FAST K-MEANS CLUSTERING VIA K-D TREES, SAMPLING, AND PARALLELISM

by

Thomas Crowell

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2019

To Caleigh, who supported me every minute of every day in my Master's journey. Thank you for always being there for me and encouraging me to keep pushing onwards.

Table of Contents

List of Tables . . . . **v**

List of Figures . . . . **vi**

Abstract . . . . **vii**

Acknowledgements . . . . **viii**

Chapter 1 Introduction **1**

Chapter 2 Previous Work **3**

2.1 Cluster Analysis 3

2.2 K-means Clustering 4
2.2.1 Tree data structure k-means 6
2.2.2 Parallel k-means 8
2.2.3 K-means with sampling 8
2.2.4 Other approaches 9

2.3 Competitive Algorithms 9
2.3.1 Mini Batch k-means 9
2.3.2 Filtering algorithm 10

Chapter 3 Engineering an Efficient and Flexible K-D Tree Based K-Means Implementation **11**

3.1 Implementing K-D Tree K-Means 11
3.1.1 Parameters 12
3.1.2 Representation of the Point Set 13
3.1.3 Tree Building 13
3.1.4 Filtering 14
3.1.5 Optimizations 17

3.2 Experimental Evaluation 18
3.2.1 Parameter Tuning 20
3.2.2 Comparison With Competitors 20
3.2.3 Effect of Dimensionality 24

Chapter 4 Impact of Sampling on K-Means **25**

4.1 Sampling Approaches 25

4.2 Exploratory Results 26

Chapter 5 A Parallel Implementation of the Filtering Algorithm 32

5.1 Parallel Computing 32

5.2 Parallelism in Lloyds Algorithm 33

5.3 Parallelizing Mini Batch K-means 33

5.4 Parallelism Strategies with the Filtering Algorithm 34
5.4.1 Parallel Tree Building 34
5.4.2 Single Stack Traversal 36
5.4.3 Work Sharing Traversal 37

5.5 Experimental Results 40

Chapter 6 Final Performance Evaluation 42

6.1 Combining K-D tree, Sampling, and Parallelism 42

6.2 Experimental Results 42

Chapter 7 Conclusion 44

7.1 Conclusion 44

7.2 Future Work 44
7.2.1 GPU implementation 45
7.2.2 Skewing Metrics 45

Bibliography 46

List of Tables

3.1 Comparison of total run time of our K-D tree parameter combinations in seconds [mean (sd)]. 21

3.2 Comparison of sequential filtering algorithm k-means implementations tree build times in seconds (total time) [mean (sd)] . . . 22

3.3 Comparison of sequential filtering algorithm k-means implementations query times in seconds (total time) [mean (sd)] 22

3.4 Comparison of k-means method run times in seconds (total time) [mean (sd)] 23

3.5 Comparison of k-means final objective function values rounded to nearest whole number [mean (sd)] 23

4.1 Objective Function and Total Running Time for Sampling on K-D tree k-means at optimal parameters. 28

5.1 Comparison of k-means method run times in seconds [mean (sd)] 41

6.1 Comparison of k-means method run times in seconds [mean (sd)] 43

List of Figures

2.1 An example of k-means clustering 5

2.2 An example of building a k-d tree from a point set. 7

3.1 An example of our pre-allocated traversal and candidate stack. 19

4.1 Relative running times and objective function values at different sampling rates compared to their baselines across each dataset on 10 centers. 29

4.2 Relative running times and objective function values at different sampling rates compared to their baselines across each dataset on 50 centers. 30

4.3 Relative run times and relative objective function values of Crowell 1% sample, Crowell 10% sample, and Mini Batch k-means implementations. 31

5.1 An example visualization of parallel tree building. 35

5.2 The global stack parallel filtering algorithm keeps a global stack for traversal order shared by all threads. 36

5.3 An example of the work sharing algorithm. 38

Abstract

K-means is a commonly used method for clustering in applications that require fast response time due to its speed. As data becomes large (millions of data points), the classical implementation may not achieve the performance necessary for these applications. By combining the filtering algorithm using k-d trees, aggressive sampling, and parallelism with dynamic load balancing, we implement a version of k-means that outperforms the standard algorithms used for these applications. We find that aggressive sampling at 1% of the dataset combined with the filtering algorithm provides significant speed-up without sacrificing accuracy. Overheads in implementing parallel methods prevent significant speed-up on smaller datasets, especially when the data has already been sampled, but our experiments show that this improves as the dataset grows.

Acknowledgements

I would first like to thank my thesis advisor Dr. Norbert Zeh for his valuable input in shaping this research. This work could not have been done without him and the time he committed to guide me along every step of the process.

I would also like to express my appreciation for assistance provided by Magdalena Jankowska in securing properly formatted data for the research and help with debugging code.

I would also like to acknowledge that this work could have not been completed without support from the TRIBE CREATE program at Dalhousie University.

Finally, I would like to thank Dr. Andrew Rau-Chaplin for encouraging me to pursue my masters and my parents, friends, and significant other for always supporting and encouraging me through it all.

Chapter 1

Introduction

K-means is a widely used technique for clustering data into defined partitions based on their similarity. It has gained prominence because of several valuable qualities, such as its ability to quickly find decent clusters as a starting point for other complex, more expensive clustering methods. Arguably, k-means most important quality is its fast run time relative to almost every other clustering method. Its speed comes at a cost of cluster accuracy compared to other clustering methods, although some applications do not require a precise clustering and place more importance on the ability to obtain a clustering quickly. One prime example of this is interactive clustering applications [29, 12].

Interactive applications require fast response time (usually below one second) to keep the application flowing and the user engaged. The need for speed makes k-means an excellent candidate for interactivity. As most clustering methods running times scale up as the input size and dimensionality increases, interactivity and fast response times become even more difficult to achieve. High dimensional data is often reduced to lower dimensions for this reason [29, 9]. In this work, we explore existing techniques for obtaining fast low-dimensional k-means implementations on large datasets. Our goal is to combine them to achieve interactive speeds on large datasets (10s of millions of records).

To do this, we explore the impact of using efficient data structures, sampling, and parallelism in a k-means implementation. These are standard techniques in the literature for improving the performance of k-means [19, 8, 30, 21, 28, 7]. Using k-d trees allows for an alternative k-means algorithm that reduces the amount of computations required to calculate a clustering and pre-allocating large chunks of memory with the use of a memory manager, which improves the efficiency of the code. Sampling allows the algorithm to run on only a subset of the data resulting in many less computations being run, but with a minor trade-off in cluster accuracy.

Parallelism allows the algorithm to partition the work required for k-d tree building and calculating the clustering such that subsets of work can be run concurrently on multiple threads. A combination of these techniques could utilize the best parts of each and allow us to work on large datasets at speeds that are sufficient for interactive applications and are much faster than existing k-means implementations.

These techniques are relatively simple to implement individually, but present challenges when combined. For instance, it is trivial to parallelize Lloyds algorithm, the most common k-means algorithm. In contrast, the filtering algorithm, a k-d tree-based approach, raises several challenges. K-d trees may be imbalanced, and each node may require a different level of computation in a given k-means run, leading to challenges in balancing the work done by each thread.

In this text, we engineered our own implementation of k-means that works with both sequential and parallel traversals to allow us to re-evaluate choices made in the literature and compare how our implementation stacks up to the fastest in the literature. Our implementation overcomes the challenges of combining parallelism, k-d trees, and sampling in a single k-means implementation. In doing this, we contribute a unique work sharing load balancing strategy, an efficient memory manager for managing k-d tree traversals in k-means, and an efficient structure for parallel and sequential k-d tree k-means traversals.

In this thesis, we will outline various approaches to k-means clustering and give a detailed explanation of how we engineered and optimized our own implementation of the k-d tree based filtering algorithm. We will then explore the effects of sampling and parallelism combined with the filtering algorithm and compare them to other fast algorithms in the literature. We also explore the challenges in creating a parallel version of the filtering algorithm and conclude with final results and avenues for future research.

Chapter 2

Previous Work

This chapter reviews previous work relevant to this thesis, including a brief introduction to cluster analysis and a detailed overview of techniques that have been used to improve the speed and accuracy of k-means clustering. K-means is a well researched topic and has evolved in several directions to improve its efficiency and accuracy on different data shapes and sizes. We are most interested in obtaining a very fast k-means implementation, without sacrificing too much accuracy. The fastest approaches in the literature frequently use certain techniques, such as sampling, parallelism, and use of appropriate data structures. We will use them as building blocks for our approach. We will go in more detail how these techniques are commonly applied to k-means, and outline some of the fastest approaches and their strengths and weaknesses.

2.1 Cluster Analysis

Cluster analysis (referred to as clustering from here on) is an important tool in statistical data analysis and machine learning. Its goal is to take an unsupervised approach to partitioning a set of data objects into k groups of similar objects (clusters), where k may or may not be defined by the user. Clustering is a very broad term, given that there are many ways to approach collecting data points into clusters. It is a challenging problem that has spawned various approaches including using density of the point set, the distribution of the point set, distance-based connectivity of the point set, and centroid-based measures where each group is represented by a summarized point in the space. Some examples of these include DBSCAN [15] and OPTICS [3] for density-based clustering, DBCLASD [31], HCS algorithm [18], and k-CONID [26] for connectivity-based clustering, and k-medoids [20] and k-means [23] for centroid-based clustering.

2.2 K-means Clustering

The k-means clustering algorithm is among the most popular clustering algorithms. The most common algorithm for implementing k-means is an iterative algorithm called Lloyds Algorithm [22] (frequently referred to as the k-means algorithm). The goal of k-means is to partition a point set into k clusters so that the sum of the squares of the (Euclidean) distances between the observations and their closest cluster centroids is minimized. Finding the optimal k-means cluster partition is an NP-Hard problem [10]. K-means is generally used as a quick technique for estimation of a clustering, or as an initial clustering for other more expensive methods [34]. It is generally favoured for its simplicity to implement, its speed, and various optimizations that tweak the algorithm to better adapt to certain shapes and sizes of data. K-means is not without limitations however, as it is susceptible to converging to local optima rather than global optima and the quality of the computed clusters can depend significantly on the initial random cluster centroid choice. K-means is also limited in that it can only find convex clusters and can struggle with non-convex clusters even if they are very well defined. Initial centroid seeding techniques such as k-means++ [4] can help k-means achieve robust convergence and more accurate clusterings and finding embeddings that reduce the complexity can allow k-means to work correctly on non-convex data [34].

In the simplest case, the user defines k, the number of centroids/clusters to compute, and a minimum percentage by which the objective function must improve in each iteration for the algorithm to start another iteration. Initially, as the algorithm requires an initial centroid set, a set of k random points are chosen as centroids (usually from the input point set). The algorithm then alternates between 2 steps: an assignment step, and an update step. These steps will be explained below. Each time the assignment step and update step are run, this is considered to be an iteration of the algorithm.

In the assignment step, the Euclidean distance between each observation and each centroid is calculated to determine which centroid is the closest in the space for an observation. An observation closest to a centroid is said to be assigned to that centroid's cluster.

In the update step, each centroid is re-defined as the coordinate-wise mean of all observations assigned to that centroid's cluster. The algorithm then computes the distortion, defined as the sum of the squares of the distances between the observations and their closest cluster centroids, for this new centroid set and terminates if the ratio between the computed distortion and the previous iteration's distortion is less than the minimum distortion loss (the user-defined minimum improvement between iterations). We can see an example of this in Figure 2.1, where the updated centers are pulled towards the mean of the points closest to them. If the algorithm does not meet this stopping condition, it returns to the update step, with the newly defined centroids as its input. If it does meet the stopping condition, the algorithm does not continue updating, and the last computed centroids in the update step are the final centroids. The assignment step is run one last time to determine which observation is assigned to which centroid, giving a partition of observations into clusters.

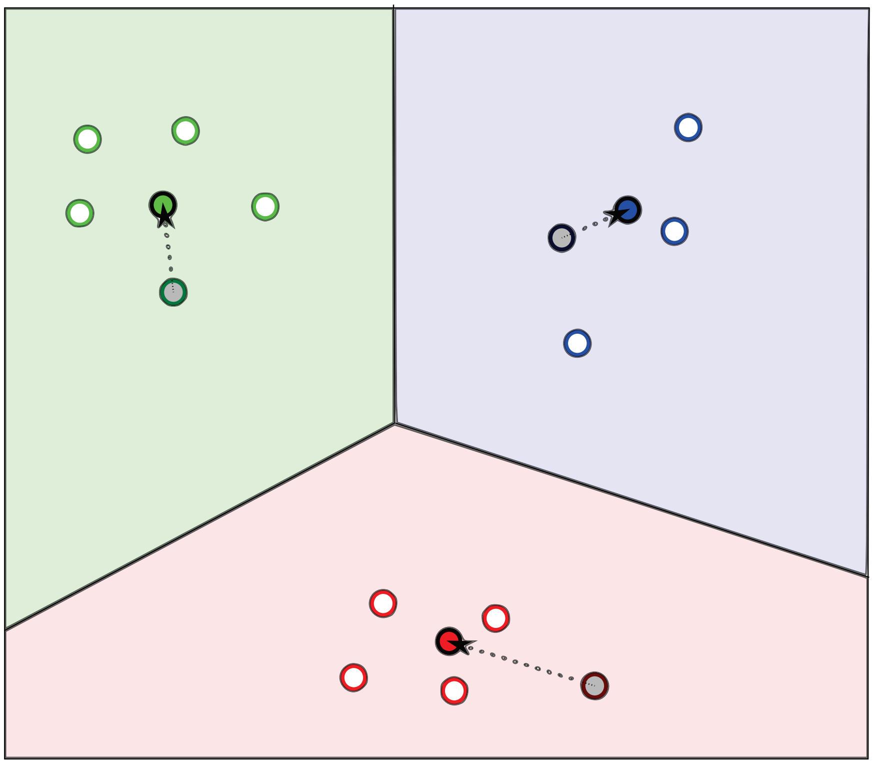
Figure 2.1: A potential k-means clustering, where the gray filled points are the initial centroids. The black outlined points are the final converged centroids, with their Vornoi partition shown and each point assigned to that centroid outlined in the same color.

