means using a **structured parallelism** approach named *algorithmic skeletons*. The problem definition of k-Means was shown and classified in the wider field of clustering. Furthermore, variants of k-Means were shown and examined in respect to a general k-Means model. The algorithmic skeleton approach was motivated and explained using the *map*-skeleton example. An overview of different optimization and parallelization approaches to the original k-Means algorithm was given and the different parallelization approaches compared to each other. Furthermore, existing research concerning the structured expression of parallel computation was reviewed and the different techniques were compared to the algorithmic skeleton approach. Given the different variants of k-Means and the restrictions on parallelization approaches, a general k-Means skeleton was introduced, providing a unified interface for the expression of k-Means based algorithms. For this skeleton, different parallelization possibilities were shown and the corresponding skeleton configuration provided. To evaluate the found parallelization schemes, multiple experiments were conducted: The parallelization properties of Llloyds algorithm were examined using all of the given parallelization schemes. The speedup and scaleup metrics were measured for the schemes and the results analyzed. To show the applicability of drop-in improvements of Lloyds algorithm, the kd-tree based variant shown in section 3.2.1 was implemented using the skeleton and parallelization capabilities of the implementation were measured using the speedup metric. Furthermore, using the k-Median and the fuzzy c-Means algorithm, it was shown how the general k-Means skeleton can be applied to algorithmic variations of Lloyds algorithm. For both cases the implementations could be parallelized using the proposed parallelization schemes, therefore providing algorithm speedup without the need to manually implement the synchronization and communication needed for the system. Evaluation showed that although parallelization decreased the overall system runtime, the achievable speedup depends on the problem size, the chosen algorithm and the usable parallelization schemes. We have shown that problem size plays a major role in the speedup achievable for Lloyds algorithm.

7.2. Outlook

This thesis has shown how to leverage algorithmic skeletons to model systems based on the k-Means algorithm. During the evaluation we have seen that speedup properties conform to observations made by related work when implementing Lloyds algorithm. On the other hand, observations were made that need further evaluation to establish guidelines for the usage of the proposed parallelization scheme.

7.2.1. Further Evaluation

Implementation language and Library During the evaluation, multiple runtime deviations were observed which could not be precisely explained with the chosen implementation. The first influence not under control is the **memory access**: Usage of a language with explicit allocation and memory management could show the influence of sequential memory access patterns, padding or cache sensitive programming. Next, due to the **garbage collection** mechanism used by the Java Virtual Machine, runtime overhead is introduced to the system. Using a non-garbage collected language would mitigate the runtime influence of garbage collection processes. Furthermore, the **Just in Time Compilation** mechanism used makes reasoning about runtime effect of different implementation parts difficult and induces overhead if parts of the program are optimized during execution. Using a compiled language would improve the ability to reason about performance impact of different implementation strategies. Therefore further work could use a low level language without the aforementioned limitations and show runtime properties of the parallelization schemes in an environment using explicit memory handling and static compilation.

Runtime Behavior on Distributed Systems The evaluation target architecture for the *Skandium* library and therefore for the evaluation chosen in this thesis was shared memory symmetric multi-processors (multi-core systems). Further research could apply the skeleton to a distributed system context (for example a MPI or middleware based skeleton implementation).

7.2.2. Leveraging the Abstraction

By providing the general k-Means skeleton interface, we can now profit from the clear separation between problem definition and implementation. This enables the following research possibilities:

Autotuning The experiments conducted in this thesis show that achievable speedup depends on the input problem size (see section 6.2.3). This means that for a given algorithm and a problem size, we can determine the number of processors where optimal speedup is achieved for a chosen system/hardware. Further research could implement a program that automatically determines this point and provides feedback to determine an optimal system configuration. More experiments should be conducted aiming to derive heuristics for this configuration.

In the case of Lloyds algorithm, Drake [31] shows that the choice for the optimal *drop-in improvement* of Lloyds algorithm depends on features of the input data (dimensionality, number of elements, natural clustering). Further research could extend the list of available algorithms by parallel k-Means variants and define a system that chooses an algorithm based on information about input data parameters and system properties (for example available computing units). An adaptive k-Means system could choose an algorithm implementation from both linear optimizations and parallelization possibilities to achieve the smallest runtime on a system, depending on system properties like thread creation costs, data access time or RAM size.

In the next step, the system could be applied to variants of k-means, similar to the implementation of the variants presented in this thesis. For each variant, effects of optimization and parallelization could be studied, leading to abstract constraints (like the applicability of partial aggregation) determining the chosen implementation.

Offloading Another way to leverage the abstraction provided by the algorithmic skeletons is to realize a skeleton implementation that relies on *offloading* computation heavy task to special hardware resources like *GPUs* or *coprocessors* (depending on availability of the resources). Furthermore, the overhead generated by this strategy could be examined and heuristics showing the expected runtime could be derived. In the last step, the offloading strategy could be incorporated into the auto-tuning system proposed in the previous section.

7.2.3. Other Data Mining Algorithms

Many algorithms applied in the field of data mining operate on large amounts of data, making parallel approaches desirable. Further research could investigate the structured parallelization of algorithms like *gradient decent* or neural networks. Integration into other machine learning libraries could make the skeleton based versions available to a broader audience.

