

Fast Clustering using MapReduce

Alina Ene*
University of Illinois
Urbana, IL
ene1@illinois.edu

Sungjin Im†
University of Illinois
Urbana, IL
im3@illinois.edu

Benjamin Moseley‡
University of Illinois
Urbana, IL
bmosele2@illinois.edu

ABSTRACT

Clustering problems have numerous applications and are becoming more challenging with the growing size of data available. In this paper, we consider designing clustering algorithms that can be used in MapReduce, the most popular programming environment for processing large datasets. We focus on the practical and popular clustering problems k -center and k -median. We develop fast clustering algorithms with constant factor approximation guarantees. From a theoretical perspective, we give the first analysis showing several clustering algorithms are in \mathcal{MR}^O , a theoretical MapReduce class introduced by Karloff et al. [26]. Our algorithms use sampling to decrease the data size and run a time consuming clustering algorithm such as local search or Lloyd’s algorithm on the reduced data set. Our algorithms have sufficient flexibility to be used in practice since they run in a constant number of MapReduce rounds. We complement these results by performing experiments using our algorithms. We compare the empirical performance of our algorithms to several sequential and parallel algorithms for the k -median problem. The experiments show that our algorithms’ solutions are similar or better than the other algorithms, while running faster than any other parallel algorithm that was tested for sufficiently large data sets.

1. INTRODUCTION

Clustering data is a fundamental problem in a variety of areas of computer science and related fields. Machine learning, data mining, pattern recognition, networking, and bioinformatics use clustering for data analysis. Consequently, there is a vast amount of research focused on the topic [3, 29, 13, 12, 4, 31, 9, 17]. Succinctly, clustering is a partition of data observations into subsets, called clusters, such that the data points assigned to the same cluster are similar according to some metric. Clustering makes data easier to process, mine for information and more human readable.

*. Partially supported by NSF grants CCF-0728782 and CCF-1016684.

†Partially supported by NSF grants CCF-0728782, CCF-1016684, and a Samsung Fellowship.

‡Partially supported by NSF grants CCF-0728782 and CCF-1016684.

In several applications, it is of interest to classify or group web pages according to their content or cluster users based on their online behavior. One such example is finding communities in social networks. Communities consist of individuals that are closely related according to some relationship criteria. Finding these communities is of interest for applications such as predicting buying behavior or designing targeted marketing plans and is an ideal application for clustering. However, the size of the web graph or social network graphs can be quite large; for instance, the web graph consists of a trillion edges [30]. When the amount of data is this large, it is difficult or even impossible for the data to be stored on a single machine, which renders sequential algorithms unusable. In situations where the amount of data is prohibitively large, the MapReduce [16] programming paradigm is used to overcome this obstacle. MapReduce and its open source counterpart Hadoop [32] are a distributed computing framework designed to process massive data sets.

The MapReduce model is quite novel; it interleaves sequential and parallel computation. Succinctly, MapReduce consists of several *rounds* of computation. There is a set of machines, each of which has a certain amount of memory available; the memory on each machine is limited, and there is no communication between the machines. In each round, the data is distributed among the machines. The data assigned to a single machine is constrained to be sub-linear in the input size. This restriction is motivated by the fact that the input size is very large [26, 14]. After the data is distributed, each of the machines performs some computation on the data that is available to them. The output of these computations are either the final result, or it becomes the input of another MapReduce round. A more precise overview of the MapReduce model is given in Section 1.1.

Problems: In this paper, we are concerned with designing clustering algorithms that can be implemented using MapReduce. In particular, we focus on two well-studied problems: metric k -median and k -center. In both of these problems, we are given a set V of n points, together with the distances between any pair of points; we give a precise description of the input representation below. The goal is to choose a k of the points. Each of the k chosen points represents a cluster and is referred to as a *center*. Every data point is assigned to the closest center and all of the points assigned to a given point form a cluster. In the k -center problem, the goal is to choose the centers such that the maximum distance between a center and a point assigned to it is minimized. In the k -median problem the objective is to minimize the sum of the distances from the centers to each of the points assigned to the centers. Both of the problems are known to be NP-Hard. Thus previous work has focused on finding approximation algorithms [24, 5, 11, 10, 4, 22, 8]. Previous algorithms are inherently sequential and difficult to adapt to a parallel computing setting with the exception of [8, 21],

which we discussed later.

Input Representation: Let $d : V \times V \rightarrow \mathbb{R}_+$ denote the distance function. The distance function d is a metric, i.e., it has the following properties: (1) $d(x, y) = 0$ if and only if $x = y$, (2) $d(x, y) = d(y, x)$ for all x, y , and $d(x, z) \leq d(x, y) + d(y, z)$ for all x, y, z . The third property is called the triangle inequality; we note that our algorithms only rely on the fact that the distances between points satisfy the triangle inequality.

Now we discuss how the distance function is given to our algorithms. In some settings, the distance function has an implicit compact representation; for example, if the distances between points are shortest path distances in a sparse graph, the graph represents the distance function compactly. However, currently there does not exist a MapReduce algorithm that computes shortest paths in a constant number of rounds, even if the graph is unweighted. This motivates the assumption that we are given the distance function *explicitly* as a set of $\Theta(n^2)$ distances, one for each pair of points, or we are given access to an *oracle* that, given two points, it returns the distance between them. Throughout this paper, we assume that the distance function is given explicitly. More precisely, we assume that the input is a *weighted complete graph* $G = (V, E)$ that has an edge xy between any two points in V , and the weight of the edge xy is $d(x, y)$ ¹. Moreover, we assume that $k = O(n^{1-\delta})$ for some constant $\delta > 0$, and the distance between any pair of points is upper bounded by some polynomial in n . These assumptions are justified in part by the fact that the number of points is very large, and by the memory constraints of the MapReduce model; we discuss the MapReduce model in more detail in Section 1.1.

Contributions: We introduce the *first* approximate metric k -median and k -center algorithms designed to run on MapReduce. More precisely, we show the following results.

THEOREM 1.1. *There is a randomized constant approximation to the k -center problem that, with high probability, it runs in a constant number of MapReduce rounds and uses memory at most $O(k^2 n^\delta)$ on each of the machines for any constant $\delta > 0$.*

THEOREM 1.2. *There is a randomized constant approximation to the k -median problem that, with high probability, it runs in a constant number of MapReduce rounds and uses memory at most $O(k^2 n^\delta)$ on each of the machines for any constant $\delta > 0$.*

To complement these results, we run our algorithms on randomly generated data sets. For the k -median problem we compare our algorithm to a parallelized implementation of Lloyd’s algorithm [28, 7, 1], arguably the most popular clustering algorithm used in practice (see [2, 23] for example), the local search algorithm [4, 22], the best known approximation algorithm for the k -median problem and a partitioning based algorithm that can parallelize any sequential clustering algorithm (see Appendix A). Our algorithms achieve a speed-up of 1000x over the local search algorithm and 20x over the parallelized Lloyd’s algorithm, a significant improvement in running time. Further, our algorithm’s objective is similar to Lloyd’s algorithm and the local search algorithm. For the partitioning based algorithm, we show that our algorithm achieves faster running time when the number of points is large. Thus for the k -median problem our algorithms are fast with a small loss in performance. For the k -center problem we compare our algorithm to the well known algorithm of [17, 20], which is the best approximation algorithm for the problem and is quite efficient. Unfortunately, for the k -center problem our algorithm’s objective is a factor four worse in some

¹We note that some of the techniques in this paper extend to the setting in which the distance function is given as an oracle.

cases. This is due to the sensitivity of the k -center objective to sampling.

