A Framework for Parallelizing Hierarchical Clustering Methods

Silvio Lattanzi[1], Thomas Lavastida[2](✉), Kefu Lu[3], and Benjamin Moseley[2]

[1] Google Zürich, Zürich, Switzerland, silviol@google.com
[2] Tepper School of Business, Carnegie Mellon University, Pittsugh, PA, {tlavasti,moseleyb}@andrew.cmu.edu
[3] Computer Science Department, Washington and Lee University, Lexington, VA, klu@wlu.edu

Abstract. Hierarchical clustering is a fundamental tool in data mining, machine learning and statistics. Popular hierarchical clustering algorithms include top-down divisive approaches such as bisecting k-means, k-median, and k-center and bottom-up agglomerative approaches such as single-linkage, average-linkage, and centroid-linkage. Unfortunately, only a few scalable hierarchical clustering algorithms are known, mostly based on the single-linkage algorithm. So, as datasets increase in size every day, there is a pressing need to scale other popular methods.
We introduce efficient distributed algorithms for bisecting k-means, k-median, and k-center as well as centroid-linkage. In particular, we first formalize a notion of closeness for a hierarchical clustering algorithm, and then we use this notion to design new scalable distributed methods with strong worst case bounds on the running time and the quality of the solutions. Finally, we show experimentally that the introduced algorithms are efficient and close to their sequential variants in practice.

Keywords: Hierarchical Clustering, Parallel and Distributed Algorithms, Clustering, Unsupervised Learning

1 Introduction

Thanks to its ability in explaining nested structures in real world data, hierarchical clustering is a fundamental tool in any machine learning or data mining library. In recent years the method has received a lot of attention [23, 24, 15, 35, 7, 10, 34, 37, 5]. But despite these efforts, almost all proposed hierarchical clustering techniques are sequential methods that are difficult to apply on large data sets.

The input to the hierarchical clustering problem is a set of points and a function specifying either their pairwise similarity or their dissimilarity. The output of the problem is a rooted tree representing a hierarchical structure of the input data, also known as a dendrogram. The input points are the leaves of this tree and subtrees induced by non-leaf nodes represent clusters. These clusters should also become more refined when the root of the corresponding subtree is at a lower level in the tree. Hierarchical clustering is useful because the number of clusters does not need to be specified in advance and because the hierarchical structure yields a taxonomy that allows for interesting interpretations of the data set. For an overview of hierarchical clustering methods refer to [31, 27, 18].

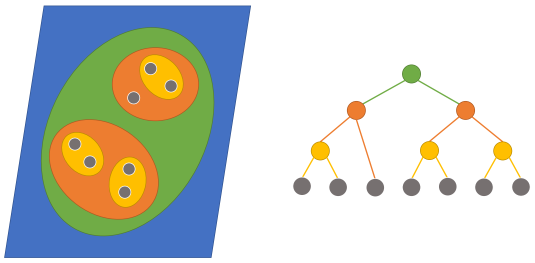


Fig. 1. A Hierarchical Clustering Tree. The grey leaves are the input data points. Internal nodes represent a cluster of the leaves in the subtree rooted at the internal node.

Several algorithms have emerged as popular approaches for hierarchical clustering. Different techniques are used depending on the context because each method has its own advantages and disadvantages. There are various classes of data sets where each method outperforms the others. For example, the centroid-linkage algorithm has been used for biological data such as genes [12], whereas, an alternative method, bisecting k-means, is popular for document comparison [36]. The most commonly used methods can be categorized into two families: agglomerative algorithms and divisive algorithms.

Divisive algorithms are top-down. They partition the data starting from a single cluster and then refine the clusters iteratively layer by layer. The most commonly used techniques to refine clusters are k-means, k-median, or k-center clustering with $k = 2$. These divisive algorithms are known as bisecting k-means (respectfully, median, center) algorithms [22]. Agglomerative algorithms are based on a bottom up approach (see [17] for details). In an agglomerative algorithm, all points begin as their own cluster. Clusters are then merged through some merging strategy. The choice of merging strategy determines the algorithm. Common strategies include single-linkage, average-linkage and centroid-linkage.

Most of these algorithms are inherently sequential; they possess a large number of serial dependencies and do not lend themselves to efficient parallelization. For example, in centroid-linkage one cannot simultaneously perform many merges because the selection of which clusters to merge may depend strongly on prior merges (and their resultant centroids).

Recently, there has been interest in making hierarchical clustering scalable [24, 23, 15, 35, 5, 33, 30, 38]. Nevertheless most prior work has focused on scaling the single-linkage algorithm; efficient MapReduce and Spark algorithms are known for this problem [24, 23, 5, 38]. This includes the result of [38] giving strong theoretical guarantees and practical performance for scaling single-linkage clustering. This is unsurprising because single-linkage can be reduced to computing a Minimum-Spanning-Tree [14], and there has been a line of work on efficiently computing minimum spanning trees in parallel and distributed settings [2, 26, 28, 1, 32]. Unfortunately this approach does not extend to other hierarchical clustering techniques. In contrast, to the best of our knowledge no efficient distributed algorithm is known for centroid-linkage or divisive clustering methods. Thus, scaling methods such as centroid-linkage and bisecting k-means are open problems and the main focus of this paper.

