Karlsruher Institut für Technologie

Bachelor thesis

K-Means in a Malleable Distributed Environment

Michael Dörr

Date: September 14, 2022

Supervisors: Prof. Dr. Peter Sanders
M.Sc. Dominik Schreiber

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, September 14, 2022

. . . .
(Michael Dörr)

Abstract

K-Means is a NP-hard problem where many data points have to be assigned to clusters. The centres of these clusters have to be arranged in a way that the squared distance of every data points to its nearest center in sum is minimized. The classic approximation algorithm that finds a local minimum is Lloyd's algorithm. Lloyd's algorithm alternates in between calculating data point assignments and optimising cluster centers. With each iteration, the resulting cumulated distance gets smaller. We design a parallel, malleable version of Lloyds algorithm with usage of the framework Mallob. For that, we make usage of the binary tree design of Mallob's inter worker communication. By using a work queue we address the problem of workers that leave the job mid computation. In our tests we show a good scaling of our application with sufficient big K-Means instances on machines with up to 128 cores. To go even further, we use these results to improve the performance of our application when scheduled in a scenario of concurrent jobs. We do that by designing a step function that sets an upper bound to the demand of workers for a job.

Zusammenfassung

K-Means ist ein NP-schweres Problem bei dem viele Daten-Punkte zu Clustern zugewiesen werden müssen. Die Zentren dieser Cluster müssen so angeordnet sein, dass die quadrierte Distanz jedes Daten-Punktes zu seinem nächsten Zentrum in Summe minimiert wird. Der klassische Approximationsalgorithmus, welcher ein lokales Minimum findet, ist Lloyds Algorithmus. Lloyds Algorithmus alterniert zwischen dem Berechnen von Daten-Punkt Zuweisungen und dem Optimieren der Cluster-Zentren. Mit jeder Iteration des Algorithmuses wird die kumulierte Distanz geringer. Wir designen eine parallele, formbare Version von Lloyds Algorithmus mit Hilfe des Frameworks Mallob. Dafür nutzen wir das Binärbaum Design Mallobs für die Kommunikation zwischen Arbeitern. Durch die Nutzung einer Arbeitswarteschlange beheben wir Probleme, die Arbeiter durch das verlassen des Jobs mitten in der Berechnung auslösen. In unseren Tests zeigen wir, dass unsere Anwendung mit hinreichend großen K-Means Instanzen gut auf Maschinen mit bis zu 128 Kernen skaliert. Weitergehend nutzen wir diese Ergebnisse um die Leistung unserer Anwendung noch weiter zu verbessern, wenn sie in Mallob gleichzeitig mit anderen Jobs läuft. Wir erreichen das dadurch, dass wir eine Treppenfunktion designen, die eine obere Schranke für die Nachfrage an Arbeitern für einen Job festlegt.

Contents

1 Introduction

In this chapter, we will describe what our motivation is based on and mention our contribution.

1.1 Motivation

The demand of automatic data analysis in commercial and scientific research is as high as ever. With the growth of the digital market and monitoring possibilities, the resulting data sets continue to grow [9, p.651]. In the meantime, single core performance growth got slower, therefore parallel computing is getting more important [3, p.10][15]. To handle this development well scaling clustering algorithms like K-Means must be used at some point. A common way to handle parallel computable tasks is the usage of distributed computing. Usually, users and operators of such a distributed computing resource do not only want to get fast results by any matter. Efficient usage of computation time is very important considering the high costs of operating computing nodes. With parallel algorithms, computation jobs do not only vary in computation time and priority, making it a classic scheduling problem [14, p.7]. They also vary in efficient parallelizability per job. Malleability is a promising paradigm to optimize efficiency in this scenario. A parallel task is malleable if it can adapt to a varying number of workers during execution. To create an efficient data analysis application we will use the Mallob framework for distributed, malleable computing [16]. With this framework we will design a malleable version of Lloyd's algorithm, the classic algorithm to approximate K-Means solutions[11].

1.2 Contribution

We will focus on the design, implementation and evaluation of a malleable K-Means application for the award winning framework Mallob [16][17][18][19]. We design the application to be moldable first, i.e., the number of workers is user-defined at program start and then fixed throughout the computation. Then we continue with addressing and solving problems that occur with this design in a malleable environment. We provide a C++ implementation. At the end, we show that computation of concurrent jobs can be finished faster by setting a upper bound for the number of workers of each job individually. For that, we show our approach to model a step function for that upper bound and our test results.

1.3 Structure of Thesis

In chapter 2, we have a look at the K-Means Problem and its practical usage. Also, we mention some advanced speedup and parallelization techniques for Lloyd's algorithm[11] and explain the main idea behind them. In chapter 3 we introduce the Mallob framework and its application interface. Based on that, we design an architecture for a new malleable K-Means application we implement with Mallob in chapter 4. In the last chapter 5 we show the results of our experiments and our observations towards upper bounds for concurrent workers and the impact of the calculated upper bound on runtimes of concurrent jobs.

2 Preliminaries

In this chapter, we will have a introduction into K-Means. Also, we will have a look at some techniques to speed up different parts of the algorithm and describe parallelization approaches.

2.1 K-Means

K-Means is a NP-hard optimization problem [4]. We are given n d-dimensional data points X and an integer k. The goal is to find k cluster centers C such that the sum of the quadratic distance of each data point $X_j \in X$ to its nearest cluster center is minimized. This equals minimizing the following function:

$$J = \sum_{i=1}^{k} \sum_{j=1}^{n} \|X_j - C_{m[j]}\|^2 \tag{2.1.1}$$

with $m[j]$ as an array that saves the id i of the cluster center nearest to data point j.
A well known algorithm to approximate a local minimum is Lloyd's algorithm 1. The minimum found by it, is dependent on the initial cluster centers [10].

Algorithm 1: Lloyd's algorithm[11]

```
Input: $X, k$
Output: $C$
/* start with random cluster centers                          */
$C \leftarrow$ initStartCenters(X)
while cluster centers changed positions do
    /* at least one iteration                                 */
    assign each $X_j$ to its nearest cluster as member
    calculate new cluster centers based on mean of all members
end
```

Lloyd's algorithm is often also called *K-Means algorithm* because it is the standard algorithm to compute a good approximation for the *K-Means problem* and the exact solution often is not needed. K-Means is often mentioned as example for unsupervised machine