References

[1] Nir Ailon, Ragesh Jaiswal, and Claire Monteleoni. "Streaming k-means approximation". In: *Advances in Neural Information Processing Systems.* 2009, pp. 10–18.

[2] Marco Aldinucci et al. "Managing Adaptivity in Parallel Systems." In: *FMCO.* Springer. 2011, pp. 199–217.

[3] Daniel Aloise et al. "NP-hardness of Euclidean sum-of-squares clustering". In: *Machine learning* 75.2 (2009), pp. 245–248.

[4] David Arthur, Bodo Manthey, and H Roglin. "k-Means has polynomial smoothed complexity". In: *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on.* IEEE. 2009, pp. 405–414.

[5] David Arthur and Sergei Vassilvitskii. "k-means++: The advantages of careful seeding". In: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics. 2007, pp. 1027–1035.

[6] Krste Asanovic et al. "A view of the parallel computing landscape". In: *Communications of the ACM* 52.10 (2009), pp. 56–67.

[7] Ioannis Assiouras. "A MapReduce Skeleton for Skandium". MA thesis. School of Informatics, University of Edinburgh, 2011.

[8] Eric Backer and Anil K Jain. "A clustering performance measure based on fuzzy set decomposition". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 1 (1981), pp. 66–75.

[9] Bahman Bahmani et al. "Scalable k-means++". In: *Proceedings of the VLDB Endowment* 5.7 (2012), pp. 622–633.

[10] Maria-Florina F Balcan, Steven Ehrlich, and Yingyu Liang. "Distributed k-means and k-median Clustering on General Topologies". In: *Advances in Neural Information Processing Systems.* 2013, pp. 1995–2003.

[11] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[12] James C Bezdek, Robert Ehrlich, and William Full. "FCM: The fuzzy c-means clustering algorithm". In: *Computers & Geosciences* 10.2 (1984), pp. 191–203.

[13] Andrew D Birrell. "An Introduction to Programming with Threads". In: (1989).

[14] Paul S Bradley, Usama M Fayyad, Cory Reina, et al. "Scaling Clustering Algorithms to Large Databases." In: *KDD.* 1998, pp. 9–15.

[15] Paul S Bradley, Olvi L Mangasarian, and W Nick Street. "Clustering via concave minimization". In: *Advances in neural information processing systems* (1997), pp. 368–374.

[16] PS Bradley, KP Bennett, and Ayhan Demiriz. “Constrained k-means clustering”. In: *Microsoft Research, Redmond* (2000), pp. 1–8.

[17] Clay Breshears. *The art of concurrency: A thread monkey’s guide to writing parallel applications.* " O’Reilly Media, Inc.", 2009.

[18] David C Brock and Gordon E Moore. *Understanding Moore’s law: four decades of innovation.* Chemical Heritage Foundation, 2006.

[19] Denis Caromel and Mario Leyton. “Fine tuning algorithmic skeletons”. In: *Euro-Par 2007 Parallel Processing.* Springer, 2007, pp. 72–81.

[20] Rich Caruana, Thorsten Joachims, and Lars Backstrom. “KDD-Cup 2004: results and analysis”. In: *ACM SIGKDD Explorations Newsletter* 6.2 (2004), pp. 95–108.

[21] Cheng Chu et al. “Map-reduce for machine learning on multicore”. In: *Advances in neural information processing systems* 19 (2007), p. 281.

[22] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Cambridge, MA, USA: MIT Press, 1991. ISBN: 0-262-53086-4.

[23] Murray Cole. “Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming”. In: *Parallel computing* 30.3 (2004), pp. 389–406.

[24] Murray I Cole. “Algorithmic skeletons: A structured approach to the management of parallel computation”. PhD thesis. University of Edinburgh, 1988.

[25] Leonardo Dagum and Rameshm Enon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[26] John Darlington and Mike Reeve. “ALICE a multi-processor reduction machine for the parallel evaluation CF applicative languages”. In: *Proceedings of the 1981 conference on Functional programming languages and computer architecture.* ACM. 1981, pp. 65–76.

[27] Sanjoy Dasgupta and Yoav Freund. “Random projection trees for vector quantization”. In: *IEEE Transactions on Information Theory* 55.7 (2009), pp. 3229–3242.

[28] Inderjit S Dhillon and Dharmendra S Modha. “A data-clustering algorithm on distributed memory multiprocessors”. In: *Large-Scale Parallel Data Mining.* Springer, 2000, pp. 245–260.

[29] Edsger W Dijkstra. *Cooperating sequential processes.* Springer, 2002.

[30] Yufei Ding et al. “Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup”. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15).* 2015, pp. 579–587.

[31] Jonathan Drake. “Faster k-means clustering.” PhD thesis. 2013.

[32] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification.* John Wiley & Sons, 2012.

[33] Joseph C Dunn. “A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters”. In: (1973).

[34] Vladimir Estivill-Castro. “Why so many clustering algorithms: a position paper”. In: *ACM SIGKDD explorations newsletter* 4.1 (2002), pp. 65–75.

[35] Benjamin J Evans and Martijn Verburg. *The well-grounded Java developer: Vital techniques of Java 7 and polyglot programming.* Manning Publications Co., 2012.

[36] Reza Farivar et al. “A Parallel Implementation of K-Means Clustering on GPUs.” In: *PDPTA*. Vol. 13. 2. 2008, pp. 212–312.