Our Contribution: In this paper we introduce fast scalable hierarchical clustering algorithms. The main results of the paper are the following:

A Theoretical Framework: This paper develops a theoretical framework for scaling hierarchical clustering methods. We introduce the notion of *closeness* between two hierarchical clustering algorithms. Intuitively, two algorithms are close if they make provably close or similar decisions. This enforces that our scalable algorithms produce similar solutions to their sequential counterpart. Using this framework, the paper formalizes the root question for scaling existing methods.

Provably Scalable Algorithms: We introduce fast scalable algorithms for centroid-linkage and the bisecting k-means, k-median and k-center algorithms. These new algorithms are the main contribution of the paper. The algorithms are proved to be close to their sequential counterparts and efficient in parallel and distributed models. These are the first scalable algorithms for divisive k-clustering as well as centroid-linkage.

Empirical results: We empirically study the algorithms on three datasets to show that they are efficient. The empirical results demonstrate that the distributed algorithms are closer to their sequential counterparts than the theory suggests. This shows that the new methods produce clusterings remarkably similar to those produced by the sequential methods.

The algorithms can be used for most data sets. The scalable bisecting k-clustering algorithms apply to data belonging to any metric space. For centroid linkage, we assume that the input data points belong to some Euclidean space so that computing the centroid of a finite set of points is well defined. In this case, our techniques generalize to any distance function between points in Euclidean space for which a family of Locality Sensitive Hash (LSH) functions is known, such as distances induced by an ℓ_p-norm for $p \in (0, 2]$ [11].

2 Preliminaries

In this section we formally define the hierarchical clustering problem, describe popular sequential approaches, and provide other necessary background information.

Problem Input: The input is a set S of n data points. The distance between points specifies their dissimilarity. Let $d(u, v) \geq 0$ denote the distance between two points $u, v \in S$. It is assumed that d is a metric.

Problem Output: The output is a rooted tree where all of the *input* points are at the leaves. Internal nodes represent clusters; the leaves of the subtree rooted at a node correspond to the points in that specific cluster.
Computational Model: We analyze our algorithms in the massively parallel model of computation [26, 20][4]. Let N be the input size. In this model we have $m = O(N^{1-\delta})$ machines with *local* memory of size $\tilde{O}(N^{\delta})$, for constant $\delta > 0$. The $\tilde{O}$ suppresses logarithmic factors. Notice that the total amount of memory used is near linear. The computation occurs in rounds and during each round each machine runs a sequential polynomial time algorithm on the data assigned to it. No communication between machines is allowed during a round. Between rounds, machines are allowed to communicate as long as each machine receives no more communication than its memory and no computation occurs. Ideally, in this model one would like to design algorithms using a number of rounds that is no more than logarithmic in the input size.
k-Clustering Methods: We recall the definitions of k-{center,median,means} clusterings. Let $C = \{c_1, c_2, \ldots, c_k\}$ be k distinct points of S called centers. For $x \in S$ let $d(x, C) = \min_{c \in C} d(x, c)$ We say that these centers solve the k-{center,median,means} problem if they optimize the following objectives, respectively: k-center: $\min_C \max_{x \in S} d(x, C)$, k-medians: $\min_C \sum_{x \in S} d(x, C)$ and finally k-means: $\min_C \sum_{x \in S} d(x, C)^2$.

The choice of centers induces a clustering of S in the following natural way. For $i = 1, \ldots, k$ let $S_i = \{x \in S \mid d(x, c_i) = d(x, C)\}$, that is we map each point to its closest center and take the clustering that results. In general it is NP-hard to find the optimal set of centers for each of these objectives, but efficient $O(1)$-approximations are known [19, 9, 25].
Classic Divisive Methods: We can now describe the classical divisive k-clustering algorithms. The pseudocode for this class of methods is given in Algorithm 1. As stated before, these methods begin at the root of the cluster tree corresponding to the entire set S and recursively partition the set until we reach the leaves of the tree. Note that at each node of the tree, we use an optimal 2-clustering of the current set of points to determine the two child subtrees of the current node.
Classic Agglomerative Methods: Recall that agglomerative methods start with each data point as a singleton cluster and iteratively merge clusters to build the tree. The choice of clusters to merge is determined by considering some cost on pairs of clusters and then choosing the pair that minimizes the cost. For example, in average-linkage the cost is the average distance between the clusters, and for centroid-linkage the cost is the distance between the clusters' centroids. In this paper we focus on the centroid-linkage method, and have provided pseudocode for this method in Algorithm 2.
Notation: We present some additional notation and a few technical assumptions. Let X be a finite set of points and x a point in X. We define the ball of radius R around x, with notation $B(x, R)$, as the set of points with distance at most

[4] This model is widely used to capture the class of algorithms that scale in frameworks such as Spark and MapReduce.

```
DivisiveClustering($S$)
if $|S| = 1$ then
    Return a leaf node corresponding to $S$
else
    Let $S_1, S_2$ be an optimal 2-clustering of $S$ /* One of the means, median,
        or center objectives is used                                  */
    $T_1 \leftarrow$ DivisiveClustering($S_1$)
    $T_2 \leftarrow$ DivisiveClustering($S_2$)
    Return a tree with root node $S$ and children $T_1, T_2$
end
```