Our algorithms show that the k -center and k -median problem belong to the theoretical MapReduce class \mathcal{MRC}^0 that was introduced by Karloff et al. [26]². Let N denote the total size of the input, and let ϵ be a fixed constant greater than zero. A problem is in the MapReduce class \mathcal{MRC}^0 if it can be solved using a constant number of rounds and an $O(N^{1-\epsilon})$ number of machines, where each machine has $O(N^{1-\epsilon})$ memory available [26]. Differently said, the problem has an algorithm that uses a sub-linear amount of memory on each machine and a sub-linear number of machines. One of the main motivations for these restrictions is that a typical MapReduce input is very large and it might not be possible to store the entire input on a single machine. Moreover, the size of the input might be much larger than the number of machines available. We discuss the theoretical MapReduce model in Section 1.1.

Adapting Existing Algorithms to MapReduce: Previous work on designing algorithms for MapReduce are generally based on the following approach. First partition the input across the machines. Each machine performs some computation that eliminates a large fraction of the input, thereby sparsifying the data. The results of this computations are collected on a single machine, which can store the data since the data has been sparsified. This machine performs some computation and the final solution is output. We can use a similar approach for the k -center and k -median problems. We first partition the points across the machines. We cluster each of the partitions. We select one point from each cluster, and put all of the selected points on a single machine. We cluster these points and output the solution. Indeed, a similar algorithm to this was considered by Guha et al. [21] for the k -median problem in the streaming model. We give the details of how to implement this algorithm in MapReduce in Appendix A along with an analysis of the algorithm’s approximation guarantees. Unfortunately, the total running time for the algorithm can be quite large, since it runs a costly clustering algorithm on $\Omega(k\sqrt{n})$ points. Further, this algorithm requires $\Omega(k^2 n)$ memory on each of the machines.

Another strategy for developing algorithms for k -center and k -median that run in MapReduce is to try to adapt existing parallel algorithms. To the best of our knowledge, the only parallel algorithms known with provable guarantees were given by Blelloch and Tangwongsan [8]; Blelloch and Tangwongsan [8] give the first PRAM algorithms for k -center and k -median. Unfortunately, these algorithms assume that the number of machines available is $\Omega(N^2)$, where N is the total input size, and there is some memory available in the system that can be accessed by all of the machines. These assumptions are too strong for the algorithms to be in used in MapReduce. Indeed, the requirements that the machines have a limited amount of memory and that there is no communication between the machines is what differentiates the MapReduce model from standard parallel computing models. Another approach is to try to adapt algorithms that were designed for the streaming model. Guha et al. [21] have given a k -median algorithm for the streaming model; with some work, we can adapt one of the algorithms in [21] to the MapReduce model. However, this algorithm’s approximation ratio degrades exponentially in the number of rounds, a significant loss in performance.

Related Work: There has been a large amount of work on the metric k -median and k -center problems. Due to space constraints, we focus only on closely related work that we have not already mentioned. Both problems are known to be NP-Hard. The first approximation algorithm that was introduced for the k -median problem

²Recall that we only consider instances of the problems in which k is sub-linear in the number of points, and the distances between points are upper bounded by some polynomial in n .

was a $O(\log n \log \log n)$ by Bartal [6, 5]. Later Charikar et al. gave the first constant factor approximation of $6 + \frac{2}{3}$ [11]. This approach was based on LP rounding techniques. The best known approximation algorithm achieves a $3 + \frac{2}{3}$ approximation in $O(n^\delta)$ time [4, 22]; this algorithm is based on the local search technique. On the other hand, Jain et al. [?] have shown that there does not exist an $1 + (2/e)$ approximation for the k -median problem unless $\text{NP} \subseteq \text{DTIME}(n^{O(\log \log n)})$. For the k -center problem, two simple algorithms are known which achieve a 2-approximation [24, 17, 20] and this approximation ratio is tight assuming that $\text{P} \neq \text{NP}$.

MapReduce has received a significant amount of attention recently. Most previous work has been on designing practical heuristics to solve large scale problems [25, 27]. Recent papers [26, 19] have focused on developing computational models that abstract the power and limitations of MapReduce. Finally, there has been work on developing algorithms and approximation algorithms that fit into the MapReduce model [26, 14]. This line of work has shown that problems such as minimum spanning tree, maximum coverage, and connectivity can be solved efficiently using MapReduce.

1.1 MapReduce Overview

In this section we give a high-level overview of the MapReduce model; for a more detailed description, see [26]. The data is represented as $\langle \text{key}; \text{value} \rangle$ pairs. The *key* acts as an address of the machine to which the *value* needs to be sent to. A MapReduce round consists of three stages: map, shuffle, and reduce. The map phase processes the data as follows. The algorithm designer specifies a map function μ , which we refer to as a *mapper*. The mapper takes as input a *single* $\langle \text{key}; \text{value} \rangle$ pair, and it outputs a sequence of $\langle \text{key}; \text{value} \rangle$ pairs. Intuitively, the mapper maps the data stored in the $\langle \text{key}; \text{value} \rangle$ pair to a machine. In the map phase, the map function is applied to all $\langle \text{key}; \text{value} \rangle$ pairs. In the shuffle phase, all $\langle \text{key}; \text{value} \rangle$ pairs with a given key are sent to the same machine; this is done automatically by the underlying system. The reduce phase processes the $\langle \text{key}; \text{value} \rangle$ pairs created in the map phase as follows. The algorithm designer specifies a reduce function ρ , which we refer to as a *reducer*. The reducer takes as input all $\langle \text{key}; \text{value} \rangle$ pairs that have the same key, and it outputs a sequence of $\langle \text{key}; \text{value} \rangle$ pairs; these pairs are either the final output, or they become the input of the next MapReduce round. Intuitively, the reducer performs some sequential computation on all the data that is stored on a machine. The mappers and reducers are constrained to run in time that is polynomial in the size of the *entire input*, and not just their input.

The theoretical *MRC* class was introduced in [26]. The class is designed to capture the practical restrictions of MapReduce as faithfully as possible; a detailed justification of the model can be found in [26]. In addition to the constraints on the mappers and reducers, there are three types of restrictions in *MRC*: constraints on the number of machines used, on the memory available on each of the machines, and on the number of rounds of computation. If the input to a problem is of size N then an algorithm is in *MRC* if it uses at most $N^{1-\epsilon}$ machines, each with at most $N^{1-\epsilon}$ memory for some constant $\epsilon > 0^3$. Notice that this implies that the total memory available is $O(N^{2-2\epsilon})$. Thus the difficulty of designing algorithms for the MapReduce model does not come from the lack of total memory. Rather, it stems from the fact that the memory available on each machine is limited; in particular, the entire input does not fit on a single machine. Not allowing the entire input to be placed on a single machine makes designing algorithms difficult, since a machine is only aware of a subset of the input. Indeed, because of this restriction, it is currently not known whether fundamental graph problems such as shortest paths or maximum match-

ings can be computed in a constant number of rounds, even if the graphs are unweighted.

In the following, we state the precise restrictions on the resources available to an algorithm for a problem in the class MRC^0 .

- **Memory:** The total memory used on a specific machine is at most $O(N^{1-\epsilon})$.
- **Machines:** The total number of machines used is $O(N^{1-\epsilon})$.
- **Rounds:** The number of rounds is constant.