While Lloyds Algorithm is very fast compared to the more sophisticated clustering methods, it is nevertheless too slow to apply to very large inputs. As a result, improving on the speed of k-means is a well-researched topic space. Approaches to improve the running time include utilization of data structures such as quadtrees [25, 8] and k-d trees [1, 19], parallelization [30, 6, 21, 33], sampling [28, 7, 11], initialization optimization [4, 5], candidate pruning [14, 17, 13], and many hybrid methods [16, 32] that combine these techniques. Among these, search trees like k-d trees provide fast nearest neighbour search capability in low-dimensional spaces, which is the key step to be performed in each iteration of Lloyds algorithm.

2.2.1 Tree data structure k-means

Search tree-like data structures, such as k-d trees and quadtrees, provide a query-efficient spatial summary of a point space. In these trees, we think of the root node of a tree as a bounding box enclosing the entire point set. The point space can be continuously partitioned into subspaces, generally where the points in the space are partitioned approximately evenly amongst the children of a node. Each division of the point space creates new bounding boxes, which are represented as the child nodes of the larger space that was divided. The point space is recursively divided until a child node contains one or very few points. We can see an example of how a k-d tree is built from a point set in Figure 2.2. We describe in more detail how we build our trees in Chapter 3. At each of the nodes in the tree, we can collect summary information about the points contained in the bounding box represented by that node, which can allow us to better guess initial centroids [8] or assign groups of points at once during the assignment step [19]. When using the tree for speeding up the assignment step, the summary information collected allows us to quickly prune candidate centroids from groups of points held in internal nodes. When a group of points has only one candidate centroid remaining, we determine that the candidate is the closest centroid for the entire group and no longer need to continue traversing the subtree. This eliminates many distance calculations and comparisons compared to other k-means implementations. As the tree is only constructed once, the algorithm is better optimized as long as the time overhead of building the tree is less than the time saved by using the tree.

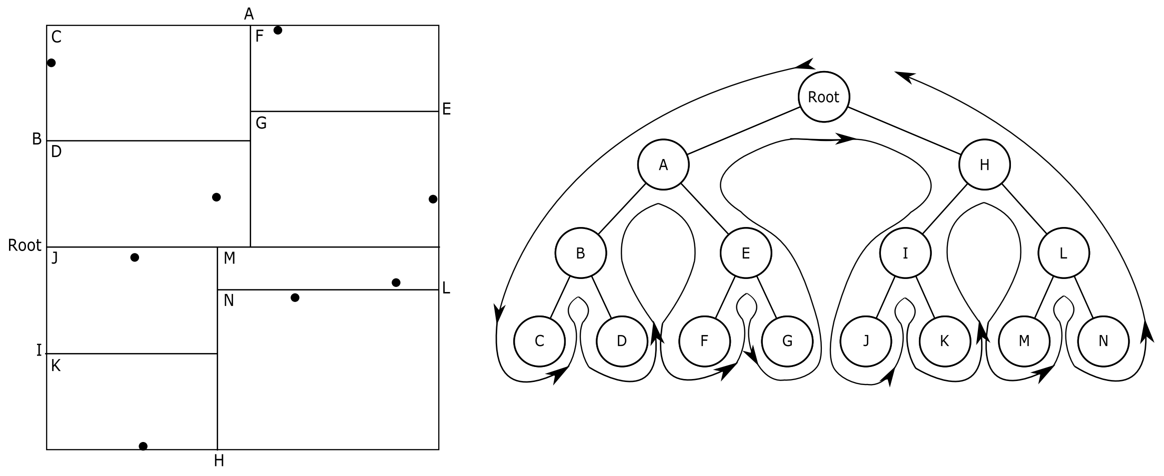

Figure 2.2: An example of building a k-d tree from a point set. Given a set of points with the bounding box on the left side of the figure, we can see each split the algorithm performs to create the tree. The line initially splitting the bounding box into 2, labeled Root, is our initial split. We then split the resulting top bounding box into 2, line A, and one of its children into 2, line B. This gives us our first two leaf nodes, boxes C and D. This is continued recursively until the entire set is partitioned. Each internal tree node represents the combined bounding box of the children it creates by splitting for that corresponding letter. This gives us the tree on the right of the figure. The traversal order for building the tree recursively is shown around the outside of the tree, with arrows to indicate the flow of the traversal.

2.2.2 Parallel k-means

Utilizing parallel algorithms is another effective way to gain speed-ups on the k-means algorithm, especially on large datasets [30, 33, 16]. Lloyds Algorithm is trivial to parallelize on a CPU (Central Processing Unit) by dividing the point set into p groups during each assignment step, where p is the number of CPU cores available. We can then allow each core to perform distance calculations and determine the nearest centroid for the $\frac{N}{p}$ points it is responsible for. Furthermore, parallelization can be taken a step further utilizing a GPU (Graphics Processing Unit) or combining parallelization with other optimization techniques. GPUs differ from CPUs in that a CPU has a small number of fast cores that are each capable of running separate instructions, whereas a GPU has many (thousands) slow, rigid cores that have limited memory and must work in unison. All threads in a GPUs thread block are required to run the same instructions in unison with each other and wait for all threads to synchronize before completion. They are also very limited in that they cannot hold much data in local thread memory, and memory transfer to and from the GPU incurs relatively high overhead. This limits the popularity of parallel GPU algorithms outside of trivially parallelizable problems. An easily parallelizable algorithm like Lloyds Algorithm fits these limitations and can be easily implemented on a GPU, while something more complex such as the filtering algorithm (k-d tree based k-means) is less popular due to the difficulty to implement and effectively utilize all the cores to overcome memory overhead.

2.2.3 K-means with sampling

Sampling utilizes running iterations of a k-means algorithm on only a subset of the whole point set and generally provides a trade-off of accuracy to increase speed. So long as the number of clusters is not too large compared to the data set, the statistical properties of the population of the input are preserved. The two most common methods of sampling involve sampling once at the beginning of the k-means run or resampling from the whole point set after each iteration. This allows the number of distance computations to be reduced by $s/N = P$, where s is the sample size and P is the percentage of the sample or batch size relative to the full point set. Given a large enough point set, and well defined true clusters, we can use a sample or batch of the points as low as 1% [29] and still retrieve a representative point set to converge closely to the optimal solution.

2.2.4 Other approaches

Other approaches involve making small changes to parts of the k-means algorithm. One can employ more complex strategies for generating initial centroids that enable the algorithm to converge more quickly and robustly than with random initial centroids [8, 4, 5]. One can also try pruning candidate centers during the assignment phase by estimating that some centroids are very close or very far from an observation, and thus eliminating the need to calculate the distance between all centroid/observation pairs [19]. Additionally, multiple k-means optimizations can be applied at once, although some may be incompatible and some may be more easily integrated together than others.

2.3 Competitive Algorithms

There are many variations of k-means that have emerged from Lloyds algorithm. Lloyds algorithm calculates a distance for every point in the point set to every cluster in every iteration, which results in a high running time if the number of observations and the number of clusters are large. We will discuss two competitive variations of k-means that succeed in optimizing the algorithm for speed on very large datasets by reducing the number of distance calculations required to obtain a clustering.

2.3.1 Mini Batch k-means

Mini Batch k-means [28] is a fast algorithm that differs from the conventional Lloyd's algorithm by working with a subset or mini batch of size b, of the full dataset in each k-means iteration. It is motivated by the need to be able to cluster datasets that are so large that they cant be processed in memory and comes at the cost of trading accuracy for speed. As b becomes very small relative to N, the algorithm cuts down significantly on the number of computations performed at the cost of cluster accuracy. It is shown to perform faster compared to other sequential implementations of Lloyds algorithm while maintaining similar cluster accuracy by running many more small, fast iterations [28].

2.3.2 Filtering algorithm

One of the strongest competitors to the Mini Batch k-means approach is the filtering algorithm [1, 19]. The filtering algorithm gains speed by using a k-d tree for querying the cluster assignments but is limited compared to Mini Batch k-means because web-scale data sets cannot be fit in memory in a k-d tree structure. Once a k-d tree has been built for the point set, the algorithm starts at the root node of the tree and propagates down through the child nodes. At each node, we keep a set of candidate centers: centroids that may be the closest centroids to some points in the subtree below that node. The root node starts with all centroids as candidate centers, and as candidates are eliminated, they are propagated to the child nodes. As the algorithm propagates down the tree, it stops at each node and computes the candidate center $z*$ that is closest to the midpoint of the bounding box of the node. For all other candidate centers, we compute whether or not they fall closer to any part of the bounding box than $z*$. If they do not, they are pruned from the candidate center set. If a node has only one center remaining in its candidate set, we assign all points in the nodes subtree to this remaining candidate center, without traversing the subtree. Similarly, if we reach a leaf node in the tree before pruning to one candidate center, the closest center is computed from the remaining candidates and the node's point sum and weight is assigned to that center. Since all but the last iteration of Lloyds algorithm assign points to a cluster only to update the cluster center, this assignment can be accomplished by adding the coordinate-wise sum of the points in the subtree and the number of points in the subtree to the coordinate-wise sum and number of points assigned to the cluster. At the end of each iteration, the centroid of each cluster can then be computed by dividing each coordinate of the point sum by the number of points. Similar to Lloyd's Algorithm, we keep track of the distortion between the observations and the centroids and terminate if the difference ratio between iterations does not meet the minimum distortion loss. The filtering algorithm performs well on low dimensional point sets and is well suited for online applications in which the data is already low dimensional or is reduced to a low dimensional embedding.

Chapter 3

Engineering an Efficient and Flexible K-D Tree Based K-Means Implementation

The starting point of our implementation of an efficient k-d tree based k-means algorithm was the implementation by Kanungo et al. [19]. We chose to engineer our own implementation for three reasons:

1. Our project required a flexible and modular implementation that allows for the combination of the basic k-d tree structure with different traversal strategies (sequential vs parallel).

2. The flexibility of our implementation allowed us to re-evaluate certain implementations choices made in the literature (how to define a nodes bounding box, how to choose the dimension in which to split a parents bounding box, and how to choose the splitting coordinate).

3. We aimed for an easy-to-understand implementation that uses modern C++ idioms while not sacrificing any performance compared to the more low-level and less modular implementation of Kanungo et al. We partially succeeded in this.

Our implementation follows three broad stages: initialization, tree building, and filtering.

3.1 Implementing K-D Tree K-Means

Our goal was to find the fastest k-means algorithm best suited to finding clusters in low dimensionality datasets, and optimize it by attempting to employ existing techniques, novel techniques, and optimized computations. We face several challenges in implementing this such as efficiently building the tree, managing memory on the tree, and implementing modular parts that could be adapted to both sequential and parallel traversals. We also needed to implement each of the tree building and traversal parameters that we wished to investigate from the literature.

3.1.1 Parameters

As the tree is being built, there are several potential choices of how we partition the point set at each node (the splitting procedure). The possibilities for a splitting procedure are to use a *midpoint split*, which splits the bounding box of a node into two boxes of the same size; a *median split*, which splits the bounding box into two boxes containing the same number of points; and a *sliding midpoint split* [24] which is similar to midpoint splitting but moves the splitting coordinate if necessary to ensure that at least one point is on each side of the split.

We also must choose which dimension of the point set we will partition by at every node. This is called the dimension selection procedure. The possibilities for the dimension selection procedure are *longest*, in which we always split on the longest dimension of the bounding box, and *round robin*, in which we start at the first dimension and sequentially cycle through all dimensions.

Finally, there are two ways we can determine the bounding box that encloses the point set after each node has been partitioned. The options for choosing the bounding box of a node are *true bounding box* in which the nodes bounding box is the smallest box that contains the points in that node, and *node bounding box*, which defines each nodes bounding box to be the box obtained by splitting its parents bounding box.

Additionally, there are the general user-provided parameters that apply to the k-means algorithms in general. These are the number of clusters to partition into, the minimum distortion loss between iterations, the maximum number of iterations to run, and the maximum number of iterations per run. These arguments will be described in more detail below.