Algorithm 1: Standard Divisive Clustering

```
CentroidClustering($S$)
Let $T$ be an empty tree
For each $x \in S$ add a leaf node corresponding to the cluster $\{x\}$ to $T$
Let $\mathcal{C}$ be the current set of clusters
while $|\mathcal{C}| > 1$ do
    $S_1, S_2 \leftarrow \arg\min_{A,B \in \mathcal{C}} d(\mu(A), \mu(B))$ /* $\mu(A) :=$ centroid of $A$     */
    Add a node to $T$ corresponding to $S_1 \cup S_2$ with children $S_1$ and $S_2$
    $\mathcal{C} \leftarrow \mathcal{C} \setminus \{S_1, S_2\} \cup \{S_1 \cup S_2\}$
end
Return the resulting tree $T$
```

Algorithm 2: Centroid Linkage Clustering

R from x in the point set X, i.e. $B(x, R) = \{y \mid d(x, y) \leq R, y \in X\}$. Let $\Delta(X) = \max_{x,y \in X} d(x, y)$ be the maximum distance between points of X. When X is a subset of Euclidean space, let $\mu(X) = \frac{1}{|X|} \sum_{x \in X} x$ be the centroid of X. Finally, WLOG we assume that all points and pairwise distances are distinct[5] and that the ratio between the maximum and minimum distance between two points is polynomial in n.

3 A Framework for Parallelizing Hierarchical Clustering Algorithms

We now introduce our theoretical framework that we use to design and analyze scalable hierachical clustering algorithms. Notice that both divisive and agglomerative methods use some cost function on pairs of clusters to guide the decisions of the algorithm. More precisely, in divisive algorithms the current set of points S is partitioned into S_1, and S_2 according to some cost $c(S_1, S_2)$. Similarly, agglomerative algorithms merge clusters S_1 and S_2 by considering some cost $c(S_1, S_2)$. So in both settings the main step consists of determining the two sets S_1 and S_2 using different cost functions. As an example, observe that $c(S_1, S_2)$ is the 2-clustering cost of the sets S_1 and S_2 in the divisive method above and that $c(S_1, S_2) = d(\mu(S_1), \mu(S_2))$ in centroid linkage.

[5] We can remove this assumption by adding a small perturbation to every point.

The insistence on choosing S_1, S_2 to minimize the cost S_1, S_2 leads to the large number of serial dependencies that make parallelization of these methods difficult. Thus, the main idea behind this paper is to obtain more scalable algorithms by relaxing this decision making process to make near optimal decisions. This is formalized in the following definitions.

Definition 1 (α-close sets). *Let c be the cost function on pairs of sets and let S_1, S_2 be the two sets that minimize $c(S_1, S_2)$. Then we say that two sets S_1', S_2' are α-**close** to the optimum sets for cost c if $c(S_1', S_2') \leq \alpha c(S_1, S_2)$, for $\alpha \geq 1$.*

Definition 2 (α-close algorithm). *We say that a hierarchical clustering algorithm is α-**close** to the optimal algorithm for cost function c if at any step of the algorithm the sets selected by the algorithm are α-**close** for cost c, for $\alpha \geq 1$.* [6]

A necessary condition for efficiently parallelizing an algorithm is that it must not have long chains of dependencies. Now we formalize the concept of chains of dependencies.

Definition 3 (Chain of dependency). *We say that a hierarchical clustering algorithm has a chain of dependencies of length ℓ, if every decision made by the algorithm depends on a chain of at most ℓ previous decisions.*

We now define the main problem tackled in this paper.

Problem 1 *Is it possible to design hierarchical clustering algorithms that have chain of dependencies of length at most $\mathrm{poly}(\log n)$ and that are α-close, for small α, for the k-means, the k-center, the k-median and centroid linkage cost functions?*

It is not immediately obvious that allowing our algorithms to be α-close will admit algorithms with small chains of dependencies. In sections 4.1 and 4.2 we answer this question affirmatively for divisive k-clustering methods and centroid linkage[7]. Then in section 4.3 we show how to efficiently implement these algorithms in the massively parallel model of computation. Finally, we give experimental results in section 5, demonstrating that our algorithms perform close to the sequential algorithms in practice.

4 Fast Parallel Algorithms for Clustering

4.1 Distributed Divisive k-Clustering

We now present an $O(1)$-close algorithm with dependency chains of length $O(\log(n))$ under the assumption that the ratio of the maximum to the minimum distance between points is polynomial in n.

[6] Note that the guarantees is on each single choice made by the algorithm but not on all the choices together.

[7] In prior work, Yaroslavtsev and Vadapalli [38] give an algorithm for single-linkage clustering with constant-dimensional Euclidean input that fits within our framework.

As discussed in Sections 2 and 3, the main drawback of Algorithm 1 is that its longest chains of dependencies an be linear in the size of the input[8]. We modify this algorithm to overcome this limitation while remaining $O(1)$-close with respect to the clustering cost objective. In order to accomplish this we maintain the following invariant. Each time we split S into S_1 and S_2, each set either contains a constant factor fewer points than S or the maximum distance between any two points has been decreased by a constant factor compared to the maximum distance in S. This property will ensure that the algorithm has a chain of dependency of logarithmic depth. We present the pseudocode for the new algorithm in Algorithms 3 and 4.

```
1  CloseDivisiveClustering($S$)
2  if $|S| = 1$ then
3      Return a leaf node corresponding to $S$
4  else
5      Let $S_1, S_2$ be an optimal 2-clustering of $S$ /* One of the means, median,
           or center objectives is used                                  */
6      $S_1, S_2 \leftarrow$ Reassign($S_1, S_2, \Delta(S)$) /* Key step, see Algorithm 4     */
7      $T_1 \leftarrow$ CloseDivisiveClustering($S_1$)
8      $T_2 \leftarrow$ CloseDivisiveClustering($S_2$)
9      Return a tree with root $S$ and children $T_1, T_2$
10 end
```

Algorithm 3: $O(1)$-Close Divisive k-Clustering Algorithm

The goal of this subsection is to show the following theorem guaranteeing that Algorithm 3 is provably close to standard divisive k-clustering algorithms, while having a small chain of serial dependencies.