[37] Edward W Forgy. “Cluster analysis of multivariate data: efficiency versus interpretability of classifications”. In: *Biometrics* 21 (1965), pp. 768–769.

[38] Steven Fortune and James Wyllie. “Parallelism in random access machines”. In: *Proceedings of the tenth annual ACM symposium on Theory of computing.* ACM. 1978, pp. 114–118.

[39] David Gelernter. “Generative communication in Linda”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.1 (1985), pp. 80–112.

[40] Horacio González-Vélez and Mario Leyton. “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers”. In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160.

[41] Attila Gursoy. “Data decomposition for parallel k-means clustering”. In: *Parallel Processing and Applied Mathematics.* Springer, 2004, pp. 241–248.

[42] Attila Gürsoy and Ilker Cengiz. “Parallel pruning for k-means clustering on shared memory architectures”. In: *Euro-Par 2001 Parallel Processing.* Springer, 2001, pp. 321–325.

[43] Ali Hadian and Saeed Shahrivari. “High performance parallel k-means clustering for disk-resident datasets on multi-core CPUs”. In: *The Journal of Supercomputing* 69.2 (2014), pp. 845–863.

[44] Mohammad Hamdan, Greg Michaelson, and Peter King. “A scheme for nesting algorithmic skeletons”. In: *Proceedings of the 10th International Workshop on Implementation of Functional Languages, IFL.* Vol. 98. 1998, pp. 195–212.

[45] Kevin Hammond et al. “The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems”. In: *Formal Methods for Components and Objects.* Springer. 2013, pp. 218–236.

[46] John A Hartigan and Manchek A Wong. “Algorithm AS 136: A k-means clustering algorithm”. In: *Applied statistics* (1979), pp. 100–108.

[47] Charles Antony Richard Hoare. “Communicating sequential processes”. In: *Communications of the ACM* 21.8 (1978), pp. 666–677.

[48] Barbara Hohlt. “Pthread Parallel K-means”. In: *UC Berkeley* (2001).

[49] Bai Hong-Tao et al. “K-means on commodity GPUs with CUDA”. In: *Computer Science and Information Engineering, 2009 WRI World Congress on.* Vol. 3. IEEE. 2009, pp. 651–655.

[50] Paul Hudak and Benjamin Goldberg. “Distributed execution of functional programs using serial combinators”. In: *Computers, IEEE Transactions on* 100.10 (1985), pp. 881–891.

[51] Mary Inaba, Naoki Katoh, and Hiroshi Imai. “Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering”. In: *Proceedings of the tenth annual symposium on Computational geometry.* ACM. 1994, pp. 332–339.

[52] Michael Isard et al. “Dryad: distributed data-parallel programs from sequential building blocks”. In: *ACM SIGOPS Operating Systems Review.* Vol. 41. 3. ACM. 2007, pp. 59–72.

[53] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0-13-022278-X.

[54] Noman Javed and Frédéric Loulergue. “OSL: Optimized bulk synchronous parallel skeletons on distributed arrays”. In: *Advanced Parallel Processing Technologies.* Springer, 2009, pp. 436–451.

[55] Ruoming Jin and Gagan Agrawal. “Combining distributed memory and shared memory parallelization for data mining algorithms”. In: *HPDM: High Performance, Pervasive, and Data Stream Mining 6th International Workshop on High Performance Data Mining: Pervasive and Data Stream Mining (HPDM: PDS’03). In conjunction with Third International SIAM Conference on Data Mining, San Francisco, CA.* 2003.

[56] Ruoming Jin, Anjan Goswami, and Gagan Agrawal. “Fast and exact out-of-core and distributed k-means clustering”. In: *Knowledge and Information Systems* 10.1 (2006), pp. 17–40.

[57] Wesley M Johnston, JR Hanna, and Richard J Millar. “Advances in dataflow programming languages”. In: *ACM Computing Surveys (CSUR)* 36.1 (2004), pp. 1–34.

[58] Manasi N Joshi. “Parallel k-means algorithm on distributed memory multiprocessors”. In: *Computer* 9 (2003).

[59] Sanpawat Kantabutra and Alva L Couch. “Parallel K-means clustering algorithm on NOWs”. In: *NECTEC Technical journal* 1.6 (2000), pp. 243–247.

[60] Tapas Kanungo et al. “A local search approximation algorithm for k-means clustering”. In: *Proceedings of the eighteenth annual symposium on Computational geometry.* ACM. 2002, pp. 10–18.

[61] Tapas Kanungo et al. “An efficient k-means clustering algorithm: Analysis and implementation”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24.7 (2002), pp. 881–892.

[62] Leonard Kaufman and Peter J Rousseeuw. “Partitioning around medoids (program pam)”. In: *Finding groups in data: an introduction to cluster analysis* (1990), pp. 68–125.

[63] Kittisak Kerdprasop and Nittaya Kerdprasop. “A lightweight method to parallel k-means clustering”. In: *International Journal of Mathematics and Computers in Simulation* 4.4 (2010), pp. 144–153.

[64] Tayfun Kucukyilmaz. “Parallel K-Means Algorithm for Shared Memory Multiprocessors”. In: *Journal of Computer and Communications* 2.11 (2014), p. 15.

[65] Vijay P. Kumar and Anshul Gupta. “Analyzing scalability of parallel algorithms and architectures”. In: *Journal of parallel and distributed computing* 22.3 (1994), pp. 379–391.