Within the k-means algorithm, we define an iteration as one run of the assignment and update steps. We define a run as a group of iterations which we use to track convergence progress. The *minimum distortion loss* defines a criterion to terminate the algorithm once the objective function (distortion) does not decrease sufficiently over a run. The distortion loss over the run is defined as $\frac{(base\ distortion\ -\ current\ distortion)}{base\ distortion}$, where the *base distortion* is the distortion taken at the beginning of the run, and the *current distortion* is the distortion taken after the current iteration finishes. The *maximum iterations per run* defines the maximum number of iterations between which we wait to update the base distortion for comparison to the current iterations distortion.

The algorithm terminates if the distortion loss at the end of a run has stayed below the required *minimum distortion loss*. The algorithm also terminates after a set number of iterations, *maximum number of iterations* if the first termination criterion has not yet been met.

3.1.2 Representation of the Point Set

In the initialization stage, we read the user defined data set into a *PointSet* data structure. Our *PointSet* structure is a vector of n-dimensional *Points*. A *Point* is defined as a structure of n coordinates, where each node represents one dimension of that data point. Each node is a union of a double: the value of the data point in that dimension, and a pointer to the value in the next dimension of that point, if it exists. The pointer to the next node also allows us to efficiently manage memory allocations, which will be described in detail later in subsection 3.1.5. We also keep a *PointSequence*, which keeps n vectors of indices sorted by each dimension of the referred *PointSet*. A *PointSequence* allows us to efficiently find the split point when building the tree, without having to scan the entire point set at each split.

3.1.3 Tree Building

For the filtering algorithm, we must modify the basic k-d tree to include additional information about the points stored at each nodes subtree. In addition to its bounding box, every node in the k-d tree stores the number of points in its subtree, the coordinate-wise sum of these points, and the sum of the squares of all coordinates of these points $SumSquares = \sum_{i=0}^{node_N} \sum_{j=0}^{n} (x_{i,j} * x_{i,j})$ where $node_N$ is the number of points contained in the nodes subtree and $x_{i,j}$ is the jth coordinate of the ith point in the subtree. These values will be used by the filtering algorithm to avoid traversing a subtree whose points are all closest to the same cluster centers.

We build the k-d tree recursively. We start with the root node, whose bounding box is chosen as the smallest box that contains the entire point set. For each node that contains more than one point, we split its bounding box and point set into two smaller boxes and the two point sets contained in them according to the chosen dimension selection, dimension splitting and bounding box selection rules. We associate these two bounding boxes with the nodes chosen children and recursively build the subtrees with these two children as roots. Once these two recursive calls return, the children store their point counts, coordinate-wise point sums, and sums of squares of the point coordinates. The corresponding values for the current nodes can then be computed by summing the values of the two children (e.g. the coordinate-wise point sum for the current node is the coordinate-wise sum of the childrens coordinate-wise sum).

3.1.4 Filtering

Our implementation of the filtering algorithm (Algorithm 1) is iterative, simulating the recursive traversal of the tree by explicitly maintaining a stack of nodes whose subtrees still need to be traversed. This will be crucial for parallelizing the algorithm using a work sharing scheduler.

The algorithm maintains the number and coordinate-wise sum of all points assigned to a given cluster (see lines 14,21). Once the tree traversal is finished, the new center of each cluster can then be computed as $center = \frac{pointsum}{numpoints}$ (see line 24). The traversal computes this information for all clusters and also computes the distortion of the current set of cluster centers, as a basis for deciding whether the required minimum distortion loss has been achieved in the current run.

First consider collecting the points in each cluster: If the current node is a leaf (see lines 19-22), we select the cluster center closest to it, add one to this centers node count and the point to the centers point sum. If the current node is an internal node, we first remove all candidate cluster centers that cannot be closest to any point in the nodes subtree (see lines 7-12 and Algorithm 2). If this leaves only one remaining cluster center, all points in this subtree need to be assigned to this cluster center. We achieve this by adding the current nodes point count and point sum to the cluster centers point count and point sum. If there is more than one candidate cluster center that could be closest to points in this subtree, we recurse on the nodes children with the reduced set of cluster centers.

To prune the nodes, we follow a procedure outlined in Algorithm 2. While not

Algorithm 1 K-means Filter

```
1: INPUT: $TS \leftarrow$ traversal stack;
2: while $TS$ is not empty do
3:     $tn \leftarrow$ tree node at top of traversal stack;
4:     $B \leftarrow tn$'s bounding box;
5:     $C \leftarrow$ the candidate center set of $tn$;
6:     if $tn$ is an internal node; then
7:         $c_{close} \leftarrow$ the closest center in $C$ to $B$'s middle;
8:         for all $c \in C \mid c \neq c_{close}$ do
9:             if Prune($c$, $c_{close}$, $B$) then
10:                $C \leftarrow C \setminus c$;
11:            end if
12:        end for
13:        if $|C| == 1$ then
14:            Record $tn$'s weight, sum of points, and sum of squared points for $c_{close}$;
15:        else
16:            Push $tn$'s left child and $C$ to traversal stack;
17:            Push $tn$'s right child and $C$ to traversal stack;
18:        end if
19:    else
20:        $c_{close} \leftarrow$ the closest center in $C$ to $tn$s point;
21:        Record $tn$'s weight, sum of points, and sum of squared points for $c_{close}$;
22:    end if
23: end while
24: Calculate the distortion and update center set;
25: return updated center set;
```

particularly intuitive at first glance, this procedure simply determines if every part of the nodes bounding box, B, is closer to c_{close}, the candidate center closest to the middle of B than a candidate center to be tested, c. If this test is true, we can infer that c_{close} will always be the closer candidate center, and prune c from the candidate set of that node.

Algorithm 2 Prune(c, B, c_{close})

```
1: INPUT: candidate center to test for pruning $c$, the nodes bounding box $B$, candidate $c_{close}$ closest to the middle of $B$;
2: $candProd \leftarrow 0$;
3: $boxProd \leftarrow 0$;
4: for all dimensions of the candidate centers $d$ do
5:     $candCompare = c[d] - c_{close}[d]$;
6:     $candProd$ += $candCompare^2$;
7:     if $candCompare > 0$ then
8:         $boxProd$ += $(B_{upper}[d] - c_{close}[d]) * candCompare$;
9:     else
10:        $boxProd$ += $(B_{lower}[d] - c_{close}[d]) * candCompare$;
11:    end if
12: end for
13: return $candProd >= boxProd * 2$;
```

To verify whether or not the condition to prune c is met, we consider the relationship between the dot products of two vectors. We observe that a point a is closer to c than c_{close} if an only if $(a-c)\cdot(a-c) < (a-c_{close})\cdot(a-c_{close})$. We can rearrange this to be:

$$\begin{gathered}
(a-c)\cdot(a-c) < (a-c_{close})\cdot(a-c_{close}) \\
a^2 - 2ac + c^2 < a^2 - 2ac_{close} + c_{close}^2 \\
-2ac + c^2 < -2ac_{close} + c_{close}^2 \\
-2ac + c^2 + (-2cc_{close} + c_{close}^2) < -2ac_{close} + c_{close}^2 + (-2cc_{close} + c_{close}^2) \\
c^2 - 2cc_{close} + c_{close}^2 < 2ac - 2ac_{close} - 2cc_{close} + 2c_{close}^2 \\
(c-c_{close})\cdot(c-c_{close}) < 2(a-c_{close})\cdot(c-c_{close})
\end{gathered}$$

To relate this to Algorithm 2, $(c-c_{close})\cdot(c-c_{close})$ is equivalent to $candProd$ (see line 6) and $(a-c_{close})\cdot(c-c_{close})$ is equivalent to $boxProd$ (see lines 8,10). It follows that the final derivation is equivalent to line 13. To ensure that this is satisfied for all points a in B, we will find the point a in B that maximizes *boxProd*. This will be a corner of B, and is found by checking the direction of the vector of *candCompare* in each dimension. If the vector direction is positive, the upper side of B in that dimension will maximize *boxProd*, and vice versa for a negative direction (see lines 7-11).

To calculate the distortion of the current set of cluster centers, we need to sum $\sqrt{\sum_{p\in P_c}\sum_{i=1}^{n}(p_i - c_i)^2}$ over all cluster centers, where P_c is the set of points assigned to c. Calculating the roots and summing them is trivial once we have calculated $\sum_{p\in P_c}\sum_{i=1}^{n}(p_i - c_i)^2$ for each cluster center. To calculate this sum efficiently, we observe that:

$$\begin{aligned}\sum_{p\in P_c}\sum_{i=1}^{n}(p_i - c_i)^2 &= \sum_{p\in P_c}\sum_{i=1}^{n}(p_i^2 - 2p_ic_i + c_i^2)\\ &= \sum_{p\in P_c}\sum_{i=1}^{n}p_i^2 - 2\sum_{p\in P_c}p_i\sum_{i=1}^{n}c_i + |P_c|\sum_{i=1}^{n}c_i^2\end{aligned}$$

The first term is equivalent to the total sum of coordinate squares of all points assigned to the cluster. Thus, the traversal also calculates this sum for all clusters.

The second term is -2 times the inner product of c and the coordinate-wise sum of the points assigned to this cluster and can thus be computed in constant time (assuming the number of coordinates is constant), Similarly, the third term can be computed in constant time from the cluster center and number of points in the cluster.

3.1.5 Optimizations

By profiling the algorithm, we found that a significant amount of time was spent on memory allocations. Most of this time was spent allocating space for the coordinates of temporary point variables created especially during tree building. By making optimizations on these operations, we save approximately an order of magnitude in average run time of our algorithm. We implemented an optimization for this by building our own memory manager for the *Point* class, called the *PointAllocator*.

The *PointAllocator* initially allocates a large chunk of memory for *Points* to eliminate the overhead of making many small allocations. It manages this chunk of memory as a linked list of available *Point*-sized chunks, built on top of the array of memory. When a *Point* needs to be created, the *PointAllocator* gives the first *Point*-sized block of memory from the front of the free list for the new *Point* and moves the front pointer of the free list to the successor of the referred block. When a *Point* goes out of scope or is destroyed, the destructor of the *Point* returns that memory to the head of the free list. If the free list runs out of allocated memory, the allocator allocates another large chunk to point to as the front of the free list.

Another significant source of run time overhead was in keeping track of our recursive stack of nodes to visit and the associated candidate centers during the filtering algorithm. Our original implementation used C++ standard library implementation of stacks and contained a significant amount of built-in *emplace* and *push_back* operations. We found these operations to be a bottleneck in our application, likely stemming from overhead in boundary checking and allocation increases as the stack grows and shrinks in size. For this reason, we implemented our own stack that would pre-allocate the stack size we require, could keep track of both the node traversal stack and the candidate center stack at the same time, and left stack safety largely in our own hands. We do this using low-level arrays and pointer manipulation. We use a separate stack for the traversal order and candidate centers for simplicity and ease of implementation. A visualization of how the stacks are managed is shown in Figure 3.1. Each element pushed to the stack contains a pointer to a node, the number of candidate centers that node has, and a pointer to the candidate center buffer at the location we keep that nodes candidate centers. On each stack push or pop operation, we also push or pop the candidate centers on or from the candidate center buffer. This allows us to quickly have access to a node, and its candidate centers as we traverse the tree. We pre-allocate the maximum stack size as N, as we can never have more than N nodes in our stack at once during a traversal, and we pre-allocate memory for $N * k$ integers in the candidate buffer.

3.2 Experimental Evaluation

Once we had optimized our implementation of the filtering algorithm, we ran exploratory experiments to determine what parameters were best suited for the shape of our target datasets. After we found the best algorithm parameters, we then tested

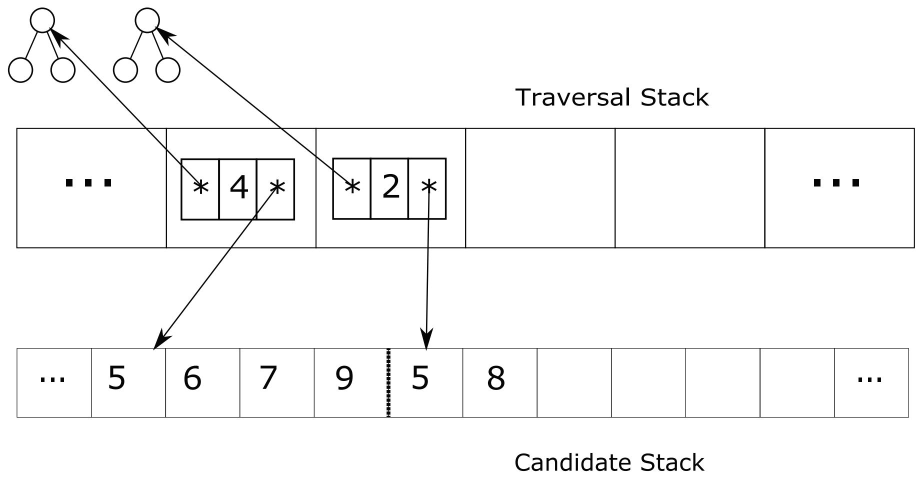

Figure 3.1: An example of our pre-allocated traversal and candidate stack. To keep track of nodes and their associated information as we traverse, we use these co-operative stacks. The structure we push to the traversal stack contains a pointer to the node to traverse, the number of candidate centers it has remaining, and a pointer to the candidate stack which holds the indices of those candidates. In this example, the node on the top of the stack has 2 candidates left: centers 5 and 8. The node second on the stack has 4 candidate centers left: 5, 6, 7, and 9.

the algorithm against other competitive algorithms to determine how their running times compare. For all of these experiments, we fixed the maximum number of iterations to 100, the minimum distortion loss to 0.1, and ran each experiment with the number of cluster centers, k, at 10 and 50. To eliminate variance, we chose the same set of initial cluster centers for all methods, and we repeat each test 10 times (with different initial cluster centers in each run) and record the mean and standard deviation.