Theorem 1. *Algorithm 3 is $O(1)$-close for the k-center, k-median, and k-means cost functions and has a chain of dependencies of length at most $O(\log n)$.*

The main difference between Algorithm 1 and Algorithm 3 is the reassignment step. This step's purpose is to ensure that the invariant holds at any point during the algorithm as shown in the following lemma. Intuitively, if both S_1 and S_2 are contained within a small ball around their cluster centers, then the invariant is maintained. However, if this is not the case, then there are many points "near the border" of the two clusters, so we move these around to maintain the invariant.

Lemma 1. *After the execution of Reassign(S_1, S_2) in Algorithm 3 either $|S_1| \leq \frac{7}{8}|S|$ or $\Delta(S_1) \leq \frac{1}{2}\Delta(S)$. Similarly, either $|S_2| \leq \frac{7}{8}|S|$ or $\Delta(S_2) \leq \frac{1}{2}\Delta(S)$.*

Proof. Let S_1, S_2 be the resulting clusters and v_1, v_2 be their centers. Consider the sets $B_i = B(v_i, \Delta(S)/8) \cap S$, for $i = 1, 2$. If $S_1 \subseteq B_1$ and $S_2 \subseteq B_2$, then both

[8] Consider an example where the optimal 2-clustering separates only 1 point at a time.

```
Reassign($S_1, S_2, \Delta$)
Let $v_1, v_2$ be the centers of $S_1, S_2$, respectively
for $i = 1, 2$ do
    $B_i \leftarrow B(v_i, \Delta/8) \cap (S_1 \cup S_2)$
end
if $S_1 \subseteq B_1$ and $S_2 \subseteq B_2$ then
    Return $S_1, S_2$
else
    $U \leftarrow (S_1 \setminus B_1) \cup (S_2 \setminus B_2)$
    if $|U| \leq n/c$ /* $c$ is constant parameter, default is $c = 4$ */
     then
        Assign $U$ to the smaller of $B_1$ and $B_2$
    else
        Split $U$ evenly between $B_1$ and $B_2$
    end
    Return $B_1, B_2$
end
```

Algorithm 4: Reassign Subroutine for Divisive Clustering

clusters are contained in a ball of radius $\Delta(S)/8$. By the triangle inequality the maximum distance between any two points in either S_1 or S_2 is at most $\Delta(S)/2$.[9]

Otherwise, consider U, the set of points not assigned to any B_i. We consider $c = 4$ as in the default case. If $|U| \leq |S|/4$, then the algorithm assigns U to the smaller of B_1 and B_2 and the resulting cluster will have size at most $3|S|/4$ since the smaller set has size at most $|S|/2$. Furthermore the other cluster is still contained within a ball of radius $\Delta/8$ and thus the maximum distance between points is at most $\Delta(S)/2$. If $|U| \geq |S|/4$ then the points in U are distributed evenly between S_1 and S_2. Both sets in the recursive calls are guaranteed to have less than $|S| - |U|/2 \leq \frac{7}{8}|S|$ points since U was evenly split. Similar properties can be shown for other values of c.

Next we can show that the algorithm is $O(1)$-close by showing that the algorithm is an $O(1)$-approximation for the desired k-clustering objective in each step.

Lemma 2. *Let p be the norm of the clustering objective desired (i.e. $p = 2$ for k-means, $p = \infty$ for k-center or $p = 2$ for k-median). The clustering produced in each iteration is a constant approximation to any desired k-clustering objective with any constant norm $p \in (0, 2]$ or $p = \infty$.*

The proof of Lemma 2 is deferred to the full version of this paper. Combining Lemma 1 and Lemma 2 we obtain Theorem 1 as a corollary.

[9] By the generalized triangle inequality this is true for $p = 1, 2$ and it is true for $p = \infty$. So this is true for the cost of k-center, k-means and k-median.

4.2 Distributed Centroid-Linkage

As discussed in Sections 2 and 3, Algorithm 2 has a linear chain of dependencies. In this subsection we show how to modify step 4 of Algorithm 2 to overcome this difficulty.