2. ALGORITHMS

In this section we describe our clustering algorithms `MapReduce-kCenter` and `MapReduce-kMedian`. For both of our algorithms, we will parameterize the amount of memory needed on a machine. The amount of memory our algorithms require is parameterized, but we assume that the memory is $\Omega(n^\delta)$ for some constant $\delta > 0$ and that the number of machines is large enough to store all of the input data across the machines. Both algorithms use `Iterative-Sample` as a subroutine which uses sampling ideas from [31]. The role of `Iterative-Sample` is to get a substantially smaller subset of points that well represent all of the points via sampling. To achieve this, `Iterative-Sample` performs the following computation in parallel across the machines: in each round, it adds a small sample of points to the final sample, it determines which points are “well represented” by the sample, and it recursively considers only the points that are not well represented. More precisely, after sampling, `Iterative-Sample` discards most points that are close to the current sample, and it recurses on the remaining (unsampled) points. The algorithm repeats this procedure until the number of points that are still unrepresented is small, and all such points are added to the sample. Once we have a good sample, we put the sampled points on a single machine and we run a clustering algorithm on just the sampled points. Here the clustering algorithm used will depend on the problem considered.

In the following, we give a more precise description of the `Iterative-Sample` algorithm. We note that the algorithm can be easily implemented in MapReduce, and we instead focus on highlighting the algorithmic ideas in the pseudocode description of our algorithms. When we mention the distance of a node x to a set S , we mean the minimum distance between x and any node in S . Our algorithm is parameterized by a constant $0 < \epsilon < \frac{\delta}{2}$ whose value can be changed depending on the system specifications. Simply, the value of ϵ determines the sample size. For each of our algorithms there is a perhaps natural trade-off between the sample size and the running time of the algorithm.

In the following, if the probability of an event is $1 - O(1/n)$, we say that the event occurs with high probability, which we abbreviate as w.h.p.

The following propositions give the theoretical guarantees of the algorithm; these propositions can also serve as a guide for choosing an appropriate value for the parameter ϵ . The first two propositions follow from the fact that, w.h.p., each iteration of `Iterative-Sample` decreases the number of remaining points — i.e., the size of the set R — by a factor of $\Theta(n^\epsilon)$. We give the proofs of these propositions in the next section. Note that the propositions imply that our algorithm belongs to MRC^0 .

PROPOSITION 2.1. *The number of iterations of the while loop of `Iterative-Sample` is at most $O(\frac{1}{\epsilon})$ w.h.p.*

PROPOSITION 2.2. *The set returned by `Iterative-Sample` has size $O(\frac{1}{\epsilon} kn^\epsilon \log n)$ w.h.p.*

³The algorithm designer can choose ϵ .

Algorithm 1 *Iterative-Sample*(V, E, k):

- 1: Set $S \leftarrow \emptyset, R \leftarrow V$.
 - 2: **while** $|R| > \frac{4}{\epsilon} kn^\epsilon \log n$ **do**
 - 3: Add each point in R with probability $\frac{9kn^\epsilon}{|R|} \log n$ independently to S .
 - 4: Add each point in R with probability $\frac{4n^\epsilon}{|R|} \log n$ independently to H .
 - 5: Assign H and S to the same machine along with the edges between any pair of points (u, v) such that u is in H and v is in S .
 - 6: Among all the points in H , find the point v that is the $8 \log n$ th farthest point from S .
 - 7: Partition the points in R uniformly into η subsets, where $\eta \geq 1$ will be defined later.
 - 8: Let $R^i \subseteq R$ denote the set of points assigned on machine i .
 - 9: Assign S and R^i to machine i along with the edges between S and R^i .
 - 10: On each machine i , remove from R^i all of the vertices whose distance to S is smaller than the distance from v to S .
 - 11: Let $R := \bigcup_{i \in [\eta]} R^i$.
 - 12: **end while**
 - 13: Output $C := S \cup R$
-

PROPOSITION 2.3. *We can implement *Iterative-Sample* in MapReduce using $O(\frac{1}{\epsilon})$ rounds with machines of memory $O(n^\delta)$ for a constant $\delta > 2\epsilon$ w.h.p.*

PROOF. Consider a single iteration of the while loop. In the first MapReduce round, the mapper samples the points in R to create the sets H and S , and maps these sets to the same machine. The reducer computes the point v on this machine. In the next round, the mapper partitions R across the machines and assigns S to each machine. The reducer removes all of the vertices in R that are closer than v to S . Thus, a single iteration of the while loop requires at most a constant number of MapReduce rounds. Knowing that this loop is iterated at most $O(\frac{1}{\epsilon})$ by Proposition 2.1 the number of rounds follows.

The memory needed on a machine is dominated by Step (8). The size of S is $O(\frac{1}{\epsilon} kn^\epsilon \log n)$ by Proposition 2.2. Further, the size of R^i is n/η . Thus the total memory needed on a machine is $O(\frac{1}{\epsilon} kn^\epsilon \log n \cdot n/\eta)$, the memory needed to store the edges between R^i and S . By choosing η to be $n^{1-\epsilon}$, we have that the total memory is bounded by $O(\frac{1}{\epsilon} kn^{2\epsilon} \log n)$. Further, by assuming that k is sufficiently smaller than n we can set δ to be a constant slightly larger than 2ϵ and the proposition follows. \square

Once we have this sampling algorithm, our algorithm MapReduce-kCenter for the k -center problem is fairly straightforward. This is the algorithm considered in Theorem 1.1.

Algorithm 2 *MapReduce-kCenter*(V, E, k):

- 1: Let $C \leftarrow \text{Iterative-Sample}(V, E, k)$.
 - 2: Run a k -center clustering algorithm \mathcal{A} on a single machine with $\langle C, k \rangle$ as input.
 - 3: Return the set constructed by \mathcal{A} .
-

However, for the k -median problem, the sample must contain more information than just the set of sampled points. This is because the k -median objective considers the sum of the point distances. To ensure that we can map a good solution on just the points in the sample to a good solution on all of the points, for each unsampled point x , we select the sampled point that is closest to x (if there are several points that are closest to x , we pick

one arbitrarily). Additionally, we assign a weight to each sampled point y that is equal to the number of unsampled points that picked y as its closest point. This is done so that, when we cluster the sampled points on a single machine, we can take into account the effect of the unsampled points on the objective. For a point x and a set of points A , let $d(x, A)$ denote the minimum distance from the point $x \in V$ to a point in A , i.e., $d(x, A) = \min_{y \in A} d(x, y)$. The algorithm MapReduce-kMedian is the following.

Algorithm 3 *MapReduce-kMedian*(V, E, k):

- 1: Let $C \leftarrow \text{Iterative-Sample}(V, E, k)$
 - 2: Send C to all machines.
 - 3: Partition the vertices in V evenly across all m machines.
 - 4: Let $V^i \subseteq V$ denote the set of nodes assigned on machine i .
 - 5: On each machine i , for each $y \in C$, compute $w^i(y) = |\{x \in V^i \setminus C \mid d(x, y) = d(x, C)\}|$.
 - 6: One machine gathers all the weights $w^i(\cdot)$ and computes $w(y) = \sum_{i \in [m]} w^i(y) + 1$.
 - 7: Run a weighted k -median clustering algorithm \mathcal{A} on that machine with $\langle C, w, k \rangle$ as input.
 - 8: Return the set constructed by \mathcal{A} .
-

In line 6, when there is more than one point in C that have the same distance from y , we break ties arbitrarily. The MapReduce-kMedian algorithm performs additional rounds to give a weight to each point in the sample C . We remark that these additional rounds can be easily removed by gradually performing this operation in each iteration of *Iterative-Sample*. The proof of all propositions and theorems will be given in the next section. The algorithm MapReduce-kMedian is the algorithm considered in Theorem 1.2. Notice that both MapReduce-kMedian and MapReduce-kCenter use some clustering algorithm as a subroutine. The running time of these clustering algorithms depend on the size of the sample. The value of ϵ should be chosen so that the sample size is as small as possible given n and k , so that the running time of these Given the previous description of the memory requirements and implementation of *Iterative-Sample*, for both of our algorithms it is straightforward to show how that they can be implemented in MapReduce, and that the memory on a machine is dominated by the memory used by *Iterative-Sample*.