[66] Terence Kwok et al. “Parallel fuzzy c-means clustering for large data sets”. In: *Euro-Par 2002 Parallel Processing.* Springer, 2002, pp. 365–374.

[67] Jim ZC Lai and Yi-Ching Liaw. “Improvement of the k-means clustering filtering algorithm”. In: *Pattern Recognition* 41.12 (2008), pp. 3677–3681.

[68] Yann LeCun, Corinna Cortes, and Christopher JC Burges. “The MNIST database of handwritten digits, 1998”. In: *Available electronically at http://yann. lecun. com/exdb/mnist* (2012).

[69] Edward A Lee. “The problem with threads”. In: *Computer* 39.5 (2006), pp. 33–42.

[70] Mario Leyton. “Advanced features for algorithmic skeleton programming”. PhD thesis. Nice, 2008.

[71] Mario Leyton and José M Piquer. “Skandium: Multi-core programming with algorithmic skeletons”. In: *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on.* IEEE. 2010, pp. 289–296.

[72] Chen Li et al. “Mux-Kmeans: Multiplex Kmeans for Clustering Large-scale Data Set”. In: *Proceedings of the 5th ACM Workshop on Scientific Cloud Computing.* ScienceCloud ’14. New York, NY, USA: ACM, 2014, pp. 25–32. ISBN: 978-1-4503-2911-8. DOI: 10.1145/2608029.2608033. URL: http://doi.acm.org/10.1145/2608029.2608033.

[73] Qiuhong Li et al. “An efficient K-means clustering algorithm on MapReduce”. In: *Database Systems for Advanced Applications.* Springer. 2014, pp. 357–371.

[74] Stuart P Lloyd. “Least squares quantization in PCM”. In: *Information Theory, IEEE Transactions on* 28.2 (1982), pp. 129–137.

[75] James MacQueen et al. “Some methods for classification and analysis of multivariate observations”. In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability.* Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297.

[76] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. “The planar k-means problem is NP-hard”. In: *Theoretical Computer Science* 442 (2012), pp. 13–21.

[77] Michael D McCool. “Structured parallel programming with deterministic patterns”. In: *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. USENIX Association. 2010, pp. 5–5.

[78] Michael D McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[79] Maged M Michael, Martin T Vechev, and Vijay A Saraswat. *Idempotent work stealing*. Vol. 44. 4. ACM, 2009.

[80] Mark Moir and Nir Shavit. “Concurrent data structures”. In: *Handbook of Data Structures and Applications* (2007), pp. 47–14.

[81] Markus Muhr and Michael Granitzer. “Automatic cluster number selection using a split and merge k-means approach”. In: *Database and Expert Systems Application, 2009. DEXA’09. 20th International Workshop on*. IEEE. 2009, pp. 363–367.

[82] Tan Pang-Ning, Michael Steinbach, Vipin Kumar, et al. “Introduction to data mining”. In: *Library of Congress*. 2006, p. 74.

[83] Hae-Sang Park, Jong-Seok Lee, and Chi-Hyuck Jun. “A K-means-like Algorithm for K-medoids Clustering and Its Performance”. In: *Proceedings of ICCIE* (2006), pp. 102–117.

[84] Dan Pelleg and Andrew Moore. “Accelerating exact k-means algorithms with geometric reasoning”. In: *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 1999, pp. 277–281.

[85] Dan Pelleg, Andrew W Moore, et al. “X-means: Extending K-means with Efficient Estimation of the Number of Clusters.” In: *ICML*. 2000, pp. 727–734.

[86] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[87] SN Tirumala Rao, EV Prasad, and NB Venkateswarlu. “A critical performance study of memory mapping on multi-core processors: An experiment with k-means algorithm with large data mining data sets”. In: *International Journal of Computer Applications* 1.9 (2010), pp. 90–98.

[88] T Hitendra Sarma, P Viswanath, and B Eswara Reddy. “A hybrid approach to speed-up the k-means clustering method”. In: *International Journal of Machine Learning and Cybernetics* 4.2 (2013), pp. 107–117.

[89] Yossi Shiloach and Uzi Vishkin. *Finding the maximum, merging and sorting in a parallel computation model*. Springer, 1981.

[90] Douglas Steinley. “K-means clustering: a half-century synthesis”. In: *British Journal of Mathematical and Statistical Psychology* 59.1 (2006), pp. 1–34.

[91] Kilian Stoffel and Abdelkader Belkoniene. “Parallel k/h-means clustering for large data sets”. In: *Euro-Par’99 Parallel Processing*. Springer, 1999, pp. 1451–1454.

[92] Mu-Chun Su and Chien-Hsing Chou. “A modified version of the K-means algorithm with a distance based on cluster symmetry”. In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 6 (2001), pp. 674–680.

[93] Herb Sutter. “The free lunch is over: A fundamental turn toward concurrency in software”. In: *Dr. Dobb’s journal* 30.3 (2005), pp. 202–210.

[94] Jinlan Tian et al. “Improvement and parallelism of k-means clustering algorithm”. In: *Tsinghua Science & Technology* 10.3 (2005), pp. 277–281.

[95] Kiri Wagstaff et al. “Constrained k-means clustering with background knowledge”. In: *ICML*. Vol. 1. 2001, pp. 577–584.