The main intuition is to change Algorithm 2 to merge any pair of clusters A, B whose centroids are within distance $\alpha\delta$, where δ is the current smallest distance between cluster centroids and $\alpha \geq 1$ is a small constant. Our algorithm will find a collection of disjoint pairs of clusters which meet this condition. The algorithm then merges all such pairs and updates the minimum distance before repeating this procedure. This procedure is described in Algorithm 5.

```
1  CloseCentroidClustering($S, \alpha$)
2  Let $T$ be an empty tree
3  For each $x \in S$ add a leaf node corresponding to $\{x\}$ to $T$
4  Let $\mathcal{C}$ be the current set of clusters while $|\mathcal{C}| > 1$ do
5      $\delta \leftarrow \min_{A,B\in\mathcal{C}} d(\mu(A), \mu(B))$
6      for $X \in \mathcal{C}$ do
7          if $\exists Y \in \mathcal{C}$ such that $d(\mu(X), \mu(Y)) \leq \alpha\delta$ then
8              Add a node corresponding to $X \cup Y$ with children $X, Y$ to $T$
9              $\mathcal{C} \leftarrow \mathcal{C} \setminus \{X, Y\} \cup \{X \cup Y\}$
10         end
11     end
12 end
13 Return the resulting tree $T$
```

Algorithm 5: Idealized Close Centroid Clustering Algorithm

By definition, Algorithm 5 will be α-close to the centroid linkage algorithm. There are two issues that arise when bounding the algorithm's worst-case guarantees. First, it is not clear how to efficiently implement lines 5-10 in the distributed setting. We will address this issue in Section 4.3, where we describe the distributed implementations. Intuitively, we apply the popular locality-sensitive-hashing (LSH) technique, allowing us to perform these steps efficiently in the distributed setting.

The second issues is that it is difficult to bound the chain of dependencies for this algorithm since we cannot guarantee that the minimum distance δ increases by a constant factor after each iteration of the while loop.[10] Nevertheless, we find that this formulation of the algorithm works well empirically despite not having a formal bound on the round complexity. See Section 5 for these results.

To understand why this algorithm might have small round complexity in practice, we developed an algorithm with strong guarantees on its running time. See the full version of this paper for a proper description of this algorithm. For

[10] It is possible to construct worst-cases instances where the minimum distance δ can decrease between iterations of the while loop.

intuition, the algorithm maintains the following two invariants. First, if two clusters X, Y are merged during the algorithm, then the distance between their centroids is $O(\log^2(n)\delta)$, where δ is the current minimum distance between any two clusters. Second, at the end of the merging step the minimum distance between the centroids of the resulting clusters is at least $(1+\epsilon)\delta$, for some constant $\epsilon > 0$.[11] These two invariants taken together imply an $O(\log^2(n))$-close algorithm for centroid linkage with $O(\text{poly}(\log n))$ length dependency chains when the ratio of the maximum to the minimum distance in S is bounded by a polynomial in n.

To achieve these invariants our new algorithm carefully merges nodes in two stages. A first where the algorithm recursively merges subsets of points that are close at the beginning of the stage. Then a second where the algorithm merges the leftover points from different merges of the first stage. With this two stage approach, we can formally bound the dependency chains and the closeness of the resulting algorithm. The precise description and analysis of this algorithm is involved and is presented in the full version of this paper. The following theorem characterizes the main theoretical result of this section.

Theorem 2. *There exists an algorithm that is $O(\log^2(n))$-close to the sequential centroid linkage algorithm and it has $O(\text{poly}(\log n))$ length chains of dependencies.*

The main trade-off involved in this result is that in order to ensure a fast running time our algorithm must be willing to make some merges that are an $O(\log^2(n))$-factor worse than the best possible merge available at that time. This is due to considering a worst-case analysis of our algorithm. In practice, we find that the closeness of a variation of Algorithm 5 is much smaller than Theorem 2 would suggest while maintaining a fast running time. See Section 5.

4.3 From Bounded Length Dependency Chains to Parallel Algorithms

We now discuss how to adapt our algorithms to run on distributed systems. In particular we show that every iteration between consequent recursive calls of our algorithms can be implemented using a small number of rounds in the massively parallel model of computation and so we obtain that both algorithms can be simulated in a polylogarithmic number of rounds.

Parallelizing Divisive k-Clustering: We start by observing that there are previously known procedures [13, 8, 4, 6] to compute approximate k-clusterings in the massively parallel model of computation using only a constant number of rounds. Here we use these procedures as a black-box.

Next, the reassignment operation can be performed within a constant number of parallel rounds. Elements can be distributed across machines and the centers v_1 and v_2 can be sent to every machine. In a single round, every element computes the distance to v_1 and v_2 and in the next round the size of B_1, B_2 and U are

[11] In order to guarantee this second invariant,our algorithm must be allowed to make merges at distance $O(\log^2(n)\delta)$.

computed. Finally given the sizes of B_1, B_2 and U the reassignment can be computed in a single parallel round.

So steps 4, 5 and 6 of Algorithm 3 can be implemented in parallel using a constant number of parallel rounds. Furthermore, we established that the algorithm has at most logarithmic chain of dependencies. Thus we obtain the following theorem:

Theorem 3. *There exist $O(\log n)$-round distributed hierarchical clustering algorithms that are $O(1)$-close to bisecting k-means, bisecting k-center or bisecting k-median.*

Parallelizing Centroid-Linkage: Parallelizing our variant of centroid-linkage is more complicated. As stated before, the main challenge is to find an efficient way to implement lines 5-10 of Algorithm 5. The solution to this problem is the use of Locality Sensitive Hashing (LSH). For simplicity of exposition we focus on the Euclidian distances and we use the sketch from [11] for norm $p \in (0, 2]$, nevertheless we note that our techniques can be easily extended to any LSH-able distance function. We refer the interested reader to the full version of this paper for complete technical details. The following Theorem is restated from [11].