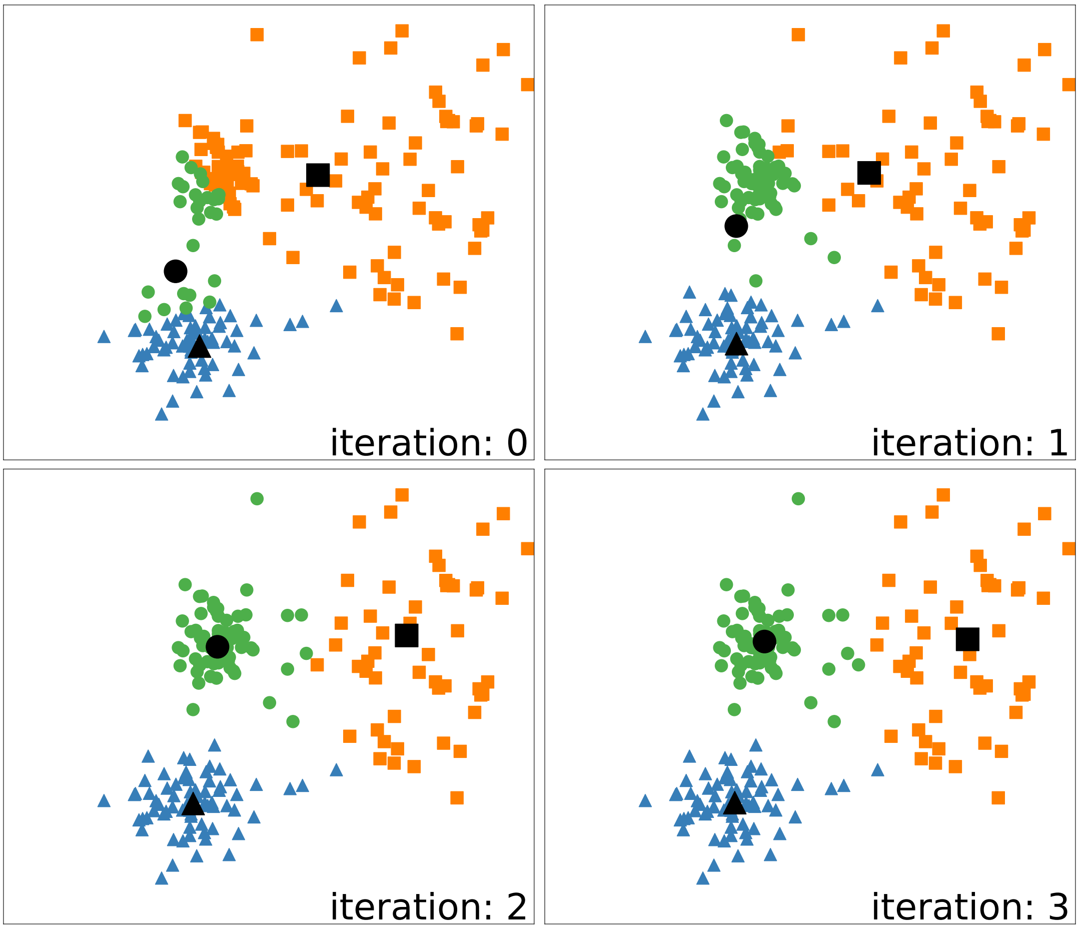


Figure 2.1: Clustering progress of a K-Means instance with $k = 3$, $d = 2$ and $n = 200$. The black points are the cluster centers

learning. Its main usage is data clustering. The result can be used in many different ways. For example, it is possible to reduce the size of a data set with too many data points for another machine learning algorithm and contain a better representation than randomly deleting points[21]. Alternative clustering methods are DBSCAN, OPTICS, Birch and Gaussian Mixture 2.2.

2.2 Advanced K-Means Techniques

Lloyd's algorithm consists of three main parts: set the start cluster centers, assign each data point to a cluster, calculate the new cluster centers. Different speedup approaches mainly focus only on one of the first two parts. *K-Means++*[1] is a method to increase the probability of selecting good start cluster centers. Using the *triangle inequality* [6] or *mini batch*

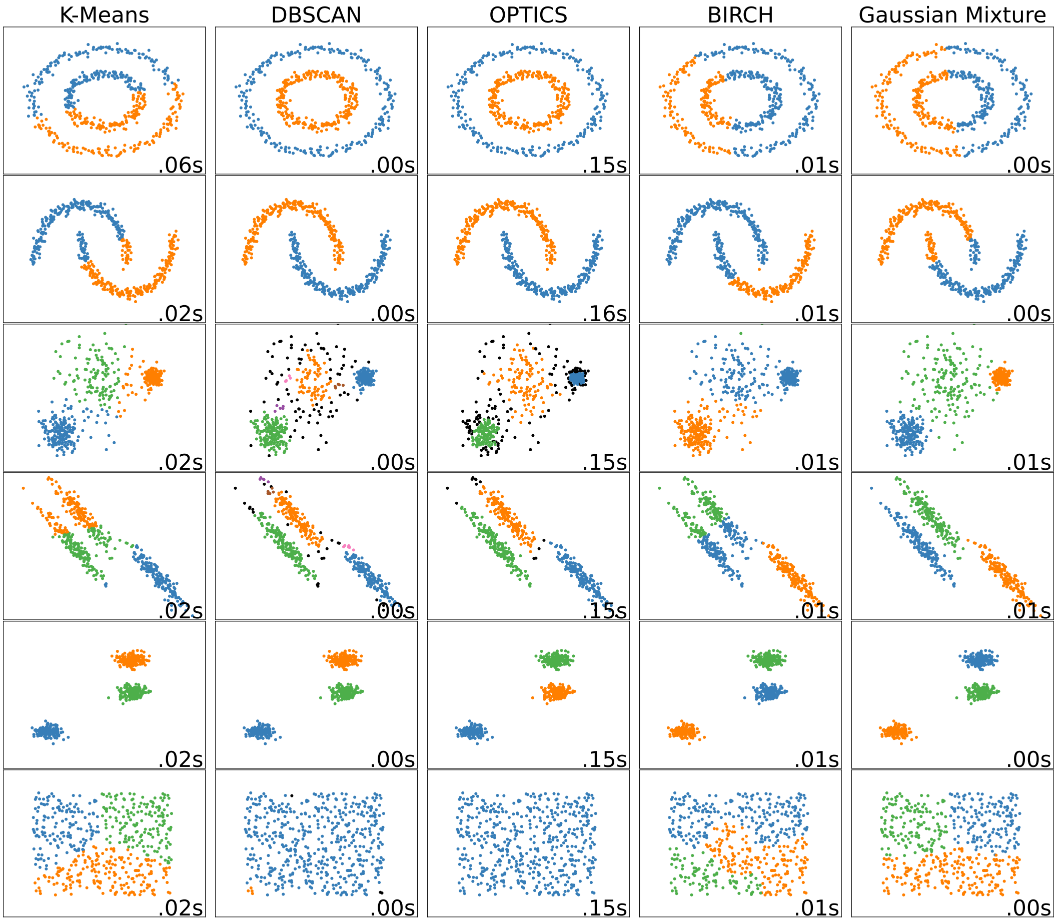


Figure 2.2: Clustering results for K-Means, DBSCAN, OPTICS, BIRCH and Gaussian Mixture. The image is generated with the source code from: https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html

K-Means [20] can increase computation speed for computing $m[j]$.
The original Lloyd's algorithm uses completely random chosen k points of the X to initiate C. Of course, better start centers can have a huge impact on the count of iterations that have to be done by the algorithm. For example, if we set the start centers to a local minimum, only one iteration has to be done because the centers would not change. Theoretically, the computation of the start centers can be arbitrarily costly. Still, it should stay fast because this can result in a new algorithm. *K-Means++* usually generates start cluster centers that require fewer iterations to reach a local minimum. Due to the way probabilities are assigned to the data points, the selected centers are more likely to be further away from each other. Algorithm 2 shows an example implementation of K-Means++. Function $\Delta(p_1, p_2) \rightarrow \mathbb{R}$ is used to calculate the euclidean distance between two points $p1$ and $p2$.
To speed up the computation of the nearest center of a data point, making usage of the tri-

Algorithm 2: K-Means++

```
Input: $X, k$
Output: $C$
$C_0 \leftarrow$ random $X_j \in X$
$i = 1$
while $i < k$ do
    $m[j] \leftarrow [i \,|\, min{=}\Delta(X_j, C_i)]$
    initiate $C_i$ as $X_j \in X$ with probability $\frac{\Delta(X_j, C_{m[j]})^2}{\sum_{X_j \in X} \Delta(X_j, C_{m[j]})^2}$
    i++
end
```

angle inequality can be helpful if the computation of distances between points costs much time. That is usually the case if the data is high dimensional.

Elkan [6] describes an algorithm that has the same cluster centers after each iteration as Lloyd's algorithm for a given data set and similar start cluster centers. The main idea bases on two Lemmas:

Lemma 1 [6]: Let X_j be a point and let b and c be centers. If $\Delta(b, c) \geq 2\Delta(X_j, b)$ then $\Delta(X_j, c) \geq \Delta(X_j, b)$

Lemma 2 [6]: Let X_j be a point and let b and c be centers. Then applies $\Delta(X_j, c) \geq \max\{0, \Delta(X_j, b)$ - $\Delta(b, c)\}$

To use Lemma one, it is necessary to compute all the pairwise distances of the cluster centers. Afterwards, these distances can be reused for every data point. Also, the data points already have to be member of a cluster. The cluster center of the currently looked at data point will be b and all further centers will be c one by one. Because probably many X_j will stay very close to b, $\Delta(b, c) \geq 2\Delta(X_j, b)$ will often be true and no distance calculation is needed. Sometimes distance calculations still are not avoidable, but calculated distances shall be used as long as possible. That is why the distance of any point X_j to its nearest center is stored when once calculated. With Lemma two it is possible to calculate an upper bound of how far away the nearest center of X_j has possibly traveled away from X_j over the last few iterations. This upper bound can be used for the distance of X_j to its center in Lemma one to avoid even more distance calculations. In his paper [6] Elkan describes his algorithm in detail.