[96] Fuhui Wu et al. “A Vectorized K-Means Algorithm for Intel Many Integrated Core Architecture”. In: *Advanced Parallel Processing Technologies*. Springer, 2013, pp. 277–294.

[97] Xindong Wu et al. “Top 10 algorithms in data mining”. In: *Knowledge and Information Systems* 14.1 (2008), pp. 1–37.

[98] Han Xiao. “Towards parallel and distributed computing in large-scale data mining: A survey”. In: *Technical University of Munich, Tech. Rep* (2010).

[99] Rui Xu, Donald Wunsch, et al. “Survey of clustering algorithms”. In: *Neural Networks, IEEE Transactions on* 16.3 (2005), pp. 645–678.

[100] Jian Yu and Miin-Shen Yang. “Optimality test for generalized FCM and its application to parameter selection”. In: *Fuzzy Systems, IEEE Transactions on* 13.1 (2005), pp. 164–176.

[101] Matei Zaharia et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[102] Mario Zechner and Michael Granitzer. “Accelerating k-means on the graphics processor via cuda”. In: *Intensive Applications and Services, 2009. INTENSIVE’09. First International Conference on*. IEEE. 2009, pp. 7–15.

[103] Bin Zhang, Meichun Hsu, and Umeshwar Dayal. “K-harmonic means-a data clustering algorithm”. In: *Hewlett-Packard Labs Technical Report HPL-1999-124* (1999).

[104] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. “BIRCH: A new data clustering algorithm and its applications”. In: *Data Mining and Knowledge Discovery* 1.2 (1997), pp. 141–182.

[105] Weizhong Zhao, Huifang Ma, and Qing He. “Parallel k-means clustering based on mapreduce”. In: *Cloud Computing*. Springer, 2009, pp. 674–679.

[106] Xiao-bin Zhi and Jiu-lun Fan. “Some Notes on K-Harmonic Means Clustering Algorithm”. In: *Quantitative Logic and Soft Computing 2010*. Springer, 2010, pp. 375–384.

[107] Esteban Zimanyi. *Advanced Data Warehouse Design: From Conventional to Spatial and Temporal Applications. Data-Centric Systems and Applications.* Springer, 2008.

A. Appendix

A.1. Survey of k-Means Parallelization Approaches

1999 Parallel k/h-means clustering for large data sets [91]	
Algorithm:	Lloyd/Forgy
Target architecture:	distributed, shared nothing, TCP, manual data distribution
Parallelization:	E-Step: p-chunks, M-step: sequential

2000 A data-clustering algorithm on distributed memory multiprocessors [28]	
Algorithm:	Lloyd/Forgy
Target architecture:	distributed, C/MPI
Parallelization:	partial merge: E-Step: p-chunks, M-step: sequential

2000 Parallel K-means clustering algorithm on NOWs [59]	
Algorithm:	Lloyd/Forgy
Target Architecture:	distributed: network of workstations, C/MPI
Parallelization:	k-Chunks

2001, Pthread Parallel K-means [48]	
Algorithm:	Lloyd/Forgy
Target Architecture:	shared memory (SMP), C/pthreads
Parallelization:	E-Step: p-Chunks, M-Step: unknown

2003 Combining distributed memory and shared memory parallelization [55]	
Algorithm:	Lloyd/Forgy
Target Architecture:	hybrid, FREERIDE middleware based
Parallelization:	PARALLELIZATION

2006 Fast and exact out-of-core and distributed k-means clustering [56]	
Algorithm:	drop-in improvement
Target Architecture:	hybrid
Parallelization:	PARALLELIZATION
Notes:	uses sampling

2007 Map-reduce for machine learning on multicore [21]	
Algorithm:	Lloyd/Forgy
Target Architecture:	Hadoop, HDFS
Parallelization:	E-Step: n-chunks (map), M-Step: k-chunks (reduce)

2008 A Parallel Implementation of K-Means Clustering on GPUs [36]	
Algorithm:	Lloyd/Forgy
Target Architecture:	GPUs, CUDA
Parallelization:	E-Step: n-chunks (centroids in thread local storage), M-Step: sequential

2009 K-means on commodity GPUs with CUDA [49]	
Algorithm:	Lloyd/Forgy
Target Architecture:	GPU, CUDA, data in RAM
Parallelization:	E-Step: n-chunks, M-Step: k-chunks (on CPU)

2009 Parallel k-means clustering based on mapreduce [105]	
Algorithm:	Lloyd/Forgy
Target Architecture:	Hadoop, HDFS
Parallelization:	E-Step: n-chunks (map), M-Step: k-chunks (reduce)

2010 A critical performance study of memory mapping on multi-core processors [87]	
Algorithm:	Lloyd/Forgy
Target Architecture:	shared memory, fread/mmap, OpenMP/pthreads
Parallelization:	E-Step: p-chunks

2010 A lightweight method to parallel k-means clustering [63]	
Algorithm:	Lloyd/Forgy
Target Architecture:	hybrid, erlang
Parallelization:	PARALLELIZATION

2014 An efficient K-means clustering algorithm on MapReduce [73]	
Algorithm:	drop-in
Target Architecture:	Hadoop, HDFS
Parallelization:	M-Step: k-chunks (reduce)
Notes:	uses samples and pruning