Theorem 4. *Fix a domain S of points a parameter $\delta > 0$ and constant parameter $\epsilon > 0$. There exists a class of hash functions $\mathcal{H} = \{h : S \to U\}$ and constants $p_1, p_2 > 0$ with $p_1 > p_2$ such that for any two points u and v in S if $d(u, v) \leq \delta$ then $Pr_{\mathcal{H}}[h(v) = h(u)] \geq p_1$ and if $d(u, v) \geq (1+\epsilon)\delta$ then $Pr_{\mathcal{H}}[h(v) = h(u)] \leq p_2$.*

Intuitively the LSH procedure allows us to group together points that are near each other. Using the previous theorem and the classic amplification technique for LSH presented in [21], it is possible to show the following theorem.

Theorem 5. *Fix a domain S of points, a parameter $\delta > 0$ and a small constant $\epsilon > 0$. Let S' be the set of points where there exists another point within distance δ and S'' be the set of points where there exists another point within distance $(1 + \epsilon)\delta$. With probability $1 - n^{-2}$ for each points $v \in S'$ there is a $O(1)$ round distributed procedure that can identify another point u such that $d(u, v) \leq (1+\epsilon)\delta$. Furthermore the same procedure identifies for some $v \in S''$ another point u such that $d(u, v) \leq (1 + \epsilon)\delta$.*

Using these tools, we now describe a parallelizable variant of Algorithm 5. See Algorithm 6 for the pseudocode. Intuitively, we apply LSH to the centroids of the current set of clusters in order to identify candidate merges that are α-close for centroid-linkage.

We note that LSH can also be applied to the theoretically efficient algorithm alluded to in Theorem 2. This allows us to get a theoretically efficient distributed algorithm that is close to centroid-linkage. The following theorem characterizes this result, however its proof is technical and deferred to the full version of this paper.

Theorem 6. *There exists an algorithm running in $O(\log^2(n) \log\log(n))$ distributed rounds and is $O(\log^2(n))$-close to the sequential centroid-linkage algorithm.*

```
1  FastCentroid($S, \alpha$)
2  Let $T$ be an empty tree
3  For each $x \in S$ add a leaf node corresponding to $\{x\}$ to $T$
4  Let $\mathcal{C}$ be the current set of clusters
5  while $|\mathcal{C}| > 1$ do
6      $\delta \leftarrow \min_{A,B \in \mathcal{C}} d(\mu(A), \mu(B))$
7      Use LSH with distance parameter $\alpha\delta$ on each point $\mu(X)$ for $X \in \mathcal{C}$
8      For each hash value $h$, let $\mathcal{C}_h$ denote all the clusters hashed to this value
9      For each $h$ place $\mathcal{C}_h$ on a single machine
10     Pair clusters that are within distance $\alpha\delta$ in $\mathcal{C}_h$ until all remaining clusters
         have no other cluster within distance $\alpha\delta$
11     Merge the paired clusters and add the corresponding nodes to the tree $T$
12     Update $\mathcal{C}$ appropriately to contain the new clusters but not the old clusters
         used in each merge.
13 end
14 Return the resulting tree $T$
```

Algorithm 6: Fast α-Close Centroid Clustering Algorithm

5 Experimental Results

In this section we empirically evaluate the algorithms in this paper. The algorithms will be referred to as *Div-k-clust.* (for the k-means algorithm) and *CentroidLink* (for the centroid algorithm). The sequential baseline algorithms are *kBase* and *cBase*. These are evaluated on three datasets from the UCI machine learning repository commonly used for clustering experimentation: `Shuttle`, `Covertype`, and `Skin` [29].

Parameters. Both of our algorithms are parameterized with an adjustable parameter. This is c in the divisive algorithm and α in the centroid algorithm. Both parameters were set to 4 in the experiments if not specified.

Evaluation Criteria. The algorithms are evaluated on their efficiency as well as the quality of the solution compared to the sequential algorithms. The closeness is a measure of quality; the number of rounds measures the efficiency. We also examine the effect of varying the parameter on the efficiency of the algorithm.

Quality Evaluation: Here we examine the closeness of the algorithm to their sequential counterparts.

CentroidLink For the *CentroidLink* algorithm the parameter α specifies the closeness. Recall that the sequential algorithm merges the pair of clusters whose centroids are closest to form a subtree; whereas, the distributed algorithm merges all pairs with distance at most an α factor greater than the smallest distance. The experimenter can freely choose how close the parallel algorithm will adhere to the sequential one with a tradeoff in the number of rounds. We are interested in the closeness of the algorithm's decisions compared to that of the sequential algorithm. We will show this by presenting the ratio of the distance between the pair the algorithm merges compared to the actual distance of the closest pair of nodes.

Div-k-clust. Recall that *Div-k-clust.* differs from *kBase* by having an extra step in which some points are reassigned before the recursion. This step can potentially cause *Div-k-clust.* to deviate from *kBase* by placing points in different subtrees than *kBase* would. The closeness should be a measure of the cost of this difference. We measure closeness as the ratio of the k-means cost before and after the reassignment.

On average, the closeness ratio of the algorithms are small constants for each data set. Tables 1(b) and 1(a) have a more detailed breakdown of the results. There, we break down the data on closeness by noting the size of the subtree the moment the algorithm makes the decision which might differ from the sequential algorithm. As there are many different sizes for subtrees, we have grouped the subtrees which are close to each other in size and averaged them, for example, subtrees of size 0-1000 are averaged together in the first row of the table. The dashes, '-', in the table indicate that there were no resultant subtrees of the corresponding size range. Note that the ratios are small in general for both algorithms.