3. ANALYSIS

3.1 Subroutine: *Iterative-Sample*

This section is devoted to the analysis of *Iterative-Sample*, the main subroutine of our clustering algorithms. Before we give the analysis, we introduce some notation. Let S^* denote any set. We will show several lemmas and theorems that hold for any set S^* , and in the final step, we will set S^* to be the optimal set of centers. The reader may read the lemmas and theorems assuming that S^* is the optimal set of centers. We assign each point $x \in V$ to its closest point in S^* , breaking ties arbitrarily but consistently. Let x^{S^*} be the point in S^* to which x is assigned; if x is in S^* , we assign x to itself. Let $S^*(x)$ be the set of all points assigned to $x \in S^*$.

We say that a point x is *satisfied* by S regarding S^* if $d(S, x^{S^*}) \leq d(x, x^{S^*})$. If S and S^* are clear from the context, we will simply say that x is satisfied. We say that x is *unsatisfied* if it is not satisfied. Throughout the analysis, for any point x in V and any subset $S \subseteq V$, we will let x^S denote the point in S that is closest to x .

We now explain the intuition behind the definition of satisfied. Our sampling subroutine's output C may not include each center in S^* . However, a point x could be "satisfied", even though $x^{S^*} \notin C$, by including a point in C that is closer to x than x^{S^*} . Intuitively, if all points are satisfied, our sampling algorithm returned a very representative sample of all points, and our clustering algorithms will perform well. However, we cannot guarantee that all points are satisfied. Instead, we will show that the number of unsatisfied points is small and, furthermore, their contribution to the clustering cost is negligible compared to the satisfied points' contribution. This will allow us to upper bound the distance between the unsatisfied points and the final solution constructed by our algorithm by the cost of the optimal solution.

Since the sets described in `Iterative-Sample` change in each iteration, for the purpose of the analysis, we let R_ℓ , S_ℓ , and H_ℓ denote the sets R , S , and H at the beginning of iteration ℓ . Note that $R_1 = V$ and $S_1 = \emptyset$. Let D_ℓ denote the set of points that are removed (deleted) during iteration ℓ . Note that $R_{\ell+1} = R_\ell - D_\ell$. Let U_ℓ denote the set of points in R_ℓ that are not satisfied by $S_{\ell+1}$ regarding S^* . Let C denote the set of points that `Iterative-Sample` returns. Let U denote the set of all unsatisfied points by C regarding S^* . If one point is satisfied by S_ℓ regarding S^* then it is also satisfied by C regarding S^* , and therefore $U \subseteq \bigcup_{\ell \geq 1} U_\ell$.

We start by upper bounding $|U_\ell|$, the number of unsatisfied points at the end of iteration ℓ .

LEMMA 3.1. *Let S^* be any set with no more than k points. Consider iteration ℓ of `Iterative-Sample`, where $\ell \geq 1$. Then $\Pr \left[|U_\ell| \geq \frac{|R_{\ell+1}|}{3n^\epsilon} \right] \leq \frac{1}{n^2}$.*

PROOF. Consider any point $x \in R_\ell$. Let $y = x^{S^*}$ be the point in S^* to which x is assigned. Recall that $S^*(y)$ denotes the set of all points that are assigned to y . Let ℓ be the number of points in $S^*(y) \cap R_\ell$ that are within distance $d(x, S^*) = d(x, y)$ from y . Note that if any of these ℓ points is added to S_ℓ in the current iteration, the point x becomes satisfied. Therefore we have

$$\Pr \left[|U_\ell \cap S^*(y) \cap R_\ell| \geq \frac{|R_\ell|}{3kn^\epsilon} \right] \leq \left(1 - \frac{9kn^\epsilon}{|R_\ell|} \log n\right)^{\frac{|R_\ell|}{3kn^\epsilon}} \leq \frac{1}{n^3}$$

The lemma now follows by taking the union bound over all points in S^* (recall that $|S^*| \leq k \leq n$). \square

Recall that we selected a threshold point v to discard the points that are well represented by the current sample S . Let $\text{rank}_{R_\ell}(v)$ denote the number of points x in R_ℓ such that the distance from x to S is greater than the distance from v to S . The proof of the following lemma follows easily from the Chernoff inequality.

LEMMA 3.2. *Let S^* be any set with no more than k points. Consider any ℓ -th iteration of the while loop of `Iterative-Sample`. Let v_ℓ denote the threshold in the current iteration, i.e. the $(8 \log n)$ -th farthest point in H_ℓ from $S_{\ell+1}$. Then we have $\Pr \left[\frac{|R_\ell|}{n^\epsilon} \leq \text{rank}(v_\ell) \leq \frac{4|R_\ell|}{n^\epsilon} \right] \geq 1 - \frac{2}{n^2}$.*

PROOF. Let $r = \frac{|R_\ell|}{n^\epsilon}$. Let $N_{\leq r}$ denote the number of points in H_ℓ that have ranks smaller than r , i.e. $N_{\leq r} = |\{x \in H_\ell \mid \text{rank}_{R_\ell}(x) \leq r\}|$. Likewise, $N_{\leq 4r} = |\{x \in H_\ell \mid \text{rank}_{R_\ell}(x) \leq 4r\}|$. Note that $\mathbf{E}[N_{\leq r}] = 4 \log n$ and $\mathbf{E}[N_{\leq 4r}] = 16 \log n$. By Chernoff inequality, we have $\Pr[N_{\leq r} \geq 8 \log n] \leq \frac{1}{n^2}$ and $\Pr[N_{\leq 4r} \leq 8 \log n] \leq \frac{1}{n^2}$. Hence the lemma follows. \square

COROLLARY 3.3. *Consider iteration ℓ of `Iterative-Sample`. Then $\Pr \left[\frac{|R_\ell|}{n^\epsilon} \leq |R_{\ell+1}| \leq \frac{4|R_\ell|}{n^\epsilon} \right] \geq 1 - \frac{2}{n^2}$.*

The above corollary immediately implies Proposition 2.1 and 2.2. Now we show how to map each unsatisfied point to a satisfied point such that no two unsatisfied points are mapped to the same satisfied point; that is, the map is injective. Such a mapping will allow us to bound the cost of unsatisfied points by the cost optimal solution. The following theorem is the core of our analysis. The theorem defines a mapping $p : U \rightarrow V$; for each point x , we refer to $p(x)$ as the *proxy* point of x .