2014 Parallel K-Means Algorithm for Shared Memory Multiprocessors [64]	
Algorithm:	Lloyd/Forgy
Target Architecture:	shared memory multicore
Parallelization:	E-Step: p-chunks, partial merge

A.2. Skeleton Configurations

A.2.1. k-Median (Sequential Maximization)

Algorithm 7: k-median-sm

```
   Data: data set: $x_1, \ldots, x_n$ with $x_i \in \mathbb{R}^d$
   Input: centroids $c_1, \ldots, c_k$
   Output: List of cluster centroids $c_1, \ldots, c_k$
1  $f_c$ = convergenceCriterion ;
2  $\triangle_e$ = function map (subset_i, centroids)
3      foreach $x_i \in$ subset do
4          $labeledData[i] \leftarrow \Big\{x_p : \Big\|x_p - \text{centroids}[l]\Big\| \leq \Big\|x_p - \text{centroids}[j]\Big\| \; \forall l, j, 1 \leq j \leq k\Big\}$;
5      end
6      return labeledData;
7  $\triangle_m$ = function seq(labeledData)
8      foreach cluster $\in$ labeledData do
9          $newCentroids[i] \leftarrow median(cluster)$;
10     end
11     return newCentroids;
12 skeleton $\leftarrow \triangle_{kmeans}(\triangle_e, \triangle_m, f_c)$ ;
13 result $\leftarrow$ skeleton.input(initial means);
```

A.2.2. k-Median (Map Maximization)

Algorithm 8: k-median-mm

```
Data: data set: $x_1, \dots, x_n$ with $x_i \in \mathbb{R}^d$
Input: centroids $c_1, \dots, c_k$
Output: List of cluster centroids $c_1, \dots, c_k$
1  $f_c$ = convergenceCriterion ;
2  $\triangle_e$ = function map (subset_i, centroids)
3      foreach $x_i \in$ subset do
4          $labeledData[i] \leftarrow \left\{x_p : \left\|x_p - \text{centroids}[l]\right\| \leq \left\|x_p - \text{centroids}[j]\right\| \; \forall l, j, 1 \leq j \leq k\right\}$;
5      end
6      return labeledData;
7  $\triangle_m$ = function seq(labeledData)
8      foreach cluster $\in$ labeledData do
9          $newCentroids[i] \leftarrow median(cluster)$;
10     end
11     return newCentroids;
12 skeleton $\leftarrow \triangle_{kmeans}(\triangle_e, \triangle_m, f_c)$ ;
13 result $\leftarrow$ skeleton.input(initial means);
```

A.2.3. kd-Tree (Random Decomposition)

Algorithm 9: kdtree-random-decomposition

```
Input: KD-Trees for data-subsets: $Kd_1, \dots, Kd_p$
Input: initial means $c_1, \dots, c_k$
Output: List of cluster centroids $c_1, \dots, c_k$
$f_c$ = convergenceCriterion ;
$\triangle_e$ = function map($Kd_i$, centroids)
    localCandidateSet ← new CandidateSet(centroids);
    Filter($Kd_i$,localCandidateSet);
    return localCandidateSet;
$\triangle_m$ = function seq(localCandidateSets)
    CandidateSet aggregated ← new CandidateSet();
    foreach $Z \in localCandidateSets$ do
        foreach $z \in Z$ do
            $aggregated[z].wgtCent \leftarrow z.wgtCent$;
            $aggregated[z].count \leftarrow z.count$ ;
        end
    end
    foreach $c \in centroids$ do
        $c \leftarrow aggregated[c].wgtCent \div aggregated[c].count$;
    end
    return centroids;
skeleton ← $\triangle_{kmeans}(\triangle_e, \triangle_m, f_c)$ ;
result ← skeleton.input(initial means);
```

A.2.4. Fuzzy c-Means (Sequential Maximization)

Algorithm 10: fcm-sm

```
Data: data set: $x_1, \dots, x_n$ with $x_i \in \mathbb{R}^d$
Input: centroids $c_1, \dots, c_k$
Input: fuzziness index: fi
Output: List of cluster centroids $c_1, \dots, c_k$
$f_c$ = convergenceCriterion ;
$\triangle_e$ = function map (subset_i, centroids, membershipMatrix)
    PartialResult partialResult ← new PartialResult();
    foreach $c_k \in centroids$ do
        foreach $x_i \in subset$ do
            $partialResult[k].vector \leftarrow membershipMatrix[k][i]^{fi} * x_i$;
            $partialResult[k].count \leftarrow membershipMatrix[k][i]^{fi}$;
        end
    end
    return partialResult;
$\triangle_m$ = function seq(partialResults)
    PartialResult aggregated ← new PartialResult();
    foreach $Z \in localCandidateSets$ do
        foreach $z \in Z$ do
            $aggregated[z].vector \leftarrow z.vector$;
            $aggregated[z].count \leftarrow z.count$ ;
        end
    end
    foreach $c \in centroids$ do
        $c \leftarrow aggregated[c].vector \div aggregated[c].count$;
    end
    return centroids;
skeleton ← $\triangle_{kmeans}(\triangle_e, \triangle_m, f_c)$ ;
result ← skeleton.input(initial means);
```