Table 1. Evaluation of the Closeness

(a) Closeness of *Div-k-clust.* to *kBase*

Size	Shuttle	Skin	Covertype
≤1000	1.51	1.61	1.51
2000	1.69	1.74	1.58
3000	1.74	1.91	1.22
4000	1.57	2.10	1.74
5000	-	1.19	-
6000	-	2.30	-
8000	1.64	-	2.01
≥10000	1.74	1.84	1.07
Overall	1.52	1.61	1.51

(b) Closeness of *CentroidLink* to *cBase*

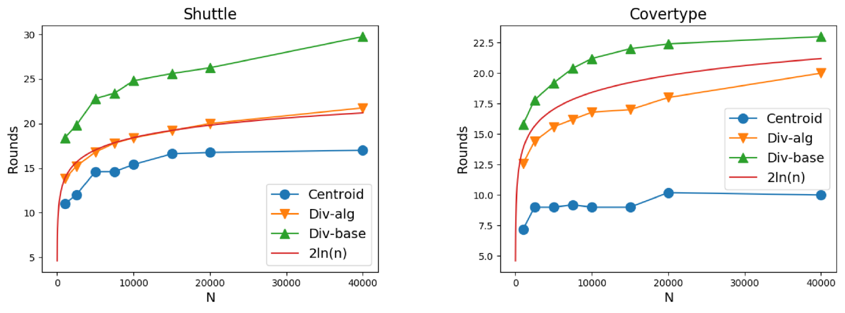
Size	Shuttle	Skin	Covertype
≤1000	2.74	2.66	2.38
2000	2.66	2.56	2.70
3000	2.76	2.25	2.72
4000	2.50	2.89	-
5000	-	3.16	1.81
6000	1.84	-	-
7000	2.48	3.40	2.11
8000	2.72	1.16	-
9000	-	-	1.92
≥10000	1	2.84	1
Overall	2.74	2.66	2.38

Efficiency Evaluation: Figure 2 plots the number of rounds used by each algorithm on each dataset. Data points are subsampled and averaged over five trials. We compare our algorithms against the baseline sequential algorithms. However, in theory, the centroid baseline is very sequential; the ith merge must depend on all $i-1$ previous merges. Therefore, it has a round complexity of $\Omega(n)$. For a more reasonable comparison, we have instead plotted the function $2\ln(n)$ for comparison as we expect our algorithms to scale logarithmically. The sequential algorithm is much worse than this.

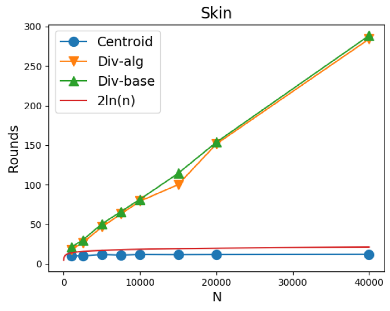
Both *Div-k-clust.* and *kBase* perform poorly on the Skin dataset. One explanation is that this 2-class dataset mostly contains data points of one class, therefore,

k-means clustering results in tall trees taking requiring rounds to compute. This is an example of a dataset in which the centroid algorithm may be preferred by a practitioner.

In general, the number of rounds are quite low, stable, and below the logarithmic function, especially for the centroid algorithm.



(a) Shuttle (b) Covertype



(c) Skin

Fig. 2. Evaluation of the Number of Rounds

Table 2 presents results from varying the parameter c or α on the number of rounds. These results were computed with $N = 10000$ on the `Shuttle` dataset. In the table, the closer the parameter is to 1, the better the algorithms simulates the sequential variant and this tradesoff with number of rounds.

6 Conclusion and Future Work

In this work we develop scalable hierarchical clustering algorithms that are close to the sequential bisecting k-clustering and centroid-linkage algorithms. The distributed algorithms run in a small number of rounds and give empirical results

c/α	*Div-k-clust.*	*CentroidLink*
1.5	21.8	13.6
2	19.1	7.6
4	18.5	7.4
8	17.3	5.8

Table 2. Effect of c/α on Rounds for Shuttle

that support our theory. An interesting open question is how to apply this paper's framework to other popular methods such as average-linkage or Ward's method.