Another algorithm to speedup iterations is mini batch K-Means. The main Idea of it is to use a batch of q randomly selected data points for each iteration instead of using all points. With this approach, the centers will probably never reach a local minimum because of the random selection of data points each iteration. Nevertheless Sculley [20] showed it will come pretty close to one. That is why the original mini batch algorithm gets initiated with a fixed number of how many iterations shall be done. Scikit-learn [13] provides implementation of Lloyd's algorithm [11], Elkan's algorithm [6], Mini-Batch K-Means Sculley [20] and start center selection with [1].

2.3 Parallel K-Means

In this section, we discuss three ideas of how to split up the calculations made in one iteration. The initialization of the random start centers is at low cost, so we will not parallelize that. Also, we will assume, that every *processing element* (PE) has copies of all data points available. Three possible parallelization strategies are to split up computation at k, d and n [22][5][23].

If we try to split up computation at k we have a lot of synchronization to do. The computation of the nearest center of a data point starts with each PE calculating the distance from one center to all points. So the next step has to be a reduce operation to compare all the distances for each data point and determine the nearest center. The resulting nearest center assignments have to be broadcast to all PEs. Afterwards, every PE can compute its new center by iterating over all points again. Now we have to synchronize all PEs once every iteration. But the size of the data that has to be shared is pretty big and every entry in the center assignment array has to be looked at by every PE. Also, the maximum count of PEs we can use is limited by k, a number that is sometimes below ten. This approach will not scale well. Probably that is the reason why very little research is done in that direction.

If we split up computation dimension wise like the approach in [5], PEs are again limited by the sometimes low dimension. In practice data often is very high dimensional, so we will still look at how this approach looks like. To compute the nearest centers every PE at first has to sum up the distance squared of the assigned dimensions for each data point and cluster. Then all final distances can be calculated in a reduction. The nearest center of a data point cannot be determined until the reduction is finished. Next, the PEs need the resulting center assignment. For that, PE 0 can compare all distances, set the nearest cluster centers and broadcast them. Otherwise, every PE would have to compare all distances by itself. After the array of center assignations is broadcast to every PE each PE can calculate the final value of its assigned dimensions. This result can be used for the next iteration. With this approach, we still have a big sequential part at the comparison of distances.

Our final approach will be to divide the data points into p slices, each one assigned to an PE. This is similar to the approach in [23]. An PE calculates the nearest centers for each data point of its slice in sequential and saves it in $m[j]$. For later use, also the sum of how many data points are assigned to each center i has to be stored in an array $\mu[i]$. Now the center assignations are computed, but so far, no synchronization was needed in contrast to the first two approaches. To compute the new centers at first, each PE computes new *partial centers* based on its data slice, as if there were no other data points. The resulting partial centers will be reduced to real centers. To compute the real centers the partial centers get weighted in proportion to their value $\mu[i]$. Therefore, not only the partial centers must be sent in the reduction, but also μ and μ must also be summed up at reduction. The resulting centers have to be broadcast and the next iteration can start. With this approach, we must synchronize only once per iteration. Also, we are not limited in how many cores we can use at most. That is the case because if n is very low, there is not much work to do anyway.

k is limited by the count of data points because more clusters than data points do not have a meaningful use. If the count of centers even comes near to the count of data points, only few iterations are needed to reach a local minimum.
For our malleable application we design in chapter 4, we use the parallel approach of splitting computation up at n. With the least costly sequential part and low limitation in maximum PE count, we expect the best scaling results.

3 The Framework Mallob

In this chapter, we will describe the most important details about Mallob, that must be known to implement an application for it. Mallobs source code is available at https://github.com/domschrei/mallob.

3.1 Scheduling

Mallob is a scheduling platform for solving malleable NP hard jobs with unknown processing times. When talking about *malleability* we will use following definitions: „A *rigid* task requires a fixed number of workers. A *moldable* task can be scaled to a number of workers at the time of its scheduling but then remains rigid. Finally, a *malleable* task is able to adapt to a fluctuating number of workers *during* its execution“ [16, p.120] [7].
Before we look at the internal structure of Mallob we should talk about its usage. Mallob has 3 different main operation modes: mono, orchestrated and on demand. When using mono application mode, Mallob will execute one singe job by a given job description at start and terminates after that. Orchestrated application mode takes a job description template file with paths to data files, a job template with additional information like the application type and a client template describing how to randomize job parameters when introducing new jobs to the system. The on demand application mode offers a job introduction interface, that lets us introduce new jobs into a running Mallob instance. Our new job will be scheduled within a tenths of milliseconds and will receive at least one worker. If there are already some other active jobs in the System, they may already use as many workers as PEs. In that case, at least one worker of a currently active job will get suspended by its PE like in 3.1 and a worker of our job is assigned to this PE. The PE can store up to five suspended workers and resume them later. Depending on the priority of our job, the number of workers scales proportionally to the other active jobs. The maximum number of workers a job can currently reasonably employ is capped by the jobs *maximum demand*. Our job probably gets more than one worker, but at first, a job always starts with one worker. After a short period of time, more workers can join the job.

Now we should describe the behavior of a worker in detail. A *worker* is an instance of an application class derived from the job class. It knows the number of workers associated to this job w and its own job index ξ.
Due to the derivation from the job class, a worker is bound to one job and has to be able to

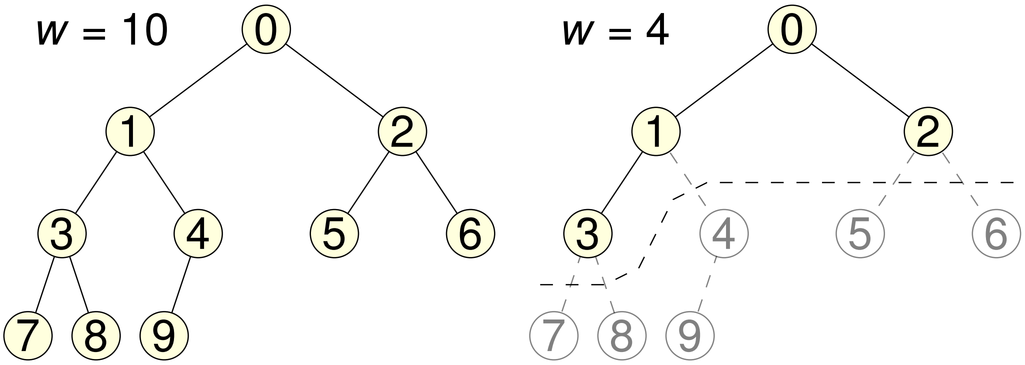


Figure 3.1: On the left side is a job tree with ten workers. Communication may only take place on the edges. The neighbourhood of a worker are the up to three workers that can be reached over these edges. On the right side we can see that workers $[4, 9]$ are suspended if the w shrinks from ten to four. Picture source: [16]

solve this job alone. To do so, every worker gets access to a process wide thread pool and must put expensive work into it. That is the case because Mallob expects the main thread to return fast after a function call of a job class method to perform scheduling decisions and receive messages from other workers. To avoid communication overload on one worker, workers are logically arranged in a binary tree structure and should only communicate with their up to three neighbours as shown in 3.1. The worker with $\xi = 0$ is also called *root* and is the *parent* of its *children* workers $\xi = 1$ and $\xi = 2$. Because the first worker of every job starts with index zero and the indices of the workers are seamlessly ascending till $w-1$, the indices of the parent and children of a worker ξ can be calculated with following formulas:

$$parentIndex(\xi) = \lceil \xi/2 \rceil - 1 \tag{3.1.1}$$

$$leftChildIndex(\xi) = \xi \cdot 2 + 1 \tag{3.1.2}$$

$$rightChildIndex(\xi) = \xi \cdot 2 + 2 \tag{3.1.3}$$

If a job loses one or more of its workers, always the workers with the highest index get suspended. Therefore, by checking $left(right)ChildIndex(\xi) < w$ a worker knows if it has a left(right) child.