The machine we tested on was running CentOS 7 version 3.10.0-862.el7, with an Intel i7-3820 CPU @ 3.60GHz with 8 cores. All of our implementations were coded in C++ to the C++11 standard and were compiled using gcc version 4.8.5 at optimization level 3. We tested the algorithms in our experiments on the ASRS100k data set, and on a Reddit dataset, both reduced to 2 dimensions dtCSM [29]. ASRS100k is a dataset of 100,000 sanitized aviation incident and operations reports from the Aviation Safety Reporting System online database reduced to 2 dimensions. The Reddit dataset is a dataset of 16,000,000 comments from the social discussion website Reddit.com reduced to 2 dimensions. We also downsample the Reddit dataset to 1,000,000 points, and 4,000,000 points so we can better observe how the algorithms we test scale.

3.2.1 Parameter Tuning

For our exploratory experiment, we compared the overall running time for each combination of splitting procedure, dimension selection procedure, and bounding box type. Since we are using consistent initial random centers across the experiment, the objective function converges to the same value regardless of splitting procedure, dimension selection procedure, and bounding box type. For this reason, we do not report the objective function for this experiment.

Table 3.1 shows that a combination of midpoint or sliding midpoint, a round robin dimension splitting procedure, and a true or node bounding box provides the fastest and most consistent results. The results for the experiments on each of the 4 possible combinations of these parameters produce extremely similar results across the board. Based on run time consistency, we will choose midpoint, round robin, and node as the best parameters for our experiments going forward.

3.2.2 Comparison With Competitors

Our first competitor is the implementation of the filtering algorithm by Kanungo et al. [19]. Table 3.2 shows that our tree building is slightly slower than Kanungo et al. This is likely because of the optimizations we took to pre-allocate memory for *Points* and traversal stacks. In our implementation, we take extra time at tree building time to pre-allocate large chunks of memory that save us time later in the algorithm, compared to Kanungo et al. which allocates dynamically as needed.

This theory is supported by evidence in Table 3.3. We see that our implementation significantly outperforms Kanungo et al. in query time. Here we define querying as the portion of the algorithm where we filter and update the candidate centers until termination. The total running times of both implementations, including both filtering, tree building, and data reading, are shown as part of Table 3.4. Our results show that our implementation outperforms Kanungo et al. by a substantial margin except on the full Reddit data set.

Finally, we compare the overall running time to Mini Batch k-means and k-d tree k-means by Kanungo et al. in Table 3.4. For this experiment, we run Mini Batch

Table 3.1: Comparison of total run time of our K-D tree parameter combinations in seconds [mean (sd)]. The three fastest run times for each dataset/k are highlighted. The columns Split, D.S., and B.B. define the dimension splitting rule, the dimension selection rule, and the bounding box type respectively. Under the dimension split rule (Split) column, Med., Mid., and S. Mid. stand for median split, midpoint split, and sliding midpoint split respectively. Under the dimension selection rule column, RR and Long stand for round robin selection and longest dimension selection respectively.

k	**Split**	**D. S.**	**B.B.**	**Dataset**			
				ASRS100k	Reddit1M	Reddit4M	Reddit16M
10	Med.	RR	True	0.362 (<0.01)	3.529 (0.06)	14.496 (0.02)	65.717 (1.34)
			Node	0.364 (<0.01)	3.515 (0.01)	14.493 (0.04)	65.903 (1.36)
		Long	True	1.279 (0.05)	8.056 (0.12)	33.489 (0.53)	145.763 (2.99)
			Node	0.364 (0.03)	3.373 (0.02)	14.234 (0.05)	64.928 (1.27)
	Mid.	RR	True	0.327 (0.06)	**2.368** (0.01)	**9.772** (0.03)	60.444 (2.88)
			Node	**0.294** (<0.01)	2.372 (0.01)	**9.777** (0.02)	**58.762** (2.35)
		Long	True	1.327 (0.06)	8.028 (0.27)	41.226 (1.24)	-
			Node	0.298 (0.05)	2.653 (0.09)	14.379 (0.05)	-
	S. Mid.	RR	True	0.304 (0.04)	**2.367** (0.01)	**9.798** (0.03)	**59.170** (2.29)
			Node	**0.293** (<0.01)	**2.364** (0.02)	9.834 (0.17)	**58.136** (2.94)
		Long	True	1.330 (0.08)	8.109 (0.33)	41.167 (1.13)	-
			Node	**0.294** (0.04)	2.627 (0.05)	14.344 (0.04)	-
50	Med.	RR	True	0.549 (0.03)	4.543 (0.01)	16.383 (0.06)	69.330 (1.15)
			Node	0.536 (<0.01)	4.566 (0.06)	16.419 (0.08)	69.747 (1.23)
		Long	True	2.920 (0.04)	15.565 (0.52)	63.059 (1.84)	265.767 (18.13)
			Node	0.508 (<0.01)	4.284 (<0.01)	15.830 (0.05)	67.808 (1.34)
	Mid.	RR	True	**0.440** (0.01)	3.356 (0.05)	**11.462** (0.03)	**62.134** (2.43)
			Node	**0.439** (<0.01)	**3.345** (0.04)	**11.476** (0.02)	63.209 (2.42)
		Long	True	2.955 (0.03)	15.809 (0.42)	70.732 (1.58)	-
			Node	**0.434** (<0.01)	3.516 (0.01)	15.872 (0.09)	-
	S. Mid.	RR	True	0.450 (0.03)	**3.327** (0.01)	11.518 (0.16)	**62.431** (2.54)
			Node	0.442 (<0.01)	**3.332** (0.01)	**11.478** (0.04)	**62.235** (2.81)
		Long	True	2.955 (0.03)	15.781 (0.41)	70.499 (1.34)	-
			Node	0.442 (0.03)	3.507 (0.01)	15.914 (0.05)	-

D. S. = Dimension Selector, B.B. = Bounding Box, Med. = Median, Mid. = Midpoint, S. Mid. = Sliding Midpoint, RR = Round Robin, Long = Longest

Table 3.2: Comparison of sequential filtering algorithm k-means implementations tree build times in seconds (total time) [mean (sd)]

k	**Algorithm**	**Dataset**			
		ASRS100k	Reddit1M	Reddit4M	Reddit16M
10	Crowell	0.089 (<0.01)	0.960 (<0.01)	4.555 (0.01)	30.953 (1.47)
	Kanungo et al.	0.064 (<0.01)	0.797 (0.05)	3.837 (0.15)	21.6 (1.07)
50	Crowell	0.088 (<0.01)	0.959 (<0.01)	4.555 (0.02)	31.664 (1.84)
	Kanungo et al.	0.059 (<0.01)	0.766 (<0.01)	3.792 (0.02)	21.200 (0.63)

Table 3.3: Comparison of sequential filtering algorithm k-means implementations query times in seconds (total time) [mean (sd)]

k	**Algorithm**	**Dataset**			
		ASRS100k	Reddit1M	Reddit4M	Reddit16M
10	Crowell	0.108 (<0.01)	0.542 (0.01)	1.778 (0.02)	11.275 (1.31)
	Kanungo et al.	0.564 (0.01)	2.735 (0.86)	9.101 (0.14)	18.253 (0.37)
50	Crowell	0.255 (<0.01)	1.504 (<0.01)	3.476 (0.02)	15.179 (1.07)
	Kanungo et al.	1.777 (0.08)	12.378 (0.13)	24.954 (0.50)	50.382 (0.73)

k-means at the suggested batch size of 1000. Mini Batch k-means is also run for 1000 iterations (10 times more iterations than the k-d tree methods) to ensure that Mini Batch k-means obtains comparable objective function values to the other methods. This number of iterations for Mini Batch k-means was determined to provide convergence to a similar objective function as the k-d tree based methods in a minimal number of iterations. Despite this, Mini Batch k-means still occasionally performs very poorly in terms of the objective function, especially with a larger k. This effect was not found to be improved by increasing the number of iterations. The mean objective function values of each implementation are shown in Table 3.5. However, even with some unreliability in objective function, Mini Batch k-means performs significantly faster than its competitors as N becomes large and could potentially be run multiple times to ensure a stable result while still having a faster running time than other methods. Kanungo et al. was run with the same parameters as our best configuration where applicable.

From this experiment, we find that our implementation succeeds at being consistently competitive or faster in speed with Kanungo et al. but is quickly outmatched by Mini Batch k-means as N becomes large. This is reasonable, given that Mini

Table 3.4: Comparison of k-means method run times in seconds (total time) [mean (sd)]

k	**Algorithm**	**Dataset**			
		ASRS100k	Reddit1M	Reddit4M	Reddit16M
10	Crowell	0.294 (<0.01)	2.372 (0.01)	9.777 (0.02)	58.762 (2.35)
	Kanungo et al.	0.726 (0.02)	4.377 (0.91)	16.493 (0.20)	53.616 (0.76)
	Mini Batch	0.608 (0.09)	1.883 (0.20)	3.727 (0.36)	10.395 (0.33)
50	Crowell	0.439 (<0.01)	3.345 (0.04)	11.476 (0.02)	63.209 (2.42)
	Kanungo et al.	1.935 (0.09)	14.024 (0.13)	32.258 (0.51)	85.662 (0.91)
	Mini Batch	1.251 (0.06)	2.395 (0.46)	4.355 (0.40)	11.350 (0.74)

Table 3.5: Comparison of k-means final objective function values rounded to nearest whole number [mean (sd)]

k	**Algorithm**	**Dataset**			
		ASRS100k	Reddit1M	Reddit4M	Reddit16M
10	Crowell	6,459,124 (21,730)	58,227,547 (495,959)	232,918,717 (1,991,585)	929,136,773 (6,118,416)
	Kanungo et al.	6,459,124 (21,730)	58,227,547 (495,959)	232,918,717 (1,991,585)	929,136,773 (6,118,416)
	Mini Batch	16,471,748 (8,926,131)	38,629,240 (13,101,570)	226,843,930 (142,177,400)	848,218,700 (259,023,655)
50	Crowell	1,290,069 (10,222)	11,959,686 (85,122)	47,734,364 (350,608)	191,817,994 (1,372,099)
	Kanungo et al.	1,290,069 (10,222)	11,959,686 (85,122)	47,734,364 (350,608)	191,817,994 (1,372,099)
	Mini Batch	>100,000,000*	>100,000,000*	>100,000,000*	>100,000,000*

*About half of Mini Batch k-means runs on 50 centers converge to poor local solutions and give extremely large objective function values, which skew the mean value. The runs that appear to converge give objective function values similar to the k-d tree k-means implementations. Increasing the number of iterations does not prevent this.

Batch k-means is independent of N, except in data reading cost, whereas the k-d tree methods time complexity scales with N. Mini Batch k-means sampling provides it a strong advantage in running time over other methods as the data size increases.

We had identified another potentially competitive implementation to our algorithm that uses k-d trees and a form of parallelism with dynamic load balancing [16]. This parallelizes the work done to traverse the tree in a comparable way we will parallelize our traversal computations (described later in subsection 5.3.3 Work Sharing Traversal). Their implementation tracks the run time and number of distance calculations performed by each thread and attempts to balance the computation of each thread by having the slower threads donate work to the faster threads until all threads run a comparable amount of work. Their experiments showed significant speed-ups compared to a parallel implementation of Lloyds Algorithm on moderately large and moderately dimensional data. We were unfortunately unable to obtain their implementation, nor discern their dynamic load balancing approach in enough detail to implement this ourselves. While we believe our approach is fine-grained enough to achieve near-perfect load balance, we regret that their implementation was not available, as it would been an interesting comparison to make.

It should also be noted that we do not explore using a k-d tree with Mini Batch k-means in determining the impact of k-d trees. In Mini Batch k-means, we repeatedly take a small sample batch of points from the entire dataset and compute our cluster centroids based on the results given by the samples. This does not apply well to a k-d tree approach, in which we seek to build a k-d tree once on the entire dataset at an additional cost to gain speed in determining the closest candidate centroids in the future. In Mini Batch k-means, we must either build a tree for every sample, which is likely to give more overhead in time than simply not using a tree, or build a tree on the full dataset initially and somehow adapt it for use on a sample, which would still not guarantee any time saved over the Mini Batch k-means implementation.

3.2.3 Effect of Dimensionality

One drawback of the filtering algorithm is its diminishing speed-ups over Lloyd's Algorithm as the dimensionality of the data increases. Lloyd's Algorithm has a time complexity of $\mathcal{O}(nkN)$ on an n-dimensional dataset, which scales linearly with n. When considering the filtering algorithm, Kanungo et al. [19] have shown that when ignoring the spread and distribution of the underlying data, the filtering algorithm has complexity $\mathcal{O}(k\sqrt{n}^n + k2^n \log N)$. This scales exponentially with the dimensionality of the data. This difference in scaling poses the question of how high-dimensional the data must get until Lloyd's algorithm becomes faster than the filtering algorithm in practice.