THEOREM 3.4. *Consider any set $S^* \subseteq V$. Let C be the set of points returned by `Iterative-Sample`. Let U be the set of all points in $V - C$ that are unsatisfied by C regarding S^* . Then w.h.p., there exists an injective function $p : U \rightarrow V \setminus U$ such that, for any $x \in U$, $d(p(x), S^*) \geq d(x, C)$.*

PROOF. Throughout the analysis, we assume that $|U_\ell| \leq |R_\ell|/(3n^\epsilon)$ and $\frac{|R_\ell|}{n^\epsilon} \leq |R_{\ell+1}| \leq \frac{4|R_\ell|}{n^\epsilon}$ for each iteration ℓ . By Lemma 3.1, Corollary 3.3, and a simple union bound, it occurs w.h.p.

Let ℓ_f denote the final iteration. Let $A(\ell) := R_{\ell+1} \setminus U_\ell$. Roughly speaking, $A(\ell)$ is a set of candidate points, to which each $x \in U_\ell \cap D_\ell$ is mapped. Formally, we show the following properties:

1. for any $x \in U_\ell \cap D_\ell$ and any $y \in A(\ell)$, $d(x, S_\ell) \leq d(y, S_\ell) \leq d(y, S^*)$.
2. $|A(\ell)| \geq \sum_{\ell' \geq \ell}^{1/\epsilon} |U_{\ell'}|$.
3. $\bigcup_{\ell=1}^{\ell_f} (U_\ell \cap D_\ell) \supseteq U$.

The first property holds because any point in $R_{\ell+1} = R_\ell - D_\ell \supseteq A(\ell)$ is farther from $S_{\ell+1}$ than any point in D_ℓ , by the definition of the algorithm.

The inequality $d(y, S_\ell) \leq d(y, S^*)$ is immediate since y is satisfied by S_ℓ for C^* . The second property follows since $|A(\ell)| \geq |R_{\ell+1}| - |U_\ell| \geq \frac{|R_\ell|}{n^\epsilon} - \frac{|R_\ell|}{3n^\epsilon} \geq \sum_{\ell' \geq \ell}^{1/\epsilon} |U_{\ell'}|$. The last inequality holds because $|U_\ell| \leq \frac{|R_\ell|}{3n^\epsilon}$ and $|U_\ell|$ decrease by a factor of more than two as ℓ grows. We now prove the third property. The entire set of nodes V is partitioned into disjoint sets $D_1, D_2, \dots, D_{\ell_f}$ and R_{ℓ_f+1} . Further, for any $1 \leq \ell \leq \ell_f$, any node in $U \cap D_\ell$ is unsatisfied by S_ℓ regarding S^* , thus the node is also in $U_\ell \cap D_\ell$. Finally, the set $R_{\ell_f+1} \subseteq C$ are clearly satisfied by C .

We now construct $p(\cdot)$ as follows. Starting with $\ell = \ell_f$ down to $\ell = 1$, we map each unsatisfied node in $U_\ell \cap D_\ell$ to $|A(\ell)|$ so that no point in $A(\ell)$ is used more than once. This can be done using the second property. The requirement of $p(\cdot)$ that for any $x \in U$, $d(p(x), S^*) \geq d(x, C)$ is guaranteed by the first property. Finally, we simply ignore the points in $\bigcup_{\ell=1}^{\ell_f} (U_\ell \cap D_\ell) \setminus U$. This completes the proof. \square

3.2 MapReduce-KCenter

This section is devoted to proving Theorem 1.1. For the sake of analysis, we will consider the following variant of the k -center problem. In the $\text{kCenter}(V, T)$ problem, we are given two sets V and $T \subseteq V$ of points in a metric space, and we want to select a subset $S^* \subseteq T$ such that $|S^*| \leq k$ and S^* minimizes $\max_{x \in V} d(x, S)$ among all sets $S \subseteq T$ with at most k points. For notational simplicity, we let $\text{OPT}(V, T)$ denote the optimal solution for the problem $\text{kCenter}(V, T)$. Depending on the context, $\text{OPT}(V, T)$ may denote the cost of the solution. Since we are interested eventually in $\text{OPT}(V, V)$, we let $\text{OPT} := \text{OPT}(V, V)$.

PROPOSITION 3.5. *Let C be the set of centers returned by `Iterative-Sample`. Then w.h.p. we have that for any $x \in V$, $d(x, C) \leq 2\text{OPT}$.*

PROOF. Let $S^* := \text{OPT}$ denote a fixed optimal solution for $\text{kCenter}(V, V)$. Let U be the set of all points that are not satisfied by C regarding S^* . Consider any point x that is satisfied by C concerning S^* . Since it is satisfied, there exists a point $a \in C$ such that $d(a, x^{S^*}) \leq d(x, x^{S^*}) = d(x, S^*)$. Then by the triangle inequality, we have $d(x, C) \leq d(x, a) \leq d(x, x^{S^*}) + d(a, x^{S^*}) \leq 2d(x, S^*) \leq 2 \max_{y \notin U} d(y, S^*) = 2\text{OPT}$. Now consider any unsatisfied x . By Theorem 3.4, we know that w.h.p. there exists a proxy point $p(x)$ for any unsatisfied point $x \in U$. Then using the property of proxy points, we have $d(x, C) \leq d(p(x), S^*) \leq d(p(x), S^*) \leq \max_{y \notin U} d(y, S^*) \leq \text{OPT}$. \square

PROPOSITION 3.6. *Let C be the set of centers returned by `Iterative-Sample`. Then w.h.p. we have $\text{OPT}(C, C) \leq \text{OPT}(V, C) \leq \text{OPT}$.*

PROOF. Since the first inequality is trivial, we focus on proving the second inequality. Let S^* be an optimal solution for $\text{kCenter}(V, V)$. We construct a set $T \subseteq C$ as follows: for each $x \in S^*$, we add to T the vertex in C that is closest to x . Note that $|T| \leq k$ by construction. For any $x \in V$, we have

$$\begin{aligned} d(x, T) &\leq d(x, x^{S^*}) + d(x^{S^*}, T) = d(x, x^{S^*}) + d(x^{S^*}, x^C) \\ &\quad \text{[Since the closest point in } C \text{ to } x^{S^*} \text{ is in } T] \\ &\leq d(x, x^{S^*}) + d(x, x^{S^*}) + d(x, x^C) \\ &= 2d(x, S^*) + d(x, C) \leq 2\text{OPT} + d(x, C) \end{aligned}$$

By Proposition 3.5, we know that w.h.p. for all $x \in V$, $d(x, C) \leq 2\text{OPT}(V, V)$. Therefore, for all $x \in V$, $d(x, T) \leq 4\text{OPT}$. Since $\text{OPT}(V, C) \leq \text{OPT}(V, T)$, the second inequality follows. \square

THEOREM 3.7. *If \mathcal{A} is an algorithm that achieves an α -approximation for the k center problem, then w.h.p. the algorithm `MapReduce-kCenter` achieves a $(4\alpha + 2)$ -approximation for the k center problem.*

PROOF. By Proposition 3.6, $\text{OPT}(C, C) \leq 4\text{OPT}$. Let S be the set returned by `MapReduce-kCenter`. Since \mathcal{A} achieves an α -approximation for the k center problem, it follows that

$$\max_{x \in C} d(x, S) \leq \alpha \text{OPT}(C, C) \leq 4\alpha \text{OPT}$$

Let x be any point. By Proposition 3.5,

$$d(x, C) \leq 2\text{OPT}$$

Therefore

$$d(x, S) \leq d(x^C, S) + d(x, x^C) \leq (4\alpha + 2)\text{OPT}$$

\square

By setting the algorithm \mathcal{A} to be the 2-approximation of [17, 20], this proves Theorem 1.1.