When the computation is finished, Mallob will fetch the result from the root worker and will tell all workers of the finished job to terminate.

3.2 Application Interface

To write new applications for Mallob it provides an programming interface. This interface is described briefly on the projects GitHub page given at the start of chapter 3.

Any application will need some input data. To read this data, Mallob expects a read function that takes a file name as input and a $jobdescriptionobject$ to modify. The read function must return true if reading was successful, otherwise false. Now we will have a look at the most important functions of the job interface.

$appl\_start()$ is the first function that gets called by Mallob. It only gets called once because it should be used as setup. For example, constants from the job description can be read here and memory that will be used can be allocated.

$appl\_suspend()$ gets called if a worker gets suspended. It can be used to notify its parent about its suspension.

$appl\_resume()$ is called, when a suspended worker joins the job again.

$appl\_terminate()$ gets called by Mallob when a worker will be deleted soon. Every running $future$ should be signaled to stop as fast as possible here.

$appl\_solved()$ gets called periodically on the root worker. Must return a $resultCode >= 0$, when the job is done, otherwise -1.

$appl\_getResult()$ gets called on the root if $appl\_solved() >= 0$. Has to return the result of the job.

$appl\_communicate()$ gets called periodically. It can be used to send job internal messages, start computation threads or collect results of computation threads.

$appl\_communicate(source, mpiTag, msg)$ gets called on workers to receive a message. Here, the worker can react to this message by sending other messages or start computation threads.

$getDemand()$ must return an integer in range of [1, _comm_size]. This integer is the maximum w that the job can reasonably employ.

In general, all *appl_* functions must return fast because they block the main thread of Mallob. That is why computational work for the job must be done in separate threads. But sending job internal messages must not take place in these threads. Communication is only allowed to take place in Mallobs main thread.

4 Malleable K-Means

In this chapter, we will first describe how we parallelized our algorithm and which changes had to be done to make it malleable.

4.1 Parallel Approach

In this section, we will design a moldable *single instruction multiple data* K-Means algorithm. A pseudocode example is given in algorithm 3 and 4. As already mentioned in section 2.3 we will map batches of data points to workers for our malleable application in section 4.2. This malleable application will base on the moldable application we will design in this section. We will go more into detail than in section 2.3 and describe which decisions were made while implementing it for Mallob. There are already well known parallel algorithms that use a pretty similar approach as we do here, like the proposed PK-Means algorithm based on MapReduce in [23]. Our the iterations of our algorithm can roughly be divided into three parts. First we broadcast the centers of the iteration (figure 4.1 steps a, b, c). Then the workers calculate their partial centers. The last step is the reduction of the partial centers to new cluster centers (figure 4.1 steps d, e, f). Now the root worker can decide if another Iteration has to be done.
Even though Mallob is mainly designed to run malleable jobs, it can also run moldable applications as long as Mallob is run in mono application mode and the root worker waits until all workers are ready.
At first, Mallob needs to read in the data of the problem from a file. It must be saved in a job description object given by Mallob. With the job description is ready to use, all K-Means workers get created and read information like k, n d and all data points from the job description. The next step is the creation of the start clusters by the root worker. To keep comparability in later tests, we do use a random numbers generator with a seed to select the points that will be the start centers. For productive use, we take random seeds.
The last step the root worker has to do before the iterations, start is to send the start message to itself. Done this way, the root worker can react to this message like all other workers, so the same code can be used. This start message contains the cluster centers and the count of workers participating in this first iteration and it is sent to every worker in the job tree. All further broadcast messages, that announce an iteration to the workers, look similar to the start message. Communication only takes place at the edges of the job tree, which has a binary tree structure.

Algorithm 3: Lloyd's algorithm moldable part 1

```
Input: $X$, $k$ , $w$, $\xi$
Output: C
$S \leftarrow [X_j \in X | \frac{n}{w} \cdot \xi \leq j < \frac{n}{w} \cdot (\xi + 1)]$ /* S contains the points of $\xi$ slice */
if $\xi == 0$ then
    /* only the root Worker does this */
    $gotResult \leftarrow false$
    $C \leftarrow$ initStartCenters($X$, $seed$) /* start with random cluster centers */
    waitFor($w$) //Wait until all Workers are ready
    oldCenters $\leftarrow C$
    sendMessage($self$, $broadcast$, $C$)
end
def onMessage($source$, $tag$, $msg$)
    if tag == broadcast then
        reset $m[j]$ and $\mu[i]$ to zeroes only
        sendBroadcastToChildren(C) /* broadcast the centers further */
        /* assign each $X$ to its nearest Cluster (euclidean distance) */
        foreach $X_j$ in $S$ do
            /* calculate distance between $X_j$ and all $C_i$ */
            $m[j] \leftarrow i$ of $C_i$ nearest to $X_j$
        end
        /* calculate new cluster centers based on mean of all members */
        $C = [[0] \cdot d] \cdot k$ /* a $k \cdot d$ dimensioned array of zeroes */
        for $j \leftarrow \frac{n}{w} \cdot \xi$; $j < \frac{n}{w} \cdot (\xi + 1)$; $j$++ do
            $i \leftarrow m[j]$
            $\mu[i] \leftarrow \mu[i] + 1$
            $C_i$ += $X_j$
        end
        for $i \leftarrow 0$; $i < k$; $i$++ do
            /* divide $C_i$ by the count of data points in this cluster */
            $C_i \leftarrow C_i / \mu[i]$
        end
        if self.hasNoChildren() then
            payload $\leftarrow$ []
            payload.append($C$)
            payload.append($\mu$)
            sendMessage($parent$, $reduce$, $payload$)
        end
    end
end
```

Algorithm 4: Lloyd's algorithm moldable part 2

```
def onMessage($source$, $tag$, $msg$)
    if $tag == reduce$ then
        /* track how many childs are left */
        childReduced($sourcs$)
        /* aggregate using formula 4.1.1 */
        $C, \mu$ = aggregate($C$, $\mu$, $msg.C$, $msg.\mu$)
        if allChildrenReduced() then
            /* continue reduction */
            if $\xi > 0$ then
                payload $\leftarrow$ []
                payload.append($C$)
                payload.append($\mu$)
                sendMessage($parent$, $reduce$, $payload$)
            end
            /* only the root worker does this */
            if $\xi == 0$ then
                if centersChanged() then
                    /* another iteration */
                    sendMessage($self$, $broadcast$, $C$)
                else
                    /* result is ready for collection */
                    $result \leftarrow C$
                    $gotResult \leftarrow true$
                end
            end
        end
    end
end
```

Therefore, broadcast time scales in $\mathcal{O}(\log w \cdot (\alpha + k \cdot d \cdot \beta))$ with α as latency factor and β as bandwidth factor. When a worker receives a message, it saves the received centers, sends them to its children and then it starts calculating the nearest centers of its points. A worker knows which points it has to calculate distances for because each worker knows its own index. The range of points assigned to a worker starts at $\frac{n}{w} \cdot \xi$ and ends at the start of the range of the next worker. That way, every worker has the same number of points assigned to it. To compute the nearest center of each point, two nested loops are used to iterate over points and centers and compute the distance of every pair using the euclidean metric. The result gets stored in $m[j]$. With these assignments, each worker has to compute its *partial centers*, based on its point range, as if there were no other points. Two partial centers can be merged to a new partial center. With C_{i_1} and C_{i_2} as partial centers of cluster i and $\mu[C_{i_j}]$ the count of how many points of worker j are assigned to that partial center the new partial center can be calculated with following formula:

$$C_{i_{[1,2]}} = \frac{C_{i_1} \cdot \mu(C_{i_1})}{\mu(C_{i_1}) + \mu(C_{i_2})} + \frac{C_{i_2} \cdot \mu(C_{i_2})}{\mu(C_{i_1}) + \mu(C_{i_2})} = \frac{C_{i_1} \cdot \mu(C_{i_1}) + C_{i_2} \cdot \mu(C_{i_2})}{\mu(C_{i_1}) + \mu(C_{i_2})} \tag{4.1.1}$$

This formula is used in the reduction by every parent worker on its partial centers and the ones of its potential child workers. At the end of the reduction, all partial centers have formed the resulting centers of this iteration.