From the literature, we note experiments that have been run to determine this. Pelleg and Moore [27] ran an experiment exploring this using a synthetic dataset of 20000 data points, dimensions ranging from 2–8, and run on 40 cluster centers. This experiment was duplicated and expanded on by Kanungo et al. [19], using data ranging from 2–35 dimensions. Both of these experiments demonstrated that the filtering algorithm performs best on data of dimensionality 2–10, and will generally outperform Lloyd's algorithm on data having dimensions up to the mid 20s.

Chapter 4

Impact of Sampling on K-Means

We have already shown that we can build a competitively fast k-means implementation using a k-d tree. We have also shown that sampling-driven Mini Batch k-means is a strong performer. Given this, it seems natural to try to utilize both k-d trees and sampling together in one algorithm to achieve even greater run time performance. While we are unable to combine Mini Batch k-means batch sampling effectively with our k-d tree implementation as discussed in Chapter 3, we can still utilize conventional sampling (applying our algorithm to a subset of the data) and compare the trade-offs in accuracy at various sampling rates.

4.1 Sampling Approaches

We initially considered naively combining Mini Batch k-means sampling strategy with the filtering algorithm but determined this was unlikely to outperform either method individually. The biggest challenge with this is that we are unlikely to reap the benefits of both k-d trees and the type of sampling used in Mini Batch k-means. Mini Batch k-means achieves quick convergence by running multiple quick iterations on random samples of the data. By using multiple samples, rather than one single sample, the candidate centers become more representative of the greater dataset with fewer iterations run. This is not particularly compatible with k-d trees, as the advantage gained by using k-d trees is the ability to re-use the tree across multiple iterations and eliminate unnecessary distance computations. With a k-d tree-based approach, we incur an overhead to build the tree, knowing that we will save time over the lifetime of the algorithm. To combine Mini Batch k-means naively with k-d tree approaches, we would need to rebuild the tree for each random mini batch sample, thus losing the advantage of tree re-use and likely incurring an overhead that will not improve the computation time on an already small batch sample. Another alternative considered to make this combination of algorithms work is to build one k-d tree on the entire dataset that is capable of performing query operations on only a sample of the data. It is difficult to conceive how this would work, but it is unlikely that the additional computation spent to correctly prune and query on only a sample using the k-d tree would be faster than simply running Mini Batch k-means without a k-d tree, mainly because the batch size of Mini Batch k-means is so small that using a data structure, especially one built on the entire data set, is unlikely to yield performance gains. For this reason, we instead opt to use a conventional sampling approach, where we take a small sample of the dataset once, at the beginning of the algorithm, build a k-d tree on the sample, and compute cluster centers from this sample as usual. This is followed by a final iteration that assigns each point in the full data set to a cluster. In experimenting with sampling, we sought to find if sampling gives significant speedup at any particular sampling rate, as well as determine the effect of sampling on the objective function. We used our most promising parameter set from our previous experiment: midpoint splitting, longest dimension splitting, and node bounding box. While it is known that sampling is an effective method to speed up a standard implementation of k-means, which does not use any data structures to speed up the iterations, it is unclear whether sampling is equally effective to speed up k-d tree-based k-means. Since a standard implementation's cost depends linearly on the input size in each iteration, the speed-up is proportional to the sampling rate. The filtering algorithm, on the other hand, does not explore the whole data set in each iteration. Since the tree structure is expected to summarize large groups of points, the main effect of sampling may be only that each iteration of the filtering algorithm visits roughly the same number of tree nodes, with fewer points in the sample represented by each visited node. Our experiments in this section explore exactly how much the filtering algorithm can benefit from sampling.

4.2 Exploratory Results

To observe the effects of sampling on our algorithm and data, we ran the algorithm at multiple sampling rates to measure the total running time and objective function. At each sampling rate, we ran our algorithm 10 times, with a new sample taken in each run. This is done to eliminate the outliers in the results that could be found in sampling extremely favourable or unfavourable samples from the data set. For our experiment, we sampled from our target data sets at rates of 1%, 10%, 20%, 30%, 40%, and 50% to observe a broad range of sampling rates and attempt to identify where the optimal trade-offs are achieved. We can then express the results as a ratio of our sampled running time and objective function to the run time and objective function on the full data set. This allows us to more easily interpret the results relative to a non-sampled run. In this experiment, the total running time, including loading in data and tree building time is taken into account because we will be reading in less data, and building a smaller tree at each sampling rate, meaning these times will also vary between sampling rates. We also pay special attention to the calculation of the objective function for this experiment. To calculate the objective function value, we run the algorithm to completion with the sample as the input, and then re-calculate the objective function from the calculated final centers on the un-sampled, complete dataset.

From Table 4.1, we see that a sampling rate of 10% appears to provide an objective function that does not differ greatly from the baseline objective function of a full run, while providing a significant speed-up in total running time. In many cases, especially as N increases, a 1% sample also becomes viable. We can put these numbers into better context by visualizing their ratios relative to the baseline running time and objective function.

In Figure 4.1, we see that with 10 centers, the objective function consistently falls within a very small percentage of the baseline in all our sampling cases. The only exception to this is a 1% sample on the ASRS100k dataset, where our results are inconsistent between runs, but are still only approximately 10% worse in objective function on average. In every other case, the mean sampled k-means run falls well within 5% of the baseline objective function value with a small relative standard deviation. This means a sampled run is consistently almost as accurate as a full run, while providing speed ups that scale with N, to more than 120 times faster than baseline on 16 million data points. We note that in the best case, our tree builds in $\mathcal{O}(N \log N)$, and a reduction of our input size by r times gives us in $\mathcal{O}(N/r(\log N - \log r))$, which can result in tree building speedups of more than r. As seen in the previous chapter in Table 3.2 and 3.3, the tree building is the bottleneck of our application. This shows that because the tree building is the majority of the run

Table 4.1: Objective Function and Total Running Time for Sampling on K-D tree k-means at optimal parameters.

k	**S. Rate**	**Dataset**			
		ASRS100k		Reddit1M	
		Obj. Func.	*Running Time*	*Obj. Func.*	*Running Time*
10	**Baseline**	**6459123**	**0.294 (<0.01)**	**58227547**	**2.372 (0.01)**
	1%	7158916	0.047 (0.03)	58203808	0.097 (0.01)
	10%	6516898	0.062 (0.01)	58130160	0.329 (0.02)
	20%	6454605	0.085 (<0.01)	58064549	0.577 (0.02)
	30%	6474484	0.128 (0.03)	57952075	0.818 (0.03)
	40%	6503355	0.146 (0.01)	58220576	1.071 (0.01)
	50%	6432264	0.192 (0.04)	58074156	1.463 (0.47)
50	**Baseline**	**1290069**	**0.439 (<0.01)**	**11959686**	**3.345 (0.04)**
	1%	1509983	0.033 (<0.01)	12503539	0.110 (0.01)
	10%	1344443	0.153 (0.03)	12028883	0.501 (0.02)
	20%	1319472	0.177 (0.03)	12026018	0.891 (0.02)
	30%	1310294	0.242 (0.07)	12008954	1.300 (0.04)
	40%	1308954	0.245 (0.01)	12052686	1.680 (0.05)
	50%	1310543	0.304 (0.04)	12016832	1.998 (0.05)
		Reddit4M		Reddit16M	
		Obj Func	*Run Time*	*Obj Func*	*Run Time*
10	**Baseline**	**232918716**	**9.777 (0.02)**	**929136773**	**58.762 (2.35)**
	1%	232119772	0.164 (0.03)	926945995	0.474 (0.01)
	10%	232863299	1.084 (0.02)	929050405	4.188 (0.30)
	20%	232217727	2.050 (0.05)	929155025	8.226 (0.22)
	30%	232373506	3.023 (0.07)	929210085	12.764 (0.91)
	40%	232314740	4.050 (0.05)	927147140	17.496 (0.67)
	50%	232747560	5.103 (0.12)	929498355	21.838 (0.41)
50	**Baseline**	**47734364**	**11.476 (0.02)**	**191817994**	**63.209 (2.42)**
	1%	48289449	0.258 (0.01)	192312547	0.745 (0.02)
	10%	48210869	1.634 (0.06)	191483953	5.258 (0.09)
	20%	48075535	2.905 (0.03)	192179029	10.020 (0.78)
	30%	47731468	4.087 (0.08)	191334346	14.633 (0.69)
	40%	47879945	5.243 (0.09)	192164693	19.311 (0.40)
	50%	48124716	6.435 (0.15)	192037300	24.337 (0.70)

time, and we may see speed-ups greater than the sampling reduction in tree building, the overall speed-up may be greater than the sampling reduction.

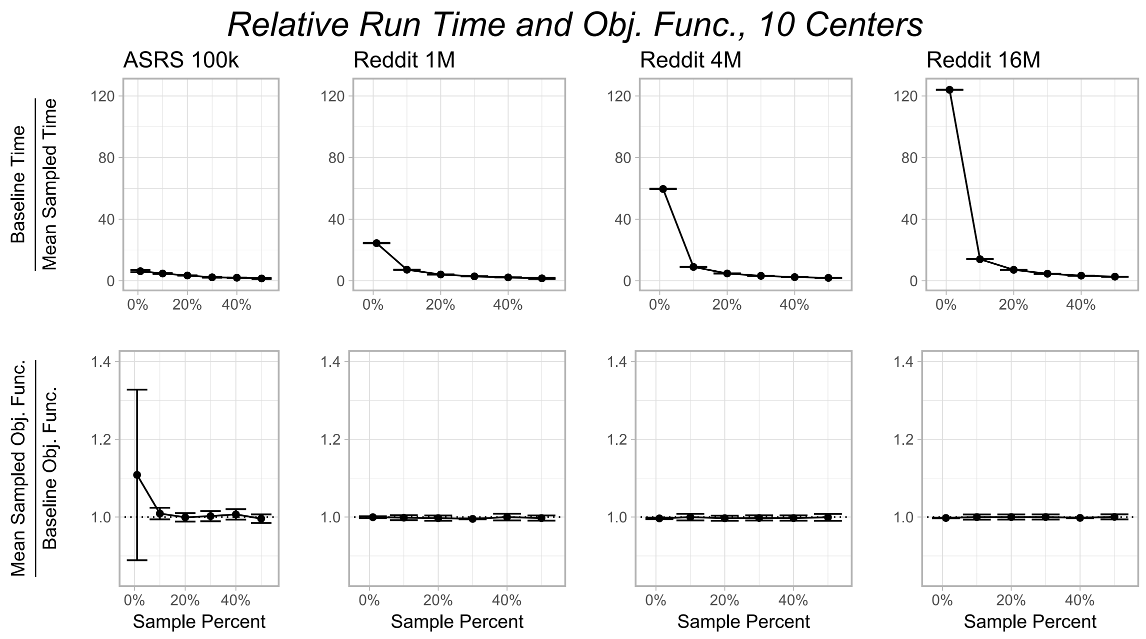

Figure 4.1: Relative running times and objective function values at different sampling rates compared to their baselines across each dataset on 10 centers. A value of x for the running time indicates that the sample is x times faster than the baseline. A value of x for the objective function indicates that the samples objective function value is x times larger than that of the baseline. The error bars represent $\pm$ one relative standard deviation from the mean sampled running time/objective function value.

On 50 centers, we see a similar trend with sampling as with 10 centers. In Figure 4.2, the mean sampled objective function value is still within 5% of the baseline in all tests except the 1% sample on the ASRS100k dataset, and still has a relatively small standard deviation. Compared to 10 centers, the relative speedups are not quite as large, but still significant. We see a maximum mean speed-up of approximately 85 times faster than baseline on a 1% sample of 16 million points. The difference in speedups between 10 and 50 centers comes from the querying portion of the algorithm, and not the tree building.