3.3 MapReduce-KMedian

In the following, we will consider the following variants of the k -median problem similar to the variant of the k -center problem considered in the previous section. In the $\text{kMedian}(V, T)$ problem, we are given two sets V and $T \subseteq V$ of points in a metric space, and we want to select a subset $S^* \subseteq T$ such that $|S^*| \leq k$ and S^* minimizes $\sum_{x \in V} d(x, S)$ among all sets $S \subseteq T$ with at most k points. We let $\text{OPT}(V, T)$ denote a fixed optimal solution for $\text{kMedian}(V, T)$ or the optimal cost depending on the context. Note that we are interested in obtaining a solution that is comparable to $\text{OPT}(V, V)$. Hence, for notational simplicity, we let $\text{OPT} := \text{OPT}(V, V)$. In the $\text{Weighted-kMedian}(V, w)$ problem, we are given a set V of points in a metric space such that each

point x has a weight $w(x)$, and we want to select a subset $S^* \subseteq V$ such that $|S^*| \leq k$ and S^* minimizes $\sum_{x \in V} w(x)d(x, S)$ among all sets $S \subseteq V$ with at most k points. Let $\text{OPT}^w(V, w)$ denote a fixed optimal solution for a $\text{Weighted-kMedian}(V, w)$.

Recall that `MapReduce-kMedian` computes an approximate k -medians on C with each point x in C having a weight $w(x)$. Hence we first show that we can obtain a good approximate k -medians using only the points in C .

PROPOSITION 3.8. *Let $S^* := \text{OPT}$. Let C be the set of centers returned by `Iterative-Sample`. Then w.h.p., we have that $\sum_{x \in V} d(x, C) \leq 3\text{OPT}$.*

PROOF. Let U denote the set of points that are not unsatisfied by C regarding S^* . By Theorem 3.4, w.h.p. there exist proxy points $p(x)$ for all unsatisfied points. First consider any satisfied point $x \notin U$. It follows that there exists a point $a \in C$ such that $d(a, x^{S^*}) \leq d(x, x^{S^*}) = d(x, S^*)$. By the triangle inequality, $d(x, C) \leq d(x, a) \leq d(x, x^{S^*}) + d(a, x^{S^*}) \leq 2d(x, S^*)$. Hence $\sum_{x \notin U} d(x, C) \leq 2\text{OPT}$. We now argue with the unsatisfied points. $\sum_{x \in U} d(x, C) \leq \sum_{x \in U} d(p(x), S^*) \leq \text{OPT}$. The last inequality is due to property that $p(\cdot)$ is injective. \square

PROPOSITION 3.9. *Let C be the set returned by `Iterative-Sample`. Then w.h.p., $\text{OPT}(V, C) \leq 5\text{OPT}$.*

PROOF. Let S^* be an optimal solution for $\text{kMedian}(V, V)$. We construct a set $T \subseteq C$ as follows: for each $x \in S^*$, we add to T the vertex in C that is closest to x . By construction $|T| \leq k$. For any x , we have

$$\begin{aligned} d(x, T) &\leq d(x, x^{S^*}) + d(x^{S^*}, T) \leq d(x, x^{S^*}) + d(x^{S^*}, x^C) \\ &\quad \text{[The closest point in } C \text{ to } x^{S^*} \text{ is in } T] \\ &\leq d(x, x^{S^*}) + d(x, x^{S^*}) + d(x, x^C) = 2d(x, S^*) + d(x, C) \end{aligned}$$

By applying Proposition 3.8, w.h.p. we have $\sum_{x \in V} d(x, T) \leq 2 \sum_{x \in V} d(x, S^*) + \sum_{x \in V} d(x, C) \leq 5\text{OPT}$. Since T is a feasible solution for $\text{kMedian}(V, C)$, it follows that $\text{OPT}(V, C) \leq 5\text{OPT}$. \square

So far we have shown that we can obtain a good approximate solution for the $\text{kMedian}(V, V)$ even when we are restricted to C . However, we need a stronger argument, since `MapReduce-kMedian` only sees the weighted points in C , and not the entire point set V .

PROPOSITION 3.10. *Consider any subset of points $C \subseteq V$. For each point $y \in C$, let $w(y) = |\{x \in V - C \mid x^C = y\}| + 1$. Then we have $\text{OPT}^w(C, w) \leq 2\text{OPT}(V, C)$.*

PROOF. Let $T^* := \text{OPT}(V, C)$. Let $\bar{C} := V \setminus C$. For each point $x \in \bar{C}$, we have $d(x, x^C) + d(x, x^{T^*}) \geq d(x^C, x^{T^*}) \geq d(x^C, T^*)$. Therefore $\sum_{x \in \bar{C}} d(x, T^*) \geq \sum_{x \in \bar{C}} (d(x^C, T^*) - d(x, x^C))$. Further we have,

$$\begin{aligned} 2 \sum_{x \in \bar{C}} d(x, T^*) &\geq \sum_{x \in \bar{C}} d(x^C, T^*) + \sum_{x \in \bar{C}} (d(x, T^*) - d(x, x^C)) \\ &\geq \sum_{x \in \bar{C}} d(x^C, T^*) \quad [d(x, T^*) \geq d(x, C), \text{ since } T^* \subseteq C] \\ &= \sum_{y \in C} \sum_{x \in \bar{C}: x^C = y} d(y, T^*) = \sum_{y \in C} (w(y) - 1) d(y, T^*) \end{aligned}$$

Hence we have $\text{OPT}(V, C) = 2 \sum_{x \in V} d(x, T^*) \geq \sum_{y \in C} w(y) d(y, T^*)$. Since T^* is a feasible solution for $\text{Weighted-kMedian}(C, w)$, it follows that $\text{OPT}^w(C, w) \leq 2\text{OPT}(V, C)$. \square

THEOREM 3.11. *If \mathcal{A} is an algorithm that achieves an α -approximation for `Weighted-kMedian`, w.h.p. the algorithm `MapReduce-kMedian` achieves a $(10\alpha + 3)$ -approximation for `kMedian`.*

PROOF. It follows from Proposition 3.9 and Proposition 3.10 that w.h.p., $\text{OPT}^w(C, w) \leq 10\text{OPT}$.

Let S be the set returned by `MapReduce-kMedian`. Since \mathcal{A} achieves an α -approximation for `Weighted-kMedian`, it follows that

$$\sum_{y \in C} w(y)d(y, S) \leq \alpha \text{OPT}^w(C, w) \leq 10\alpha \text{OPT}$$

We have

$$\begin{aligned} \sum_{x \in V} d(x, S) &= \sum_{y \in C} d(y, S) + \sum_{x \in \bar{C}} d(x, S) \\ &\leq \sum_{y \in C} d(y, S) + \sum_{x \in \bar{C}} d(x, (x^C)^S) \\ &\leq \sum_{y \in C} d(y, S) + \sum_{x \in \bar{C}} (d(x, x^C) + d(x^C, S)) \\ &= \sum_{y \in C} w(y)d(y, S) + \sum_{x \in \bar{C}} d(x, C) \end{aligned}$$

By Proposition 3.8 (with S^* equal to $\text{OPT}(V, V)$), we get that

$$\sum_{x \in V} d(x, C) \leq 3 \sum_{x \in V} d(x, S^*) = 3\text{OPT}$$

Therefore

$$\sum_{x \in V} d(x, S) \leq (10\alpha + 3)\text{OPT}$$

□

Using the $(6 + 2/3)$ -approximation algorithm of [11], we complete the proof of Theorem 1.2.