References

1. Andoni, A., Nikolov, A., Onak, K., Yaroslavtsev, G.: Parallel algorithms for geometric graph problems. In: Symposium on Theory of Computing (STOC) 2014
2. Bader, D.A., Cong, G.: Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. J. Parallel Distrib. Comput. **66**(11), 1366–1378
3. Bahmani, B., Moseley, B., Vattani, A., Kumar, R., Vassilvitskii, S.: Scalable k-means++. PVLDB **5**(7), 622–633 (2012)
4. Balcan, M., Ehrlich, S., Liang, Y.: Distributed k-means and k-median clustering on general communication topologies. In: NIPS. pp. 1995–2003 (2013)
5. Bateni, M., Behnezhad, S., Derakhshan, M., Hajiaghayi, M., Lattanzi, S., Mirrokni, V.: On distributed hierarchical clustering. In: NIPS 2017 (2017)
6. Bateni, M., Bhaskara, A., Lattanzi, S., Mirrokni, V.S.: Distributed balanced clustering via mapping coresets. In: NIPS 2014 (2014)
7. Charikar, M., Chatziafratis, V.: Approximate hierarchical clustering via sparsest cut and spreading metrics. In: SODA 2017 (2017)
8. Charikar, M., Chekuri, C., Feder, T., Motwani, R.: Incremental clustering and dynamic information retrieval. SICOMP **33**(6), 1417–1440 (2004)
9. Charikar, M., Guha, S., Tardos, É., Shmoys, D.B.: A constant-factor approximation algorithm for the k-median problem. J. Comput. Syst. Sci. **65**(1), 129–149 (2002)
10. Dasgupta, S.: A cost function for similarity-based hierarchical clustering. In: STOC 2016. pp. 118–127 (2016)
11. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: SoCG 2004 (2004)
12. Eisen, M.B., Spellman, P.T., Brown, P.O., Botstein, D.: Cluster analysis and display of genome-wide expression patterns. Proceedings of the National Academy of Sciences **95**(25), 14863–14868 (1998)
13. Ene, A., Im, S., Moseley, B.: Fast clustering using MapReduce. In: KDD. pp. 681–689 (2011)
14. Gower, J.C., Ross, G.J.S.: Minimum spanning trees and single linkage cluster analysis. Journal of the Royal Statistical Society. Series C (Applied Statistics) **18**(1), 54–64 (1969)
15. Gower, J.C., Ross, G.J.S.: Parallel algorithms for hierarchical clustering. Parallel Computing **21**(8), 1313 – 1325 (1995)
16. Har-Peled, S., Mazumdar, S.: On coresets for k-means and k-median clustering. In: STOC 2004 (2004)

17. Hastie, T., Tibshirani, R., Friedman, J.: Unsupervised Learning, pp. 485–585. Springer New York, New York, NY (2009)
18. Heller, K.A., Ghahramani, Z.: Bayesian hierarchical clustering. In: ICML 2005. pp. 297–304 (2005)
19. Hochbaum, D.S., Shmoys, D.B.: A unified approach to approximation algorithms for bottleneck problems. J. ACM **33**(3), 533–550 (1986)
20. Im, S., Moseley, B., Sun, X.: Efficient massively parallel methods for dynamic programming. In: STOC 2017 (2017)
21. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: STOC 1998. pp. 604–613 (1998)
22. Jain, A.K.: Data clustering: 50 years beyond k-means. Pattern Recognition Letters **31**(8), 651 – 666 (2010)
23. Jin, C., Chen, Z., Hendrix, W., Agrawal, A., Choudhary, A.N.: Incremental, distributed single-linkage hierarchical clustering algorithm using mapreduce. In: HPC 2015. pp. 83–92 (2015)
24. Jin, C., Liu, R., Chen, Z., Hendrix, W., Agrawal, A., Choudhary, A.N.: A scalable hierarchical clustering algorithm using spark. In: BigDataService 2015. pp. 418–426 (2015)
25. Kanungo, T., Mount, D.M., Netanyahu, N.S., Piatko, C.D., Silverman, R., Wu, A.Y.: A local search approximation algorithm for k-means clustering. Comput. Geom. **28**(2-3), 89–112 (2004)
26. Karloff, H.J., Suri, S., Vassilvitskii, S.: A model of computation for mapreduce. In: SODA 2010. pp. 938–948 (2010)
27. Krishnamurthy, A., Balakrishnan, S., Xu, M., Singh, A.: Efficient active algorithms for hierarchical clustering. In: ICML 2012 (2012)
28. Lattanzi, S., Moseley, B., Suri, S., Vassilvitskii, S.: Filtering: a method for solving graph problems in mapreduce. In: SPAA 2011 (Co-located with FCRC 2011). pp. 85–94 (2011)
29. Lichman, M.: UCI machine learning repository (2013), http://archive.ics.uci.edu/ml
30. Mao, Q., Zheng, W., Wang, L., Cai, Y., Mai, V., Sun, Y.: Parallel hierarchical clustering in linearithmic time for large-scale sequence analysis. In: 2015 IEEE International Conference on Data Mining. pp. 310–319 (Nov 2015)
31. Murtagh, F., Contreras, P.: Algorithms for hierarchical clustering: an overview. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery **2**(1), 86–97 (2012)
32. Qin, L., Yu, J.X., Chang, L., Cheng, H., Zhang, C., Lin, X.: Scalable big graph processing in mapreduce. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. pp. 827–838. SIGMOD '14 (2014)
33. Rajasekaran, S.: Efficient parallel hierarchical clustering algorithms. IEEE Trans. Parallel Distrib. Syst. **16**(6), 497–502 (Jun 2005)
34. Roy, A., Pokutta, S.: Hierarchical clustering via spreading metrics. In: NIPS 2016. pp. 2316–2324 (2016)
35. Spark: https://spark.apache.org/docs/2.1.1/mllib-clustering.html (2014)
36. Steinbach, M., Karypis, G., Kumar, V.: A comparison of document clustering techniques. In: In KDD Workshop on Text Mining (2000)
37. Wang, J., Moseley, B.: Approximation bounds for hierarchical clustering: Average-linkage, bisecting k-means, and local search. In: NIPS (2017)
38. Yaroslavtsev, G., Vadapalli, A.: Massively parallel algorithms and hardness for single-linkage clustering under lp distances. In: ICML 2018. pp. 5596–5605 (2018)