This experiment provides good evidence that sampling is a viable method for improving the running time of k-d tree based k-means without sacrificing significantly

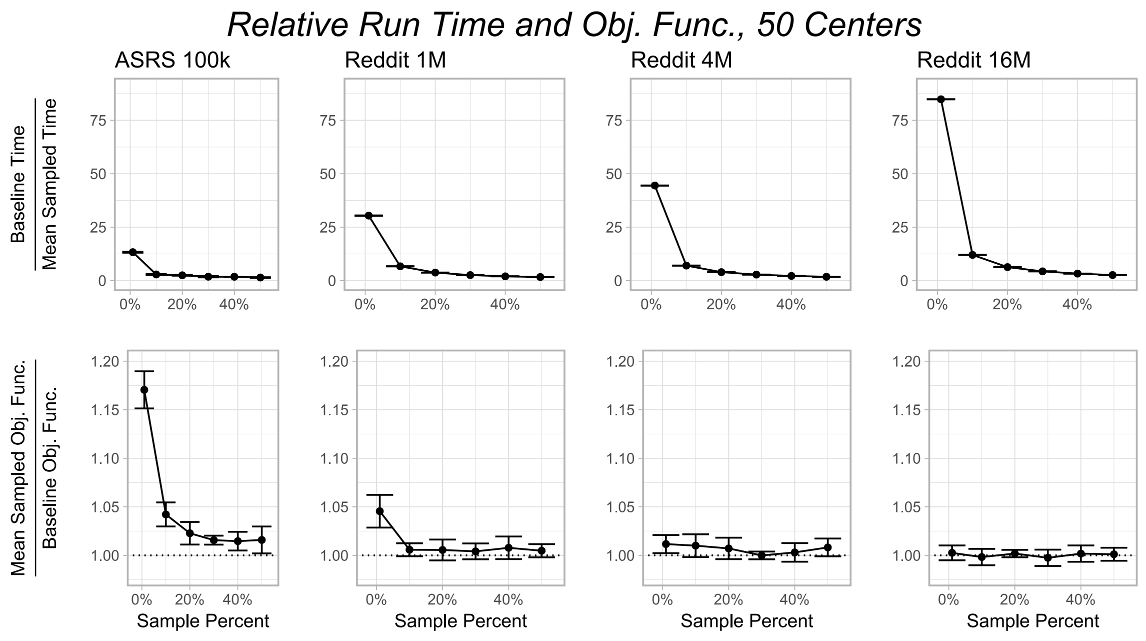

Figure 4.2: Relative running times and objective function values at different sampling rates compared to their baselines across each dataset on 50 centers. A value of x for the running time indicates that the sample is x times faster than the baseline. A value of x for the objective function indicates that the samples objective function value is x times larger than that of the baseline. The error bars represent $\pm$ one relative standard deviation from the mean sampled running time/objective function value.

in the accuracy of the final clustering, which is often good enough for most applications. As the data becomes large, we can obtain a suitable cluster set within 5% of our objective function on the full dataset, while obtaining significant speed gains that scale with the input size N. In most cases, these speed-ups improve the running time to be better than Mini Batch k-means while maintaining stability of the cluster solution that Mini Batch k-means does not (see Figure 4.3). While adding sampling to the k-means algorithms greatly improves running time at a minor cost in objective function, there are still other ways to potentially gain speed.

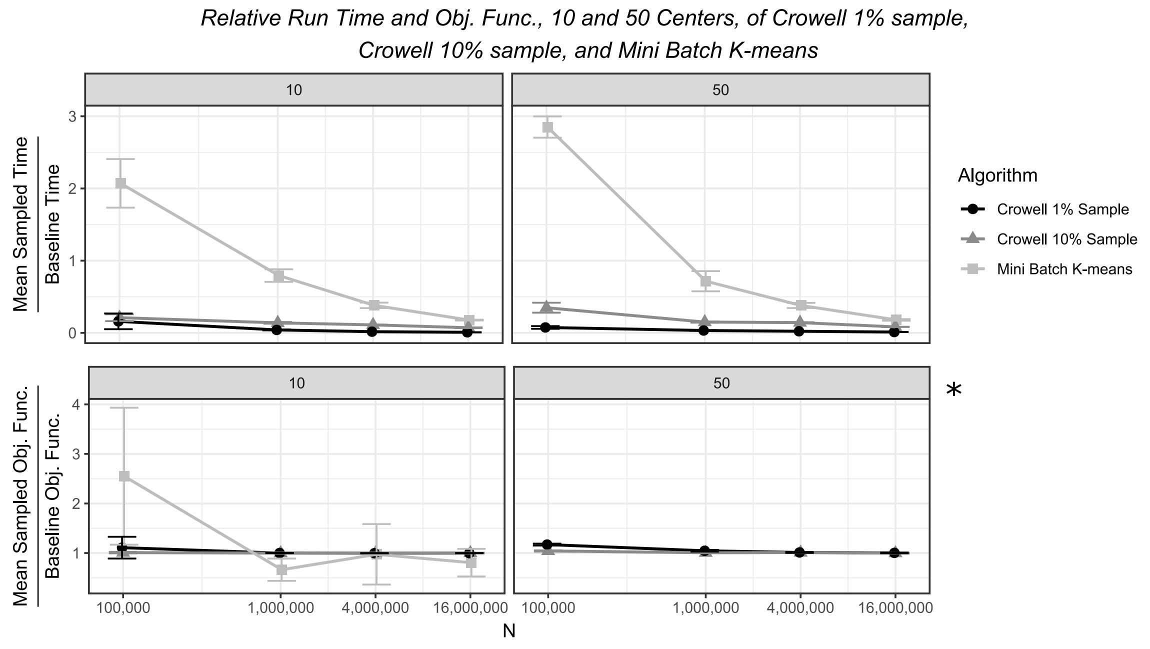

Figure 4.3: Relative run times and relative objective function values of Crowell 1% sample, Crowell 10% sample, and Mini Batch k-means implementations. The run time and objective function values are scaled relative to the run time and objective function values of a baseline sequential run on the full dataset. This means a value above one is worse relative to the baseline and a value below 1 is better. The top row of graphs show the relative run times by each implementation for each dataset, while the bottom row shows the relative objective function. The first column of graphs show the results for 10 cluster experiments and the second column for 50 cluster experiments. The error bars represent the standard deviation of the values scaled relative to the baseline. The x-axis is shown on a log scale.
*Note: The objective function values for Mini Batch k-means on 50 clusters were excluded from the graph because they were extremely erratic and gave relative values many orders of magnitudes larger than shown on the graph.

Chapter 5

A Parallel Implementation of the Filtering Algorithm

The final ingredient of our ultimate k-means implementation is to exploit parallelism to achieve faster running times. Parallelism is trivially applied to Lloyds algorithm by running the distance calculations in parallel. It is less trivially applied to the filtering algorithm due to the potentially unbalanced nature of the k-d tree and unpredictability of how many calculations will need to be run in a subtree to determine all that subtrees nearest candidate centers. We demonstrate how to effectively parallelize the filtering algorithm. Since most of the time of our algorithm is spent on computing the cluster centers once the tree is built, our initial focus was to parallelize the updating of cluster centers while constructing the k-d tree sequentially. This led to the situation where the tree construction accounted for a significant portion of this partially parallelized implementation and we added a parallel implementation of the tree building procedure as well.

5.1 Parallel Computing

Parallel computing allows us to divide chunks of independent work among multiple CPU or GPU cores to be run in parallel with each other, rather than sequentially. These parallel subtasks are called threads. Many applications will have sections that cannot be parallelized and must be run sequentially, and some parallelizable sections may not be perfectly parallelizable. A theoretical speed up based on the percentage of an application that is parallelizable can be approximated by Amdahls Law [2].

Part of the challenge of parallelizing an algorithm is splitting the amount of work while limiting the overhead that comes with communication between threads. In some problems, this is trivial, but for others this can be very challenging. The effort (and runtime overhead) of parallelization may not be worth it for parts of an algorithm that is already very cheap. This is because the decreasing the run time of something that is already cheap does not significantly impact the overall run time of the algorithm.

Furthermore, overheads that appear in parallelizing that part may make that part slower than its sequential version. For this reason, a parallel implementation should be driven by where the bottlenecks of the program are.

5.2 Parallelism in Lloyds Algorithm

Lloyds Algorithm is highly parallelizable due to its independent distance and summary calculations in the assignment and update steps respectively. With P threads, we can have each thread responsible for $\frac{N}{P}$ points in the assignment step. Each thread will do each of the k distance calculations to the candidate centers for their given points and will keep track of the calculated sums and weights associated with each candidate center. Once each thread has terminated, they will combine their results by adding their candidate center sum and weight vectors. In the update step, we can also parallelize the calculations necessary to update the next centroid set, although this may not give significant speedup unless $k \gg P$. Overall, each thread does $\frac{kN}{P}$ work per iteration, which is perfect load balance and can be achieved virtually without communication overhead. Due to the regular data access patterns by each thread (points and centers are scanned in sequence) and the fact that at least the assignment phase of each iteration includes only very localized branches, Lloyds algorithm is also very easy to implement on a GPU, to gain additional computing power. Note, however, that sequential implementations of Mini Batch k-means and the filtering algorithm are orders of magnitude faster than Lloyds algorithm. Thus, without massive parallelism, a parallel implementation of Lloyds algorithm is still not competitive with even sequential implementations of these algorithms. This can be supported by experiments in the literature [6, 21], which show 10-100 times speed-ups on GPUs over base k-means compared to upwards of 500 times shown in Sculley [28] and shown further here.

5.3 Parallelizing Mini Batch K-means

Given the fast performance of sequential Mini Batch k-means and its simple structure, it is worthwhile to ask whether Mini Batch k-means can be parallelized for further speed gains. There are at least two challenges in doing so: Since each iteration works with a very small sample of the input, there is not enough work to be done in each iteration for parallelism to be effective. One strategy to overcome this would be to run multiple batches in parallel, one per thread. However, given the iterative nature of Mini Batch k-means, this is essentially the same as running one large batch (spread across multiple threads). This may allow Mini Batch k-means to converge faster, thereby reducing its running time, but it is unlikely to yield even close to a factor P speed-up. Thus, we did not explore parallelizing Mini Batch k-means further.

5.4 Parallelism Strategies with the Filtering Algorithm

Unlike Lloyds Algorithm, the Filtering Algorithm is much more difficult to evenly partition into chunks for parallel processing. With a k-d tree and filtering, we do not strictly know ahead of time which distance calculations between points and centers need to be performed, and we are working with trees and sub-trees instead of a simple set of points. For this reason, it is not as simple as initially partitioning the calculations into equal groups and distributing them among the threads. Additionally, we cannot even necessarily partition our k-d tree into P sub-trees, as the k-d tree may not be perfectly balanced. To implement parallel computing with the filtering algorithm, we face two options. One option is to partition at the beginning of the algorithm, and risk that we may have extremely unbalanced work groupings, leading to inefficient use of threads. The other option is to dynamically balance the work to the threads as the work is discovered. These options will be discussed in subsections 5.3.2 and 5.3.3 below. We can also parallelize the tree building process of the algorithm, but this is also non-trivial because the final tree may not be perfectly balanced. An imbalanced tree leads to potential for an imbalance in the work each thread does and can cause inefficient and slow parallelism.

5.4.1 Parallel Tree Building

To parallelize the tree building process, we will calculate a small portion of the top of the tree sequentially, and then partition the remaining sub-trees to be calculated concurrently by threads. More specifically, we will sequentially move down the tree $\lfloor \log_2(P) \rfloor + OPF$ levels, where P is the number of threads we will run on, and OPF is some over-partitioning factor, which will leave us with $2^{\lfloor \log_2(P) \rfloor + OPF}$ total sub-trees to be calculated. We know that each sub-tree may not require the same amount of processing power to complete. For this reason, we gather more sub-trees than threads we will run on so that we can over-partition the work to each of our threads. The degree of the over-partitioning is determined by OPF, with each thread receiving 2^{OPF} sub-trees to build when P is a power of 2. By giving a random sample of sub-trees to each thread, it is expected that variations in subtree size will even out across threads with each thread receiving some larger and some smaller subtrees. Each thread runs the regular sequential tree building algorithm on its set of sub-trees and finish building the top of the tree sequentially after the bottom sub-trees have each been built. This runs the majority of the tree construction with efficient parallelism, and with little to no additional overhead over a sequential run. A visualization of this method is shown in Figure 5.1. The sequential construction of the top part of the tree would likely become a bottleneck in environments where P is large, such as GPUs or high-end servers. $P = n^{\varepsilon}$ leads to an ε-fraction of the total tree building work to be in the top portion of the tree, which we build sequentially. This can be mitigated by using a different parallelization strategy for the top of the tree (because there are few nodes but each has a large number of points associated with it), but we chose not to do this because our target is a standard multicore system with a very modest number of cores compared to the millions of points we aim to cluster efficiently.

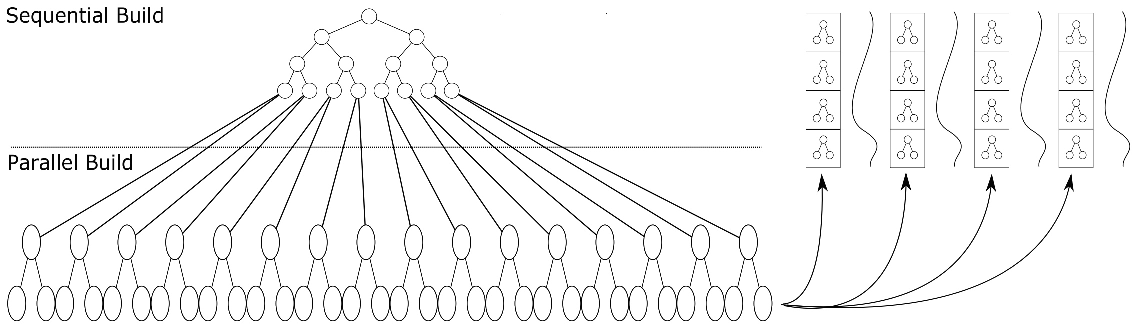

Figure 5.1: An example visualization of parallel tree building. In this example, we are running on 4 threads (P) and are using an over-partitioning factor (OPF) of 2. The tree nodes at level $\lfloor \log_2(4) \rfloor + 2 = 4$ and above of the tree will be built sequentially, and the $2^{\log_2(4)+2} = 2^4 = 16$ subtrees at level 5 will be distributed evenly amongst the threads.