A.3. Performance Measurement Methodology

A.3.1. Data Sets

- birch (n=100,000, d=2): Generated Data with a fixed set of 100 clusters, taken from: [104]
- kddcup04(n=139,658, d=74): Test data for the protein homology task taken from the KDDCup 2004 [20]
- mnist784(n= 10,000, d=784): The MNIST database of handwritten digits, from [68]

A.3.2. Measurement Script

```
#!/bin/bash

JAVA_EXECUTABLE="/usr/lib/jvm/jre1.8.0_65/bin/java"
#JAVA_EXECUTABLE="java"

jarName="skandium.jar"

n=100000;
d=8;
k=100;
iterations=5;
startCpuNumber=16;
endCpuNumber=31;

input="/dev/shm/randomPoints-d30-n1000000.csv";
output="measurement-results.db";


for repetition in `seq 1 10`;
do
 for projectName in "sd-sm" "sd-mm" "sd-hp" "sd-pm";
  do
   for i in `seq -s' ' $startCpuNumber $endCpuNumber`;
    do
    threadOffset=$[$startCpuNumber -1];
    threads=$[$i - $threadOffset];
    tasksetParam="$startCpuNumber-$i"
    tasksetCommand="taskset␣-ca␣$tasksetParam"
    #<flavour> <n> <k> <d> <i> <threads> <partitions> <taskset>
    params="-in␣$input␣-out␣$output␣-live␣-f␣$projectName␣-n␣$n
␣␣␣␣␣-k␣$k␣-d␣$d␣-i␣$iterations␣-p␣$threads␣-t␣$tasksetParam"
    runCommand="nice␣-n␣10␣$tasksetCommand
␣␣␣␣␣$JAVA_EXECUTABLE␣-jar␣$jarName␣$params"
    echo $runCommand
    $runCommand
    done
   done
done
```

A.3.3. System Specifications

	Hardware
CPU	Quad-Core AMD Opteron(tm) 8384
Clock rate	2,70 GHz
Cores	32
Cores/Socket	4
L1(D) Cache	64 KB
L2 Cache	512 KB
L3 Cache	6 MB
RAM	64619 MB
	Software
Operating System	Linux 3.11.10-29-desktop (SUSE Linux)
Java	java version "1.8.0_45"
JRE	Java(TM) SE Runtime Environment (build 1.8.0_45-b17)
JVM	Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)
taskset	taskset from util-linux 2.23.2

A.3.4. Source Code

The source code of the tested system is provided on the attached DVD in the folder */source*. The root java-package for the implementations is
cl.niclabs.skandium.examples.kmeans.
The algorithms used in the measurements can be found in the following sub-packages:

1. Lloyds algorithm (subpackage: *lloyd*)
   - Lloyds algorithm: sequential implementation with static data (sd-seq): *sequential.SDSeqKmeans*
   - Lloyds algorithm: sequential maximization scheme (sd-sm): *sequentialmaximization*
   - Lloyds algorithm: map/parallel maximization scheme (sd-mm): *mapmaximization*
   - Lloyds algorithm: hybrid partition scheme (sd-hp): *hybridpartition*
   - Lloyds algorithm: partial merge scheme (sd-pm): *partialmerge*
2. kd-tree optimization: *treebased*
   - sequential implementation (kd-tree): *sequential*
   - random decomposition (kd-rd): *randomdecomposition*
3. k-Medians: *kmedian*

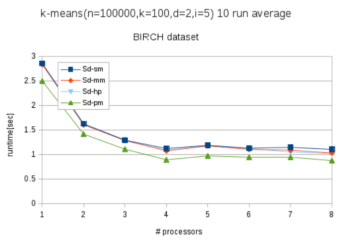
- sequential implementation: *SDSeqKmedian*
- sequential maximization (kmd-sd-sm): *SDSMKmedian*
- map-maximization (kmd-sd-mm): *SDMMKmedian*

4. fuzzy c-means: *cmeans*
    - sequential implementation: *SDSeqcmeans*
    - sequential maximization: *SDSMFCMeans*

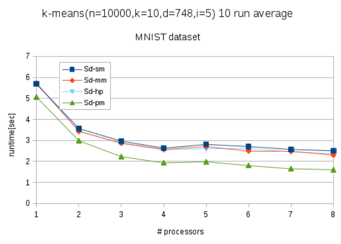
A.3.5. Raw Runtime Data

The runtime measurement data is provided on the attached DVD in the folder */evaluation*.

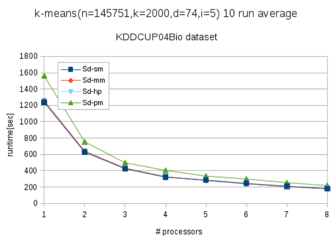
A.4. Runtime Plots



(a) Runtime for Lloyds algorithm (BIRCH dataset)

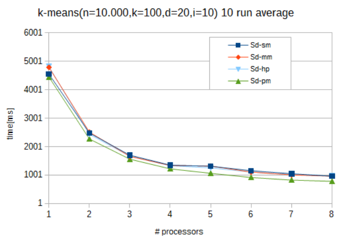


(b) Runtime for Lloyds algorithm (MNIST dataset)

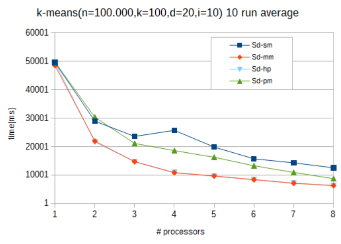


(c) Runtime for Lloyds algorithm (KDDCUP04Bio dataset)