4. EXPERIMENTS

In this section we provide an experimental study of the algorithms introduced in this paper [18]. The focus for this section is on the k -median objective because this is where our algorithm gives the largest increase in performance. Unfortunately, our sampling algorithm does not perform well for the k -center metric. This is because the k -center objective is quite sensitive to sampling. Since the maximum distance from a point to a center is considered in the objective, if the sampling algorithm misses even one important point then the objective can drastically increase. Due to space, the details of these experiments are omitted. From now on, the metric we discuss is fixed as the k -median objective.

We compare our algorithm `MapReduce-kMedian` to the local search algorithm [4, 22], `MapReduce-Divide-kMedian` and a parallelized version of Lloyd’s algorithm [28, 7, 1]. For both `MapReduce-kMedian` and `MapReduce-Divide-kMedian` we used Lloyd’s algorithm and local search as subroutines in separate experiments. The local search algorithm is the best known approximation algorithm. The algorithm `MapReduce-Divide-kMedian` is a partitioning based parallelization of any arbitrary sequential clustering algorithm and details of this algorithm are given in Appendix A. Finally, Lloyd’s algorithm is perhaps the most popular algorithm for clustering used in practice and details of the parallelized version can be found in Appendix B. When either `MapReduce-kMedian` or `MapReduce-Divide-kMedian` is used with local search as

a subroutine then these algorithms give constant approximation guarantees as shown in Section 3.3 and Appendix A. However, Lloyd’s is quite popular in practice and it is natural to use this algorithm as a subroutine for either algorithm. For ease of explanation we denote the algorithms as `LocalSearch` for local search, `Parallel-Lloyd` for the parallelized Lloyd’s algorithm, `Divide-Lloyd` for `MapReduce-Divide-kMedian` coupled with Lloyd’s algorithm, `Divide-LocalSearch` for `MapReduce-Divide-kMedian` coupled with local search, `Sampling-LocalSearch` for `MapReduce-kMedian` with local search and `Sampling-Lloyd` for `MapReduce-kMedian` with Lloyd’s algorithm.

Experiment Overview: We generate a random set of points in \mathbb{R}^3 . Our data set consists of k centers and randomly generated points around the centers to create clusters. The k centers are randomly positioned in a unit cube. The number of points generated within a cluster is randomly generated using a Zipf distribution. Concretely, let $\{C_i\}_{1 \leq i \leq k}$ be the set of clusters. Given a fixed number of points, a unique point is associated to the center C_i with probability $i^\alpha / \sum_{i=1}^k i^\alpha$ where α is the parameter of the Zipf distribution. Notice that when $\alpha = 0$, all clusters will have roughly the same size and as α grows the size of the clusters become uniform. The distance of a point from its center follows a normal distribution with a fixed global standard deviation σ . Each experiment with the same parameter set was repeated three times and the average was calculated. When running local search or Lloyd’s algorithm, the seed centers were set arbitrarily.

All experiments were performed on a single machine. When running `MapReduce` algorithms, we simulate each machine that is used by the algorithm. For a given round, we record the time it takes for the machine that runs the longest in the round. Then we sum this time over all the rounds to get the final running time of the parallel algorithms. In these experiments, the communication cost was ignored. The specifications of the machine were Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz and with memory size 8GB. We used the standard `clock()` function to measure the time that each experiment takes. All parallel algorithms were simulated assuming that there are 100 machines. For the algorithm `MapReduce-kMedian` the value of ϵ was set to .1 for the sampling probability.

Results: Due to space constraints, we give a brief summary of our results. The data can be found in Figures 1 and 2. For the data in the figures, the number of points is the only variable, and other parameters are fixed: $\sigma = 0.1$, $\alpha = 0$ and $k = 25$. The cost of the algorithms’ objectives is normalized to the cost of `Parallel-Lloyd` in the figures. Our experiments show that `Sampling-Lloyd` and `Sampling-LocalSearch` achieve a significant speedup over `Parallel-Lloyd` (over 20x), a speedup of more than ten times over `Divide-LocalSearch` and an enormous speedup over `LocalSearch` (over 1000x) as seen in Figure 1. Further, this speedup is achieved with negligible loss in performance; our algorithm’s objective performs better than the `Parallel-Lloyd` when the number of points is sufficiently large and approaches the cost of the `LocalSearch` as the number of points grow. Finally, we consider the performance of `Sampling-LocalSearch` and `Sampling-Lloyd` against `Divide-Lloyd` on very large data sets in Figure 2. These algorithms were chosen because they are the most scalable and perform well. These trials show that `Sampling-LocalSearch` achieves slightly slower running time and similar performance as `Divide-Lloyd`. The algorithm `Sampling-Lloyd` achieves

	Number of points	10,000	20,000	40,000	100,000	200,000	400,000	1,000,000
cost	Parallel-Lloyd	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	Divide-Lloyd	1.030	1.088	1.132	1.033	1.051	0.994	1.023
	Divide-LocalSearch	0.999	1.024	1.006	0.999	1.008	0.999	1.010
	Sampling-Lloyd	1.086	1.165	1.051	1.138	1.068	1.095	1.132
	Sampling-LocalSearch	1.018	1.019	1.011	1.006	1.024	1.025	1.029
	LocalSearch	0.948	0.964	0.958	N/A	N/A	N/A	N/A
time	Parallel-Lloyd	0.0	3.3	6.0	18.0	29.3	52.7	205.7
	Divide-Lloyd	0.0	0.3	0.3	1.3	1.0	1.0	2.7
	Divide-LocalSearch	5.0	5.0	5.0	6.0	9.0	19.0	70.7
	Sampling-Lloyd	0.3	0.0	0.3	0.7	1.3	3.0	4.0
	Sampling-LocalSearch	1.7	2.3	3.0	4.3	6.0	8.3	11.0
	LocalSearch	666.7	943.0	2569.3	N/A	N/A	N/A	N/A

Figure 1: The relative cost and running time of clustering algorithms when the number of points is not too large. The costs are normalized to that of Parallel-Lloyd. The running time is given in seconds.

	Number of points	2,000,000	5,000,000	10,000,000
cost	Parallel-Lloyd	1.000	1.000	1.000
	Divide-Lloyd	1.018	1.036	1.000
	Sampling-Lloyd	1.064	1.106	1.073
	Sampling-LocalSearch	1.027	1.019	1.015
time	Parallel-Lloyd	458.0	1333.3	702.3
	Divide-Lloyd	8.3	24.7	50.7
	Sampling-Lloyd	8.0	18.3	38.0
	Sampling-LocalSearch	16.3	29.3	86.3

Figure 2: The relative cost and running time of the scalable algorithms when the number of points are large. The costs were normalized to that of Parallel-Lloyd. The running time is given in seconds.

a speedup of about 25% over Divide-Lloyd when the number of points is sufficiently large. Overall the experiments show that our sampling algorithm when coupled with Lloyd’s algorithm as a subroutine runs faster than any previously known algorithm that we considered and this speedup is achieved with little loss in performance. Experiments were performed when varying the parameters α , k and σ and similar results were obtained. Due to space these results are omitted.

Acknowledgments: The authors thank Kyle Fox for his help with the implementation of the algorithms and several discussions.

APPENDIX

A. A SIMPLE PARTITIONING ALGORITHM

In this section we show a simple partitioning based algorithm that can be implemented in MapReduce. We note that this algorithm and the following analysis have also been considered by Guha et al. [21] in the streaming model.