5.4.2 Single Stack Traversal

The first option we tried to implement our dynamic load balancing strategy was to use a single work stack that all threads access collectively to exchange subtrees to traverse. In this method, each thread will pop a subtree to be traversed from the stack whenever it is idle and pushes smaller subtrees onto the stack. This approach utilizes a parallel depth-first search approach. Initially, the master thread pushes the root node on the stack. Any thread working on a node v filters vs candidate centers and, if more than one candidate center remains, schedules vs children w_1 and w_2 to be traversed recursively. It holds on to w_1 to traverse w_1 itself and pushes w_2 onto the stack, to be visited by the next available thread. This ensures that whenever a thread is idle, it has the possibility to get new work if there is new work available and remains relatively balanced even if some threads get more computationally intensive subtrees than others. An example of this is shown in Figure 5.2.

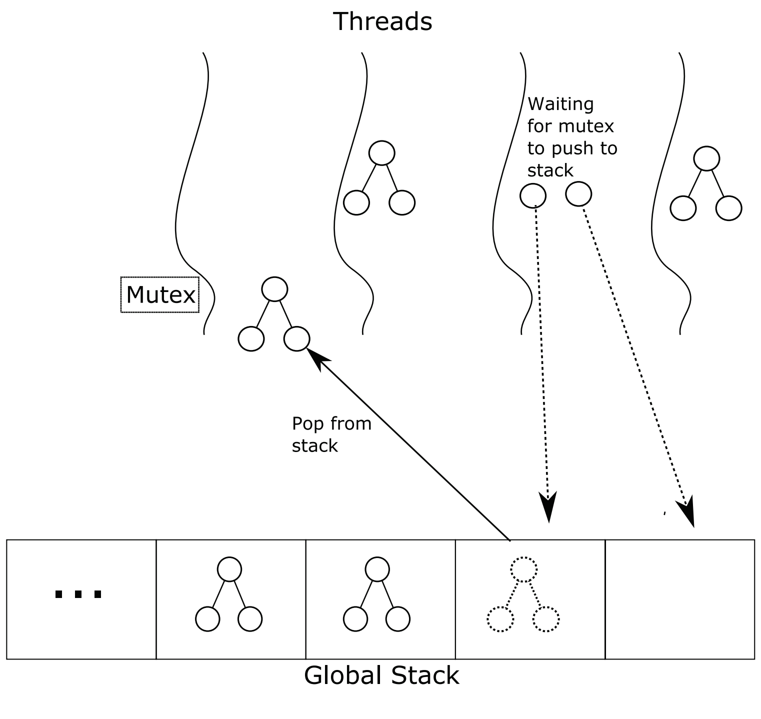

Figure 5.2: The global stack parallel filtering algorithm keeps a global stack for traversal order shared by all threads. Threads pop from the stack when they require more work, and push new work to the stack after they have processed a node. These operations are mutex protected to ensure the integrity of the stack and avoid work duplication leading to incorrect results.

While this method ensures a mostly balanced workload for each thread and is a relatively simple solution, it has a large bottleneck. To ensure that multiple threads do not attempt to take the same node from the stack, as well as to allow the threads to continually check for new work, we must place a mutex lock around operations on the stack. This results in many calls to lock and unlock the mutex, which is a large overhead over the course of the algorithm and can greatly limit parallelism. For this reason, performance in various cases will favour other k-means algorithms, or a work sharing approach to the filtering algorithm.

5.4.3 Work Sharing Traversal

Work sharing is another method of dynamically balancing the threads work (see Algorithm 3.). In this parallel algorithm, each thread maintains its own local work dequeue (double ended queue), rather than all the threads sharing one global stack. Each worker thread begins by racing to gain a mutex lock in order to take the initial work, the root node (Alg. 3, lines 8-12). Each thread that does not get the initial work will push itself onto an idle queue and await further instructions (Alg. 3, line 27; Alg. 4). At each node processing iteration, the working threads will compute the node on their local work dequeue and push remaining work back to their dequeue (Alg.3 line 37). At the end of each node processing iteration, the working thread will first check to see the number of tree nodes on its local dequeue. If it has more than one node, then the thread will check to see if there are any threads on the idle queue. If there are, the working thread takes the work from the bottom of its dequeue and puts it in the dequeue of the idle thread, and then signals the idle thread to wake up and continue working (Alg.3 line 38; Alg. 5). This process is visualized in Figure 5.3. We take from the bottom of the dequeue because the bottom node on the stack will contain the node having the largest sub-tree of all available work on the dequeue. If there are no threads on the idle queue, or the working thread has only one node on its dequeue, then it will continue working as usual. If the working thread has no remaining work, and the idle queue has size $< P - 1$ (meaning there are other threads still working), the thread will add itself to the idle queue. If the idle queue size is $P - 1$, then this thread is the last worker, and all the work will have been completed (Alg. 3 lines 15-18). In this case, rather than going idle, the thread will record its final results (Alg. 3 lines 22-25), and then signal every other thread in the idle queue to record their results, one by one (Alg. 3 lines 19-21, 28-33). Once the centroid results from each thread have been recorded, the threads will exit, and a new iteration can begin.

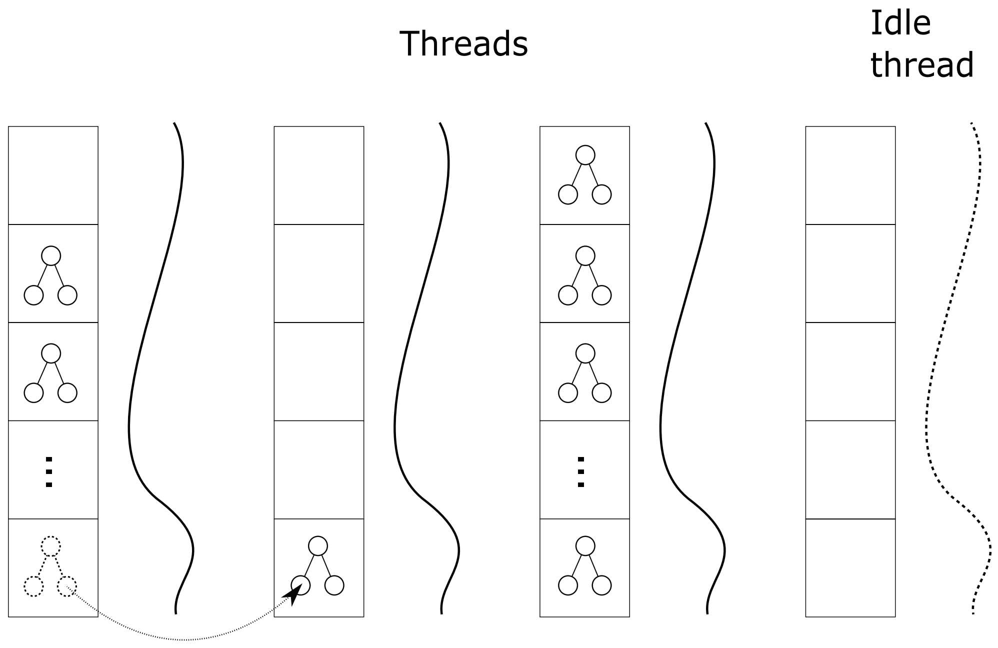

Figure 5.3: An example of the work sharing algorithm. In this figure, the first thread has just finished an iteration and has popped thread two from the idle queue. Thread one places the work from the bottom of its local stack on the local stack of thread two and signals thread two to start working again. Thread three is currently in the middle of an iteration, while thread four is on the idle queue. When thread three finishes its current iteration, it will check for idle threads, discover thread four, and place its work from the bottom of the local stack on thread fours stack.

In this parallel work sharing algorithm, we make use of mutex locks and barriers to ensure that we know for certain when threads are idle or working. We place a mutex around all operations made to the idle queue, so that we ensure that working threads can know for certain if threads are idle, as well as ensure that multiple working threads dont try to send work to the same idle thread. We use a barrier with a conditional variable to allow our idle threads to wait, while still being ready to continue working when signalled. When checking for idle threads, we make a quick initial check without acquiring the mutex, and if there appear to be idle threads, we then acquire the mutex to check for certain. This allows us to remove the overhead of every thread locking and unlocking the mutex on every node processing iteration by replacing it with much

Algorithm 3 Parallel Work Sharing K-means Filter

```
1: [$hp$] $\leftarrow$ set of threads;
2: $gm \leftarrow$ global mutex;
3: $tm \leftarrow$ thread mutex;
4: $idleQ \leftarrow$ idle thread queue;
5: $nBusy \leftarrow$ number of non-idle threads;
6: $workFinished \leftarrow$ FALSE;
7:
8: lock $gm$; {First thread gets the initial work}
9: if No thread has taken the initial work then
10:     Push the initial work to $p.traversalStack$;
11: end if
12: unlock $gm$;
13:
14: while 1 do
15:     if isEmpty($p.traversalStack$) then
16:         lock $gm$;
17:         if $nBusy == 1$ then
18:             $workFinished = TRUE$;
19:             for $p_i$ in $idleQ$ do
20:                 wake $p_i$;
21:             end for
22:             lock $gm$;
23:             $p_i.recordResults()$;
24:             unlock $gm$;
25:             break;
26:         end if
27:         $p_i.waitIdle(tm, idleQ, nBusy)$;
28:         if $workFinished$ then
29:             lock $gm$;
30:             $p_i.recordResults()$;
31:             unlock $gm$;
32:             break;
33:         else
34:             continue;
35:         end if
36:     end if
37:     $visitNode(p.traversalStack.top())$;
38:     $p_i.shareWork(nBusy, idleQ, gm)$;
39: end while
```

Algorithm 4 thread.waitIdle($tm, idleQ, nBusy$)

```
1: $tm \leftarrow$ thread mutex;
2: $idleQ \leftarrow$ idle thread queue;
3: $nBusy \leftarrow$ number of non-idle threads;
4: lock $tm$;
5: $idleQ.push(this)$;
6: $nBusy - -$;
7: wait until another thread signals to wake up;
8: unlock $tm$;
```

Algorithm 5 thread.shareWork($nBusy, idleQ, gm$)

```
1: $gm \leftarrow$ global thread mutex;
2: $idleQ \leftarrow$ idle thread queue;
3: $nBusy \leftarrow$ number of non-idle threads;
4: if $p.traversalStack.size() > 1$ then
5:     if $nBusy < |P|$ then
6:         lock $gm$ {Share any extra work to idle threads} {Check to be sure after mutex is locked}
7:         if $nBusy < |P|$ then
8:             pop extra work from front of this stack;
9:             place extra work on the stack of the first idle thread in idle queue;
10:            wake the thread;
11:        end if
12:        unlock $gm$;
13:    end if
14: end if
```

cheaper checks that are correct most of the time.

5.5 Experimental Results

To evaluate the speed-up gained by these methods, we implemented both single queue and work sharing approaches for testing against our competitor algorithms. We also planned to compare to a GPU implementation, but we had severe difficulty compiling these implementations in our machine environment. As was to be expected, our single-queue implementation did not outperform our work sharing approach in any tests. Therefore, we will exclude it from further experiments. In this experiment, our parallel implementations will be run on 8 cores, as that provided the optimal speed for our implementation.

From Table 5.1, we can see that our parallel implementation performs marginally better than our sequential implementation on larger inputs, as well as with higher values of k. This small speed-up when comparing our sequential and parallel implementations is not the full picture. To compare our implementations to other implementations, we include the data read time as part of the reported run time. When comparing our parallel and sequential implementations, these values should be the same, and they mask the true speed-up of our parallelism. Despite this, when accounting for data read time, we only see approximately 2.5 times speed up in the best scenario (16 million points, 50 centers), and do not see any significant speed up (and sometimes the parallel implementation is in fact slower) with data sets of 1 million points and less. This value is well below the 8 times speed up we would achieve with perfect parallelism and is short of realistic expectations for parallelism. This is likely due to the overheads of frequent communication and data sharing between threads and the inevitability that the work divided among parallel threads will not be perfectly balanced.

We also maintain a fully optimized objective function, unlike our sampled experiments and Mini Batch k-means. Despite this, our run time does not compare to time gained with sampling-based approaches. To improve our approach to the best it can be, we can combine sampling with our parallel k-d tree approach.