With finishing the calculation of the centers, we have reached the end of an iteration. Now the root worker has to decide if another iteration has to be done. In our deciding function, we will check if the centers are not moving much anymore. For factor $\epsilon > 0$ that is the case if for every dimension of every center the new entry is in range of $[|oldEntry| \cdot (1 - \epsilon)$, $|oldEntry| \cdot (1 + \epsilon)]$. If the centers moved far enough, the root center announces another iteration by broadcasting a start message containing the new centers and the number of participating workers. Otherwise, the centers are staged as ready to output.

4.2 Malleable Architecture

Now we will modify and extend the algorithm described in section 4.1 to make it malleable. For that, we have to decide what to do if one or more worker join or leave at *any* time. Workers can ask Mallob for their ξ and the current w. It is important that we can rely on always having at a root worker in our job. No other worker will become the root worker or get $\xi = 0$ in this job, as long as the job exists. Equally important is the balanced structure of the job tree. At any moment, it is guaranteed that $\xi \in [0, w - 1]$. That is the case because Mallob always suspends the workers with the highest index first. When u new workers join a job with job size w, they will get indices $[w, w + u - 1]$ and the new job size is $w + u$. Because the algorithm operates in iterations, we only have to look at a single iteration. Every other iteration will behave similar.

At startup, every worker has set itself up by loading parameters like k, d and n from the job

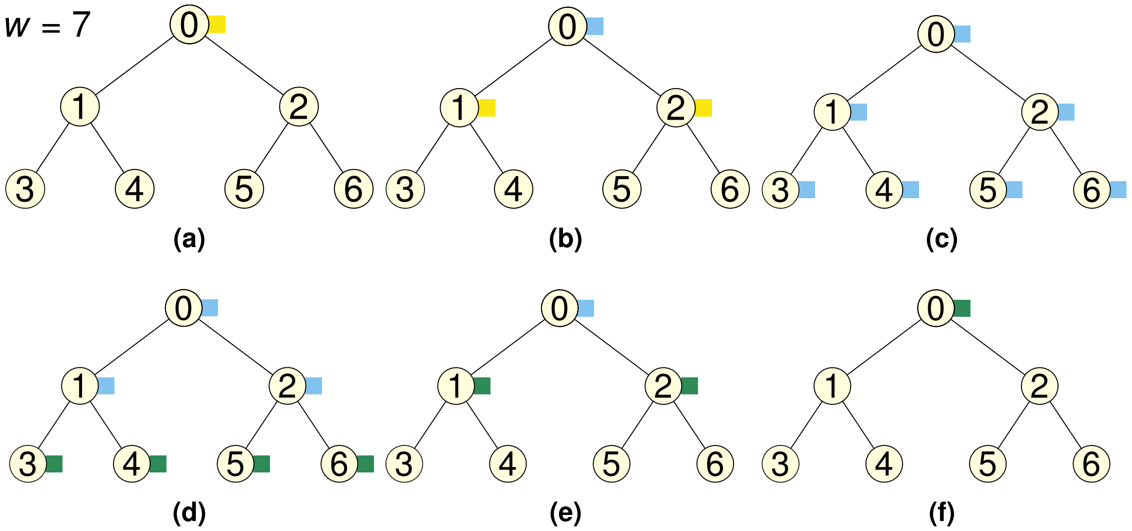


Figure 4.1: An Iteration visualized. Yellow: start message must be sent to both children. Blue: worker is calculating/waiting for its children. Green: worker finished calculating and got the partial centers of its children. Note that workers [3,6] do not have to send the start message. That is why they turn blue immediately after receiving the start message in step c.

description, allocate memory and save their index. The root worker initializes the cluster centers and puts them into the start message for the first iteration. So far, everything is done sequentially by every worker and no malleability has to be considered.

Now the iterations will start. Therefore, we must decide how we handle joining and leaving workers.

If a worker joins, w will increase immediately. What if we let the worker participate in a started iteration? Even when w increased, the worker still has to set itself up. This must also be considered because the new worker is unable start working before it has finished setting up. In the worst case, while we wait for workers to set up, another worker can join and we would have to wait again. The next problem is that the worker needs work to do. So far, a worker knows its slice of work starts at $\frac{n}{w} \cdot \xi$. Just taking some work of the new worker's parent would be possible with little changes, but will not help for a faster iteration. That is the case because the reduction must wait for the worker that takes the most time to finish its work. That is why we must take a bit of work from every other worker. But it is very expensive to change the work distribution because all existing workers have to be notified again. In that case, we also must decide, what workers must save and delete if they have calculated points inside or outside their range.

Because of all this, we do not want to wait for new workers in an iteration or calculate a new distribution of points. When the root worker asks Mallob for w the returned job size will be the basis for the distribution for the whole iteration. Every other worker will get

and use this job size alongside with the centers in the broadcast start message. Except for the root worker no other worker needs to look up w directly from Mallob ever.
This decision causes a performance problem. In the first iteration the root worker probably will always have to do the whole range of data points itself because the other workers are not ready soon enough. This results in a very long lasting first iteration for big jobs. Waiting just a bit longer could help, but that is dependent on the size of the job. On the other hand, small jobs have very short iterations where unnecessary waiting should be avoided. That is why we make an exception from not waiting for workers for the first iteration. If the root worker works alone, calculated 25% of $m[j]$ at most and $w > 1$, then we skip iterations and send new start messages with the old C. This is not only applied to the first iteration, but probably it will not be done in another iteration. Most of the times, either the job finishes before w increases from one or the root worker will not be working alone in later iterations.
Now we have decided how workers join. But we also must decide what to do if a worker leaves or is not ready. When the root worker requests w some workers might not be ready until the start message reaches them. When a start message reaches a worker, that has not finished starting up, it would be possible to wait until it is ready. That way, the waiting time would add up on the calculation time. Depending on the instance, this can be very long or very short compared to a single iteration. We decided that we will not wait and handle this worker like a suspended one. If a worker which is suspended or not ready gets a start message, it returns the message back to its parent with a *returnedToSender* flag set. When a parent receives such a message, it knows what the index of the child is and will compute the child's slice of points itself.
With that in mind, we must design a way how the parent will behave and which data it must store to be able to do additional work. Two very important points have to be considered: At first, a child can be suspended at any state of the parent. Second, maybe the suspended child had children at the start of the iteration. To handle both, we will use a *work queue* where we push in the indices of the additional data slices the parent has to do. That way, the parent can decide, when it wants to do the work and we can collect all the indices of the children and grandchildren. We cannot rely on that the child has received the suspension of its potential children, so the parent has to calculate the indices of its grandchildren itself. We do that with algorithm 5. The parent gets the index of its suspended child by message. This index is directly put into the *check queue* before the start of the recursion. As long as the check queue is not empty, recursion continues and one index is popped from the queue. If this index is lesser than the w of this iteration, this index is put into the work queue. With that index, we also calculate the grandchild indices of a child with formula 3.1.2 and 3.1.3. These two new indices are put into the check queue. Now the index that we used to calculate the new indices can be put into the work queue. If the popped index was greater equal w nothing happens to it and the next index is popped. This algorithm terminates because the two new indices calculated from the popped index are always greater than the popped index. Therefore, the indices will be greater than the current w, that is a fixed number in this algorithm. As result, the check queue will run empty because indices greater than job size will not be pushed in.