The following proposition follows from the algorithm description. The total memory used by the algorithm is $O(k^2n)$ in the case that the edge distances need be explicitly stored on the machines. This is because in Step (9), $O(\sqrt{n})$ clusters of k points are sent to a single machine along with their pairwise edges. It can be seen that this is essentially the minimum memory needed because if we make the number of clusters smaller than the number of edges within each cluster will be larger than $O(n)$.

PROPOSITION A.1. *We can implement MapReduce-Divide-kMedian in MapReduce using $O(1)$ rounds.*

Algorithm 4 MapReduce-Divide-kMedian(V, E, k):

- 1: Let $n = |V|$.
 - 2: Partition V into disjoint sets S_1, \dots, S_ℓ of size $O(\sqrt{n})$.
 - 3: **for** $i = 1$ to ℓ **do**
 - 4: Assign S_i and all the edges between vertices in S_i to the i -th machine.
 - 5: On the i -th machine, run a k -median clustering algorithm \mathcal{A} with $\langle S_i, k \rangle$ as input to find a set $C_i \subseteq S_i$ of k centers.
 - 6: On the i -th machine, for each $y \in C_i$, compute $w(y) = |\{x \in S^i \setminus C_i \mid d(x, y) = d(x, C_i)\}| + 1$.
 - 7: **end for**
 - 8: Let $C = \bigcup_{i=1}^{\ell} C_i$.
 - 9: Send the vertices of C together with the numbers $w(\cdot)$ to a single machine.
 - 10: Run a Weighted-kMedian algorithm \mathcal{A} on that machine with $\langle C, w, k \rangle$ as input.
 - 11: Return the set constructed by \mathcal{A} .
-

From the analysis given in [21], we have the following theorem which can be used to bound the approximation factor of MapReduce-Divide-kMedian.

THEOREM A.2 (THEOREM 2.2 IN [21]). *Consider any set of n vertices arbitrarily partitioned into disjoint sets S_1, \dots, S_ℓ . The sum of the optimum solution values for the k -median problem on the ℓ sets of vertices is at most twice the cost of the optimum k -median problem solution for all n vertices.*

COROLLARY A.3 ([21]). *If the algorithm \mathcal{A} achieves an α -approximation for the k -median problem, the algorithm MapReduce-Divide-kMedian achieves a 3α -approximation*

for the k -median problem.

B. PARALLELIZED LLOYD'S ALGORITHM

In this section, we give a sketch of the parallel Lloyd's algorithm used in the experiments since it is not a well known algorithm. More detail can be found in [7, 1]. The algorithm begins by partitioning the points evenly across the machines and these points will remain on the machines. The algorithm initializes the k centers to be an arbitrary set of points. In each iteration, the parallel Lloyd's algorithm improves the centers as follows. The k centers are sent to each of the machines. Each machine, for the set of points assigned to the machine, assigns each point to the closest center. We assume that points are in an Euclidian space. For each cluster, the average of the points in the cluster is computed along with the number of points assigned to the center. All machines send this information to a single machine. For each center, this machine aggregates the points assigned to the center over all partitioned sets along with the centers and then updates the center to be the average of these points. It is important to note that the solution computed by the algorithm is the same as the sequential version of Lloyd's algorithm.

C. REFERENCES

- [1] Google: Cluster computing and mapreduce. <http://code.google.com/edu/submissions/mapreduce-minilecture/listing.html>.
- [2] Pankaj K. Agarwal and Nabil H. Mustafa. k -means projective clustering. In *PODS*, pages 155–165, 2004.
- [3] David Arthur and Sergei Vassilvitskii. k -means++: the advantages of careful seeding. In *SODA*, pages 1027–1035, 2007.
- [4] Vijay Arya, Naveen Garg, Rohit Khandekar, Adam Meyerson, Kamesh Munagala, and Vinayaka Pandit. Local search heuristics for k -median and facility location problems. *SIAM J. Comput.*, 33(3):544–562, 2004.
- [5] Yair Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *FOCS*, pages 184–193, 1996.
- [6] Yair Bartal. On approximating arbitrary metrics by tree metrics. In *STOC*, pages 161–168, 1998.
- [7] Michael Berry. Mapreduce and k -means clustering. <http://blog.data-miners.com/2008/02/mapreduce-and-k-means-clustering.html>, 2008.
- [8] Guy E. Blelloch and Kanat Tangwongsan. Parallel approximation algorithms for facility-location problems. In *SPAA*, pages 315–324, 2010.
- [9] Moses Charikar, Chandra Chekuri, Ashish Goel, and Sudipto Guha. Rounding via trees: Deterministic approximation algorithms for group steiner trees and k -median. In *STOC*, pages 114–123, 1998.
- [10] Moses Charikar and Sudipto Guha. Improved combinatorial algorithms for the facility location and k -median problems. In *FOCS*, pages 378–388, 1999.
- [11] Moses Charikar, Sudipto Guha, Éva Tardos, and David B. Shmoys. A constant-factor approximation algorithm for the k -median problem. *J. Comput. Syst. Sci.*, 65(1):129–149, 2002.
- [12] Ke Chen. On k -median clustering in high dimensions. In *SODA*, pages 1177–1185, 2006.
- [13] Ke Chen. A constant factor approximation algorithm for k -median clustering with outliers. In *SODA*, pages 826–835, 2008.
- [14] Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-cover in map-reduce. In *WWW*, pages 231–240, 2010.
- [15] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
- [16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, pages 137–150, 2004.
- [17] M. E. Dyer and A. M. Frieze. A simple heuristic for the p -centre problem. *Operations Research Letters*, 3(6):285 – 288, 1985.
- [18] Alina Ene, Sungjin Im, and Benjamin Moseley. k -median clustering implementation. <http://www.cs.uiuc.edu/homes/bmosele2/clusteringimpl/>, 2011.
- [19] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms*, 6(4), 2010.
- [20] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293 – 306, 1985.
- [21] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev

- Motwani, and Liadan O’Callaghan. Clustering data streams: Theory and practice. *IEEE Trans. Knowl. Data Eng.*, 15(3):515–528, 2003.
- [22] Anupam Gupta and Kanat Tangwongsan. Simpler analyses of local search algorithms for facility location. *CoRR*, abs/0809.2554, 2008.
- [23] Ralf Herwig, Albert J. Poustka, Christine Müller, Christof Bull, Hans Lehrach, and John O’Brien. Large-Scale Clustering of cDNA-Fingerprinting Data. *Genome Research*, 9(11):1093–1105, November 1999.
- [24] Dorit S. Hochbaum and David B. Shmoys. A best possible heuristic for the k-center problem. *MATHEMATICS OF OPERATIONS RESEARCH*, 10(2):180–184, May 1985.
- [25] U. Kang, Charalampos Tsourakakis, Ana P. Appel, Christos Faloutsos, and Jure Leskovec. Hadi: Fast diameter estimation and mining in massive graphs with hadoop. Technical report, School of Computer Science, Carnegie Mellon University Pittsburgh, December 2008.
- [26] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.
- [27] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.
- [28] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–136, 1982.
- [29] Aiysha Ma and Ishwar K. Sethi. Distributed k-median clustering with application to image clustering. In *PRIS*, pages 215–220, 2007.
- [30] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.
- [31] Mikkel Thorup. Quick k-median, k-center, and facility location for sparse graphs. *SIAM J. Comput.*, 34(2):405–432, 2004.
- [32] Tom White. *Hadoop: The Definitive Guide*. O’Reilly Media, 2009.