Table 5.1: Comparison of k-means method run times in seconds [mean (sd)]

k	**Algorithm**	**Dataset**			
		ASRS100k	Reddit1M	Reddit4M	Reddit16M
10	Crowell (Sequential)	0.294 (<0.01)	2.372 (0.01)	9.777 (0.02)	58.762 (2.35)
	Crowell (Sampling 1%)	0.047 (0.03)	0.097 (0.01)	0.164 (0.03)	0.156 (<0.01)
	Crowell (Work Sharing)	0.421 (0.05)	2.669 (0.10)	10.209 (0.58)	47.460 (1.29)
	Kanungo et al.	0.726 (0.02)	4.377 (0.91)	16.493 (0.20)	53.616 (0.76)
	Mini Batch	0.608 (0.09)	1.883 (0.20)	3.727 (0.36)	10.395 (0.33)
50	Crowell (Sequential)	0.439 (<0.01)	3.345 (0.04)	11.476 (0.02)	63.209 (2.42)
	Crowell (Sampling 1%)	0.033 (<0.01)	0.110 (0.01)	0.258 (0.01)	0.157 (<0.01)
	Crowell (Work Sharing)	0.706 (0.03)	3.092 (0.17)	10.714 (0.12)	48.288 (0.63)
	Kanungo et al.	1.935 (0.09)	14.024 (0.13)	32.258 (0.51)	85.662 (0.91)
	Mini Batch	1.251 (0.06)	2.395 (0.46)	4.355 (0.40)	11.350 (0.74)

Chapter 6

Final Performance Evaluation

Up to this point, each of our attempted methods for decreasing running time have been applied relatively independently of each other. Here we combine them in an attempt to maximize run time performance. As we have already previously combined the filtering algorithm (k-d trees) with parallelism and sampling separately, it is simply a matter of combining the approaches. This should allow for the best performance on low dimensional, large datasets.

6.1 Combining K-D tree, Sampling, and Parallelism

While there are not many challenges in combining the filtering algorithm, sampling, and parallelism that have not already been covered in their respective sections, combining the three methods does not guarantee an increase in run time performance. As observed in previous experiments, parallel k-means does not gain significant time on smaller datasets. By using sampling, we are greatly reducing the size of the dataset, meaning that we may not see the benefits of parallelism with sampling until the data becomes extremely large.

6.2 Experimental Results

We ran the work sharing implementation combined with the best sampling rates from our previous experiments with sampling and compared them against the other best implementations. As we can see from Table 6.1, our combination approach succeeds at achieving run times appropriate for interactive applications on very large datasets. This approach is slightly slower than our 1% sampling approach in most cases, but still remains much faster than other competitive algorithms. Despite this difference in our sampling approach and combination approach, it is possible that this experiment is limited by the size of the data we tested. In chapter 5, we showed that a parallel

Table 6.1: Comparison of k-means method run times in seconds [mean (sd)]

k	**Algorithm**	**Dataset**			
		ASRS100k	Reddit1M	Reddit4M	Reddit16M
10	Crowell (Sequential)	0.294 (<0.01)	2.372 (0.01)	9.777 (0.02)	58.762 (2.35)
	Crowell (Sampling 1%)	0.047 (0.03)	0.097 (0.01)	0.164 (0.03)	0.156 (<0.01)
	Crowell (Work Sharing)	0.421 (0.05)	2.669 (0.10)	10.209 (0.58)	47.460 (1.29)
	Crowell (W. S. + 1%)	0.044 (<0.01)	0.105 (0.04)	0.188 (0.02)	0.535 (0.05)
	Crowell (W. S. + 10%)	0.103 (0.02)	0.339 (0.03)	1.017 (0.04)	4.037 (0.09)
	Kanungo et al.	0.726 (0.02)	4.377 (0.91)	16.493 (0.20)	53.616 (0.76)
	Mini Batch	0.608 (0.09)	1.883 (0.20)	3.727 (0.36)	10.395 (0.33)
50	Crowell (Sequential)	0.439 (<0.01)	3.345 (0.04)	11.476 (0.02)	63.209 (2.42)
	Crowell (Sampling 1%)	0.033 (<0.01)	0.110 (0.01)	0.258 (0.01)	0.157 (<0.01)
	Crowell (Work Sharing)	0.706 (0.03)	3.092 (0.17)	10.714 (0.12)	48.288 (0.63)
	Crowell (W. S. + 1%)	0.069 (<0.01)	0.120 (<0.01)	0.241 (0.01)	0.815 (0.04)
	Crowell (W. S. + 10%)	0.140 (0.02)	0.625 (0.02)	1.441 (0.06)	4.432 (0.09)
	Kanungo et al.	1.935 (0.09)	14.024 (0.13)	32.258 (0.51)	85.662 (0.91)
	Mini Batch	1.251 (0.06)	2.395 (0.46)	4.355 (0.40)	11.350 (0.74)

approach only begins to improve on our sequential approach between our 4 million and 16 million point datasets. In our combination approach, a 1% sample on our largest dataset leaves us with only 160000 points, which is well below the size we might expect the parallelism of our combination method to allow us to overtake our pure sampling method. To verify this, we require datasets much larger than were available, which is beyond the scope of this work.

Chapter 7

Conclusion

In this thesis, we have outlined several approaches to improving k-means, their contributions to improving the speed of k-means, and their challenges. We have also devised a way to combine these approaches and get contributions from all of them in one algorithm. Our implementations have been compared against our goal of achieving interactive run times on datasets of millions of points and also compared to some of the best existing k-means algorithms to see the impact on the research space.

7.1 Conclusion

Using a combined approach of a k-d tree, sampling, and parallelism, we can outperform the fastest k-means competitors with minimal to no loss in objective function. Additionally, we can achieve sub-second run times on datasets up to 16 million points in a low-dimensional space, and potentially maintain speeds appropriate for interactive applications at much larger data sizes. While our parallelism implementation was not as effective as we had hoped, it still provided speed-up on larger data sizes, and has the potential to continue scaling with data size beyond the scope of this work.

7.2 Future Work

Other improvements could still be made to improve the efficiency of our method that are beyond the scope of this paper. Furthermore, this work could also be adapted towards other areas of clustering such as k-means with a skewing metric towards some features, such as that used in Soto et al. [29] or utilizing these techniques with other clustering algorithms. We also had limitations in the size of the data that we were able to obtain, which could be used to verify the efficacy of our combination approach over a pure sampling approach.

7.2.1 GPU implementation

Given the focus on large datasets for this work, it would appropriate to explore this approach using a GPU. To implement this algorithm on a GPU, one would need to first explore if there is any benefit to parallelizing the tree construction on the GPU, or to simply construct it on the CPU. The challenge in efficiently implementing tree construction on the GPU is that the GPU has thousands of cores compared to the CPU, so we will need to initially traverse many levels of the tree in order to gather enough sub-trees for each GPU thread to process. This could leave sub-trees that do not provide enough work to each core to outweigh the costs of gathering those sub-trees. Additionally, we will incur a potential additional overhead penalty by having to transfer data from the CPU to the GPU and back again. The next challenge is in the work sharing filtering algorithm. First, a GPU core has limited local memory, which may put limits on how big of a sub-tree a single thread can process. On a GPU with thousands of cores, it will also take time for the work to propagate out to every thread, leaving many threads idle for much of a k-means iteration. Furthermore, shared memory limits on the GPU may additionally complicate the ability for threads to share their work with each other. While this presents challenges, we may see significant speed-up when using a GPU for extremely large datasets such that each GPU thread gets enough work to surpass losses from overhead.

7.2.2 Skewing Metrics

Implementing a skewing method for k-means, such as the one proposed in Soto et. al. [29], is another potential extension to this algorithm. To do so, one would need to keep a weight for each point and candidate cluster combination, which would propagate throughout the tree to apply to groups of points held in sub-trees. This poses a challenge in memory efficiency, as doing this naively would require $k*(2N-1)$ weights held within the tree. This would also require modifications to take the weights into account when calculating distortion throughout each iteration.

Bibliography

[1] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. An efficient k-means clustering algorithm. In *Proc. First Workshop High Performance Data Mining, March 1998*, 1998.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, volume 30 of *AFIPS Conference Proceedings*, pages 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967.

[3] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. OPTICS: ordering points to identify the clustering structure. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 49–60. ACM Press, 1999.

[4] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 1027–1035. SIAM, 2007.

[5] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *PVLDB*, 5(7):622–633, 2012.

[6] Hongtao Bai, Lili He, Dantong Ouyang, Zhan-Shan Li, and He Li. K-means on commodity gpus with CUDA. In Mark Burgin, Masud H. Chowdhury, Chan H. Ham, Simone A. Ludwig, Weilian Su, and Sumanth Yenduri, editors, *CSIE 2009, 2009 WRI World Congress on Computer Science and Information Engineering, March 31 - April 2, 2009, Los Angeles, California, USA, 7 Volumes*, pages 651–655. IEEE Computer Society, 2009.

[7] Jeremy Bejarano, Koushiki Bose, Tyler Brannan, Anita Thomas, Kofi Adragni, Nagaraj Neerchal, and George Ostrouchov. Sampling within k-means algorithm to cluster large datasets. Technical report, Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, 2011.

[8] Partha S. Bishnu and Vandana Bhattacherjee. Software fault prediction using quad tree-based k-means clustering algorithm. *IEEE Trans. Knowl. Data Eng.*, 24(6):1146–1150, 2012.

[9] Christos Boutsidis, Anastasios Zouzias, Michael W. Mahoney, and Petros Drineas. Randomized dimensionality reduction for k-means clustering. *IEEE Trans. Information Theory*, 61(2):1045–1062, 2015.

[10] Sanjoy Dasgupta. *The Hardness of k-Means Clustering*. Technical report (University of California, San Diego. Department of Computer Science and Engineering). Department of Computer Science and Engineering, University of California, San Diego, 2008.

[11] Ian Davidson and Ashwin Satyanarayana. Speeding up k-means clustering by bootstrap averaging. In *IEEE data mining workshop on clustering large data sets*, 2003.

[12] Marie desJardins, James MacGlashan, and Julia Ferraioli. Interactive visual clustering. In David N. Chin, Michelle X. Zhou, Tessa A. Lau, and Angel R. Puerta, editors, *Proceedings of the 12th International Conference on Intelligent User Interfaces, IUI 2007, Honolulu, Hawaii, USA, January 28-31, 2007*, pages 361–364. ACM, 2007.

[13] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 579–587. JMLR.org, 2015.

[14] Charles Elkan. Using the triangle inequality to accelerate k-means. In Tom Fawcett and Nina Mishra, editors, *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*, pages 147–153. AAAI Press, 2003.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231. AAAI Press, 1996.

[16] Giuseppe Di Fatta and David Pettinger. Dynamic load balancing in parallel kd-tree k-means. In *10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, June 29-July 1, 2010*, pages 2478–2485. IEEE Computer Society, 2010.

[17] Greg Hamerly. Making k-means even faster. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2010, April 29 - May 1, 2010, Columbus, Ohio, USA*, pages 130–140. SIAM, 2010.

[18] Erez Hartuv and Ron Shamir. A clustering algorithm based on graph connectivity. *Inf. Process. Lett.*, 76(4-6):175–181, 2000.

[19] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, 2002.

[20] Leonard Kaufman and Peter J. Rousseeuw. *Clustering By Means of Medoids.* Delft University of Technology : Reports of the Faculty of Technical Mathematics and Informatics. Faculty of Mathematics and Informatics, 1987.

[21] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. Speeding up k-means algorithm by gpus. *J. Comput. Syst. Sci.*, 79(2):216–229, 2013.

[22] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Trans. Information Theory*, 28(2):129–136, 1982.

[23] James MacQueen. Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[24] Songrit Maneewongvatana and David M. Mount. Its okay to be skinny, if your friends are fat. In *Center for Geometric Computing 4th Annual Workshop on Computational Geometry*, volume 2, pages 1–8, 1999.

[25] P.C. Meenakshi, S. Meenu, M. Mithra, and P. Leela Rani. Fault prediction using quad tree and expectation maximization algorithm. *International Journal of Applied Information Systems*, 2(4):36–40, 2012.

[26] Fabian Garcia Nocetti, Julio Solano-González, and Ivan Stojmenovic. Connectivity based k-hop clustering in wireless networks. *Telecommunication Systems*, 22(1-4):205–220, 2003.

[27] Dan Pelleg and Andrew W. Moore. Accelerating exact k-means algorithms with geometric reasoning. In Usama M. Fayyad, Surajit Chaudhuri, and David Madigan, editors, *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 15-18, 1999*, pages 277–281. ACM, 1999.

[28] D. Sculley. Web-scale k-means clustering. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 1177–1178. ACM, 2010.

[29] Axel J. Soto, Ryan Kiros, Vlado Kešelj, and Evangelos E. Milios. Exploratory visual analysis and interactive pattern extraction from semi-structured data. *TiiS*, 5(3):16:1–16:36, 2015.

[30] Jiadong Wu and Bo Hong. An efficient k-means algorithm on CUDA. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*, pages 1740–1749. IEEE, 2011.

[31] Xiaowei Xu, Martin Ester, Hans-Peter Kriegel, and Jörg Sander. A distribution-based clustering algorithm for mining in large spatial databases. In Susan Darling Urban and Elisa Bertino, editors, *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, pages 324–331. IEEE Computer Society, 1998.

[32] Krishna Yadav and Jwalant Baria. Mini-batch k-means clustering using map-reduce in hadoop. *International Journal of Computer Science and Information Technology Research*, 2(2):336–342, 2014.

[33] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing, First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings*, volume 5931 of *Lecture Notes in Computer Science*, pages 674–679. Springer, 2009.

[34] Alexander Zien and Joaquin Quiñonero Candela. Large margin non-linear embedding. In Luc De Raedt and Stefan Wrobel, editors, *Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, August 7-11, 2005*, volume 119 of *ACM International Conference Proceeding Series*, pages 1060–1067. ACM, 2005.