Algorithm 5: Calculation of deep children indices

```
def list<int> childIndexesOf(int $pId$, int $w$)
    $work \leftarrow []$
    $check \leftarrow []$
    $cId \leftarrow 0$
    $check.push(pId)$
    while $!check.empty()$ do
        $cId \leftarrow check.pop()$
        if $cId < jobVolume$ then
            $work.push(cId)$
            $check.push(cId * 2 + 1)$
            $check.push(cId * 2 + 2)$
        end
    end
    return $work$
end
```

5 Experimental Evaluation

In this chapter, we will look at some details of the implementation of our K-Means Mallob application, present the results of our experiments and how we used them to speed up concurrent job computations.

5.1 Implementation

In this section, we will talk about implementation details. The application's source code is available at https://github.com/MichaelDoerr/mallob/tree/kmeans. Mallob is written in C++ and so is our K-Means application. We start with the creation of our source files.
We create our K-Means application in the application folder of Mallob. All of our source files are within that folder. In one file we implement the job description file reader and in another one or implementation of the job interface. Some classes of Mallob need new cases for our application. In the creation of a new job context, the reader for incoming jobs, the creation of job descriptions and the json interface these new cases are needed. Now everything is prepared to start with the implementation.
At first, Mallob needs a function that reads in the data of the problem from a file. We implement a function $bool\ read(filename, jobDescription)$ in *src/app/kmeans_job.cpp*. Many example K-Means data sets online, save points in rows and separate dimensions by comma or space. Loading the data is done by iterating over the float values and saving them into the job description object given by Mallob. No separation between points are made, that is why at the start of the file the k, d and n are expected. These 3 numbers are also saved at the start of the job description object.
$appl\_start()$ sets k, d, n and also the start address of the first point. Later, the start address of each point can be obtained by calculating $pointsStart + (dimension \cdot point)$ where point is the index point j. After that, the $vectors\ clusterCenters$, $localClusterCenters$ and $clusterMembership$ are created. The vector $localClusterCenters$ stores the centers that are calculated from the worker's slice of data. $clusterMembership[j]$ stores $m[j]$ and its size is n. That way, it does not have to be resized when a worker gets more or bigger data slices than initially. The data points are directly read from the $jobdescription$ when they get accessed.
$appl\_suspend()$ sends a message with ξ and the $returnedToSender$ flag set true.
$appl\_resume()$ resets the worker to a clean state, ready for a new iteration.

$appl\_terminate()$ sets the $terminate$ flag. Computation threads check this flag periodically and stop when set.
$appl\_communicate()$ sends the first start message if it is is called the first time by the root. Also, it checks if the $work\ std :: vector$ is not empty and puts work from it into the $ProcessWideThreadPool$ provided by Mallob. Entries of $clusterMembership$ are calculated here. If the $work$ is empty, the worker finished calculating and does not have to wait for its children, the $localClusterCenters$ are calculated and given into the $reducer$.
$initReducer(JobMessage\&\ msg)$ initiates an $JobTreeAllReduction$ object from Mallob based on the start message for the current iteration. Only the reduction part is used because the root worker first has to decide, if another iteration has to be done. Also, msg is sent to the children to continue the broadcast of the start message.
$appl\_communicate(source, mpiTag, msg)$ returns the message to its sender, if the worker has not finisthed $appl\_start()$ or is suspended. Otherwise, $Msg\_Broadcast\_Data$ tagged messages start a new iteration. The payload is saved in $clusterCenters$. If the $\xi < w$, $initReducer(\ msg)$ is called and the worker's slice of $clusterMembership$ gets calculated by the $ProcessWideThreadPool$. $Msg\_Job\_Tree\_Reduction$ tagged messages are put into the reducer.
$getDemand()$ returns maximum demand based on the result of 5.4.1. It is rounded to $2^l - 1$ for best job tree balancing.
$calcNearestCenter(metric, intervalId)$ calculates the part of $clusterMembership$ that is assigned to $intervalId$. This is done by iterating over the data points and centers and saving the center with the shortest distance to the data point in $intervalId$.
$calcCurrentClusterCenters()$ calculates the result of $localClusterCenters$ for the current iteration. The workers own data points and the data points of suspended children are used for that. The calculation uses formula 4.1.1 in loop.
$centersChanged(\epsilon)$ is used to decide if another iteration should be done. It returns false, if 5.1.1 with center k and dimension d. That means that the entries do not change more than ϵ in ratio to their own size. Otherwise, true is returned. $floatmetric(p1, p2, d)$ returns the distance of two points to each other that can be compared to find the nearest center of a data point. Implementation changed while development and does return the squared euclidean distance for performance reasons 6.

$$|newCenters[k][d] - oldCenters[k][d]| \leq |\epsilon \cdot \frac{newCenters[k][d] + oldCenters[k][d]}{2}| \tag{5.1.1}$$

Algorithm 6: Distance calculation

```
def float metric(Point $p1$, Point $p2$, size $d$)
    $\sum \leftarrow 0$
    $\delta \leftarrow 0$
    for $l \leftarrow 0; l < d; ++l$ do
        $\delta \leftarrow p1[l] - p2[l]$
        $\sum$ += $\delta \cdot \delta$
    end
    return $\sum$
end
```

5.2 Experimental Setup

We tested the application on three machines.
Machine 1 runs WSL2 with Ubuntu 20.04 installed on a AMD Ryzen 7 5800X 8 Core machine with 16GB DDR4. Our application was developed[1] on that machine.
Machine 2 is an Ubuntu 20.04 machine with 4 sockets × Intel Xeon Gold 6138 20 Core and 768GB DDR4. In total, it has 80 physical cores and 160 logical cores.
Machine 3 is an Ubuntu 20.04 machine with 2 sockets × AMD EPYC 7713 processor 64 Core and 1TB of DDR4. In total, it has 128 physical cores and 256 logical cores.
The test K-Means instances for the section 5.3 were *covertype* from [8] and *benign_traffic* [12] from a Samsung_SNH_1011_N_Webcam. The points of both instance files were shuffled once before the first test. Most tests we run on Machine 3.

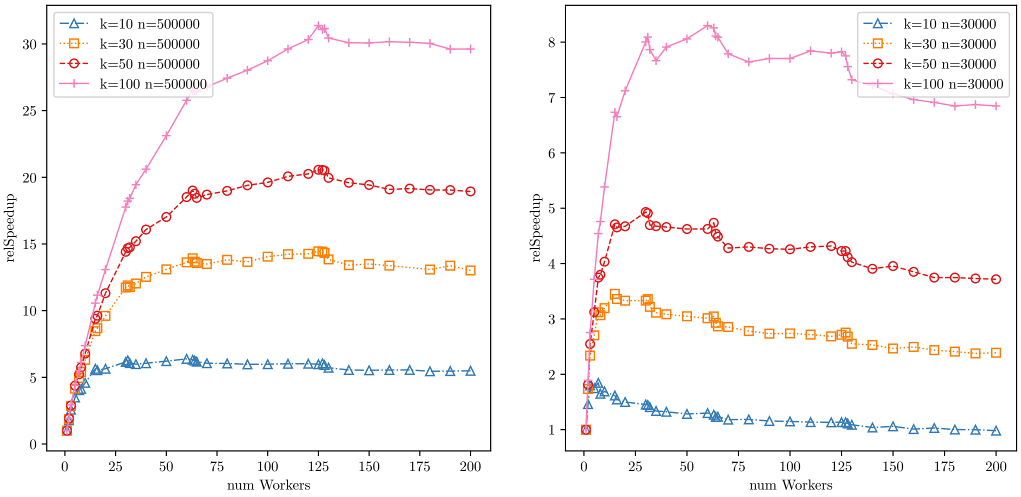
5.3 Scaling Results

We run multiple tests, with different parameters for k, the count of data points n to be used and count of workers for mono application mode w. The resulting self speedups were very dependent on how much work per iteration an instance has to do. The main work is done at calculating the closest center of a data point. The effort is in $O(n \cdot k \cdot d)$ because for n data points the distances to k centers have to be calculated. The calculation of a distance consists of subtracting two d dimensional vectors from each other and squaring and adding up floats of d dimensions.
In figure 5.1a we clearly see rising speedups up to 127 workers. Also, we can see kinks at 2^i workers. That is caused by the structure of the job tree. At 2^i a new level of worker is

[1] Mallob only supports native Linux systems. On this Windows machine with WSL2, mono application mode worked without noticeable problems. Orchestrated application mode does not terminate, after the job limit is reached and on demand application mode does not read new jobs. The problem is probably, that Mallob does not get notified about changes in the folder *.api/jobs.0/in/*.