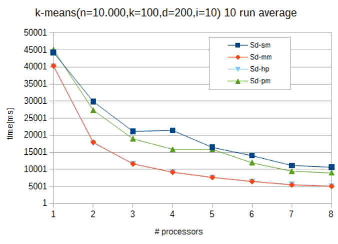
Figure 21: Runtime measurements of Lloyds algorithm for various datasets



(a) Runtime for Lloyds algorithm (Costmodel dataset 1)



(b) Runtime for Lloyds algorithm (Costmodel dataset 2)



(c) Runtime for Lloyds algorithm (Costmodel dataset 3)

Figure 22: Runtime measurements of Lloyds algorithm for various datasets

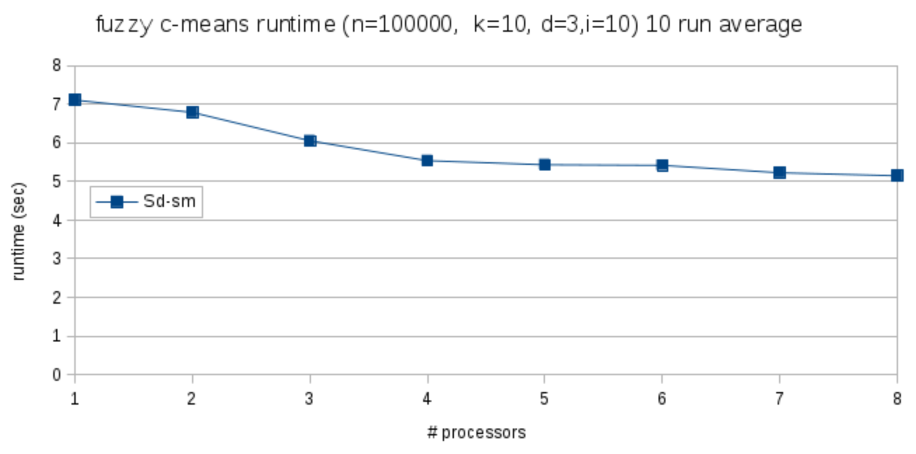


Figure 23: runtime measurements for the fuzzy c-means algorithm using the sequential maximization parallelization scheme

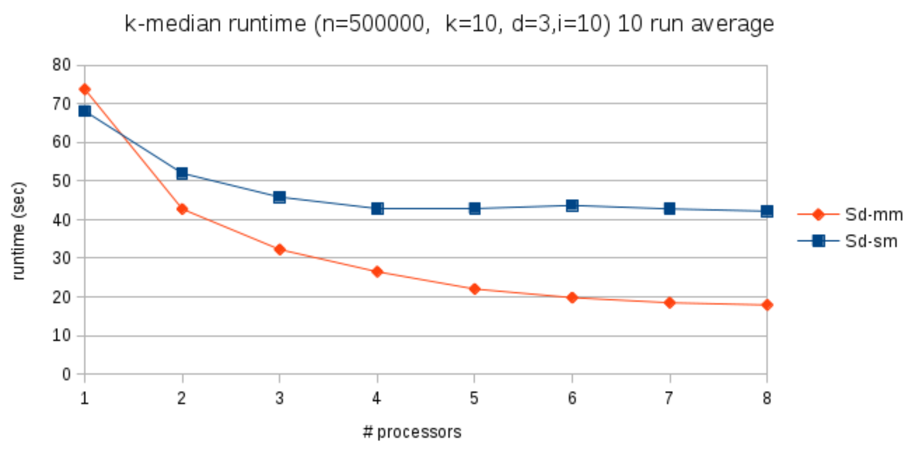


Figure 24: Runtime measurements for the k-median algorithm using the map maximization parallelization scheme

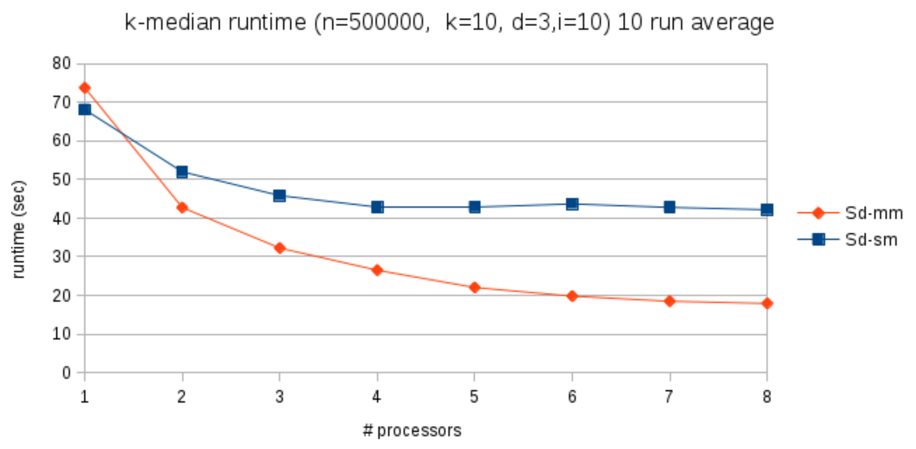


Figure 25: Runtime comparison of the sequential algorithm using one and eight cores available to the system

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den April 13, 2016 ....