Table 5.1: Performance of our application compared to scikit-learn K-Means

instance k	Runtime in seconds			
covtype 100	Sequential	32 PEs	64 PEs	128 PEs
Our Application	1 569.256	85.191	59.229	50.368
scikit-learn	38.08	10.198	10.338	

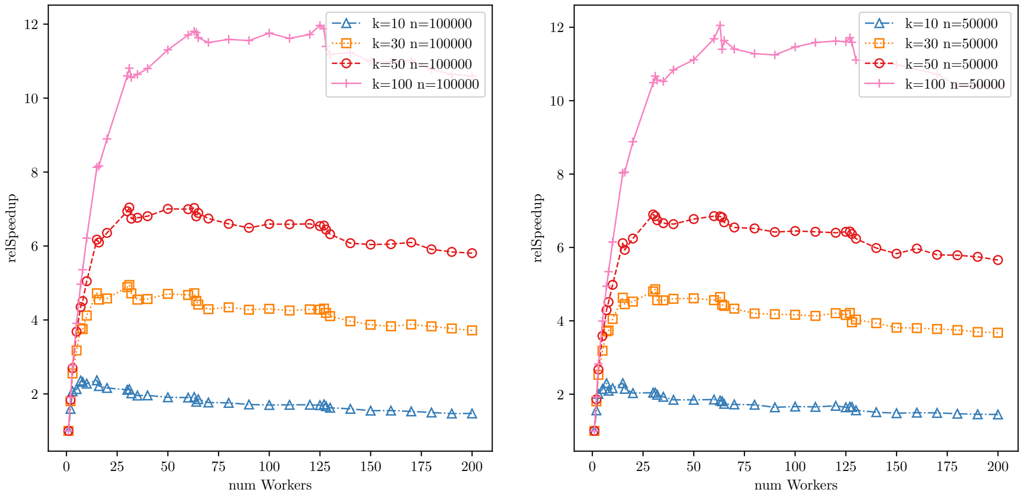


(a) 54 dimensional *covertype* self speedup **(b)** 115 dimensional *benign_traffic* self speedup

Figure 5.1: For both tests, mono application mode was started on machine 3. The MPI flag oversubscribe was set to use more than 128 workers in some tests. Both instances were started with four different counts of cluster centers k and 36 different counts of workers in range $[1, 200]$. n is the count of the used data points.

created, causing more sequential communication. With rising counts of workers, the slices of data points assigned to each worker get smaller. A single additional worker has less impact on the slice size, the more workers are already in the job. The low impact on slice size and more communication overhead cause a *garland* effect that we can clearly see in 5.1b at $k = 50$ and $k = 100$. When more workers are added to the same level, the rise of speedup slowly recovers. This effect inspired us to create a function that calculates an upper bound for the job demand in section 5.4.

Also, we compared our application to *scikitlearns* K-Means library for python [13]. With a significantly longer development time of twelve years, it is faster than our implementation by at least one order of magnitude as we can see in table 5.1. The comparison tests run on machine 3 with covtype $k = 100$, $n = 500\,000$ and exactly 150 iterations. The last entry of table 5.1 is empty because scikit-learn was not able to run with more than 68 threads and crashed with an error message.



(a) *covertype* self speedup

(b) *benign_traffic* self speedup

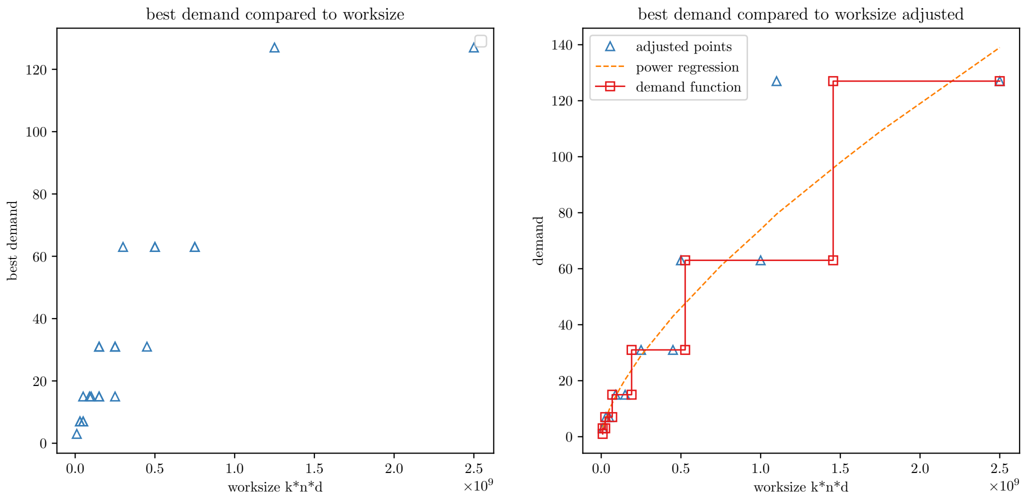
Figure 5.2: Both tests seem to have alike scaling, even though n and d are different. But if we multiply n and d from both instances, 5 000 000 is the resulting factor both times. This indicates that the instance size can be described by $n \cdot k \cdot d$.

5.4 Performance of scheduled Jobs

Mallob's main feature is scheduling concurrent, malleable jobs. With an upper bound for the demand of a job, we can improve the distribution of workers over the jobs. For example, we take an 80 core machine with 80 workers, instance e $k = 50$ from 5.1a and instance f $k = 30$ from 5.1b. If we restrict the demand of e to 63 and the demand of f to 15, only 78 workers are used and the speedups are at 18.766 for e and 3.453 for f. If both jobs get equal numbers of 40 workers each, the speedups are 16.079 for e and 3.084 for f. That also means the jobs take longer to finish while using more workers in comparison to the version with restricted job demands. To create a function that describes this bounds, we run tests with *covertype* and *benign_traffic*. For *covertype* we used all (k, n) combinations out of $\{10, 30, 50, 100\} \times \{100\,000, 300\,000, 500\,000\}$. For *benign_traffic* we used the (k, n) combinations from $\{10, 30, 50, 100\} \times \{10\,000, 30\,000, 50\,000\}$. We collect the highest scaling points, that usually occur at $2^i - 1$ because of the balanced job tree. For each point we, save pairs of $(n \cdot k \cdot d, workers)$. As shown in 5.3 the highest scaling points can be approximated with a power function. For the adjusted points in 5.3b we calculated the power regression 5.4.1.

$$5.41 \cdot 10^{-5} \cdot (n \cdot k \cdot d)^{0.682} \tag{5.4.1}$$

We have to keep in mind that the job tree is shaped like a binary tree. That is why our best demand points always are at $2^l - 1$ workers. We also want our demand function to return upper bounds at the end of a job tree level. To round the power regression to this point, we first have to calculate its value v for our current job. v is between $2^{l-1} - 1$ and $2^l - 1$ for a



(a) Every point represents a measured best demand of a test with multiple workers.

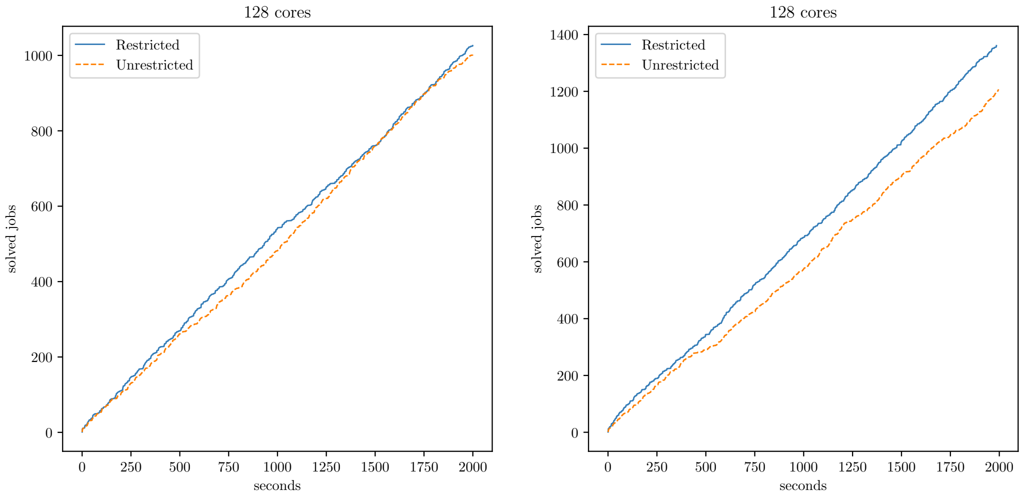
(b) Points are filtered and adjusted for a more accurate power regression.

Figure 5.3: The best demand points must be filtered from duplicates and too optimistic values. Otherwise, the calculated power regression is warped to the direction of the most points. That is why we only left two points per row. The resulting power function is rounded to $2^l - 1$ to balance the job tree.

specific l. $2^{l-1} - 1$ is returned if $v \leq 2^{l-1} + 2^{l-2}$, otherwise $2^l - 1$ is returned. The resulting step function is now our demand function. It is visualized in 5.3b.

With the new demand function, we used machine 3 with a set of 20 different instances from [2] including *covtype* and *benign_traffic*. In orchestrated application mode, we count how many jobs can be solved by Mallob in a given time interval of 2000 seconds. For each solved job, a point (currentTime, solvedJobsUntilNow++) is added. First results showed nearly no difference of impact from the two different demand functions like in 5.4a. In the restricted run of 5.4a the more demanding jobs stack up while low demanding jobs get solved only bit faster than in the unrestricted run. Between second 1154 and 1170 no job is solved. In comparison to the average solving time of one job every two seconds, this is very slow. This results in a better performance only between second 500 and 1500. While the demand restricted version does not perform better in this case, it also does not perform worse. With a changed workload of more smaller sized jobs like in 5.4b and less concurrent jobs, a significant speedup of the restricted run can be observed. This is a more important scenario because in reality usually less concurrent jobs are active at a time. A user that inserts a job in Mallob probably will not insert another job right after Mallob returns the result of the first job.

While the restricted run of 5.4b solved 1361 jobs, the unrestricted run only managed to solve 1207 jobs. This results in a speedup of 1.13 for the Restricted run.



(a) Solved jobs per time with maximum 8 concurrent jobs

(b) Solved jobs per time with maximum 6 concurrent jobs and more smaller sized jobs

Figure 5.4: The *Restricted* runs use our demand and the *Unrestricted* have no demand limit.

6 Conclusion

We have introduced K-Means as NP hard problem and discussed the different approximation algorithms that are currently used to solve it with sufficient accuracy. Also, we have looked at different ways to parallelize the classic Lloyds approximation algorithm for K-Means. Then we introduced Mallob as framework to write malleable applications for problem solving. For a better start in application writing for Mallob we described the application interface and important details of the internal structure of the framework. With the decision to divide computational work at data level, we first designed a moldable algorithm based on Lloyds algorithm. To get a malleable application from our moldable algorithm, we discussed which problems can occur. We decided what we do with joining and leaving workers at any time. With the finished design of our malleable algorithm, we had insight of how to implement it with Mallob. The result is a well scaling application that can be used for on demand job solving. With an evaluated upper bound for workers, we optimized concurrent job runtimes without interfering with job priorities.

For future work, the implementation of our algorithm can be optimized in general. With dynamical adjustment of the size of work slices, the *garland* effect in 5.3 can be fixed for better performance. Also, the usage of highly optimized matrix multiplication libraries can be considered.

A bigger work can be the examination of Elkans algorithm Elkan [6] regarding malleability. It is faster with high dimensional data what makes it interesting for practical usage. Because of higher memory and communication demand, good scaling with malleability is not guaranteed.

Bibliography

[1] D. Arthur and S. Vassilvitskii. *k-means++: The advantages of careful seeding*. Tech. rep. Stanford, 2006.

[2] T. Barton. *clustering-benchmark*. June 2019. URL: https://github.com/deric/clustering-benchmark.

[3] T. H. Cormen et al. *Introduction to Algorithms*. 2nd ed. The MIT Press, 2001. ISBN: 0-262-03293-7.

[4] S. Dasgupta. *The hardness of k-means clustering*. Department of Computer Science and Engineering, University of California, 2008.

[5] L. Du, Y. Du, and M.-C. F. Chang. „A Reconfigurable 64-Dimension K-Means Clustering Accelerator With Adaptive Overflow Control“. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.4 (2020), pp. 760–764. DOI: 10.1109/TCSII.2019.2922657.

[6] C. Elkan. „Using the triangle inequality to accelerate k-means“. In: *Proceedings of the 20th international conference on Machine Learning (ICML-03)*. 2003, pp. 147–153.

[7] D. G. Feitelson. „Job scheduling in multiprogrammed parallel systems“. In: (1997).

[8] D. of Forest Sciences College of Natural Resources Colorado State University. *Forest CoverType dataset*. Aug. 1998. URL: https://archive.ics.uci.edu/ml/datasets/Covertype.

[9] A. K. Jain. „Data clustering: 50 years beyond K-means“. In: *Pattern Recognition Letters* 31.8 (2010). Award winning papers from the 19th International Conference on Pattern Recognition (ICPR), pp. 651–666. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2009.09.011. URL: https://www.sciencedirect.com/science/article/pii/S0167865509002323.

[10] T. Kanungo et al. „A local search approximation algorithm for k-means clustering“. In: *Computational Geometry* 28.2-3 (2004), pp. 89–112.

[11] S. Lloyd. „Least squares quantization in PCM“. In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.

[12] Y. Meidan. *detection_of_IoT_botnet_attacks_N_BaIoT Data Set*. Mar. 2018. URL: https://archive.ics.uci.edu/ml/datasets/detection_of_IoT_botnet_attacks_N_BaIoT.

[13] F. Pedregosa et al. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[14] M. L. Pinedo. *Scheduling*. Vol. 29. Springer, 2012.

[15] K. Rupp. *48 Years of Microprocessor Trend Data, July 16, 2020*. DOI: `10.5281/zenodo.3947824`.

[16] P. Sanders and D. Schreiber. „Decentralized Online Scheduling of Malleable NP-hard Jobs“. In: *International European Conference on Parallel and Distributed Computing*. In press. Springer. 2022.

[17] D. Schreiber. „Engineering HordeSat Towards Malleability: mallob-mono in the SAT 2020 Cloud Track“. In: *SAT Competition 2020* (), p. 45.

[18] D. Schreiber. „Mallob in the SAT Competition 2021“. In: *SAT Competition 2021* (), p. 38.

[19] D. Schreiber and P. Sanders. „Scalable SAT Solving in the Cloud“. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2021, pp. 518–534. DOI: `10.1007/978-3-030-80223-3_35`.

[20] D. Sculley. „Web-Scale k-Means Clustering“. In: *Proceedings of the 19th International Conference on World Wide Web*. WWW ’10. Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, pp. 1177–1178. ISBN: 9781605587998. DOI: `10.1145/1772690.1772862`. URL: `https://doi.org/10.1145/1772690.1772862`.

[21] W. L. Al-Yaseen, Z. A. Othman, and M. Z. A. Nazri. „Multi-level hybrid support vector machine and extreme learning machine based on modified K-means for intrusion detection system“. In: *Expert Systems with Applications* 67 (2017), pp. 296–303.

[22] Y. Zhang et al. „The Study of Parallel K-Means Algorithm“. In: *2006 6th World Congress on Intelligent Control and Automation*. Vol. 2. 2006, pp. 5868–5871. DOI: `10.1109/WCICA.2006.1714203`.

[23] W. Zhao, H. Ma, and Q. He. „Parallel k-means clustering based on mapreduce“. In: *IEEE international conference on cloud computing*. Springer. 2009, pp. 674–679.