Incremental, Distributed Single-Linkage Hierarchical Clustering Algorithm Using MapReduce

Chen Jin, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Alok Choudhary
Northwestern University, Evanston, IL
{chen.jin, zzc472, , whendrix, ankitag, choudhar}@eecs.northwestern.edu

Author Keywords
Hierarchical Clustering, Incremental Hierarchical Clustering, MapReduce

ABSTRACT

Single-linkage hierarchical clustering is one of the prominent and widely-used data mining techniques for its informative representation of clustering results. However, the parallelization of this algorithm is challenging as it exhibits inherent data dependency during the hierarchical tree construction. Moreover, in many modern applications, new data is continuously added into the already huge datasets. It would be impractical to reapply the clustering algorithm on the augmented datasets from scratch.

In this paper, we propose a unified algorithm which can not only cluster the large dataset, but also incorporate the newly arrived data incrementally. More specifically, we formulate the single-linkage hierarchical clustering problem as a Minimum Spanning Tree (MST) construction problem on a complete graph. The algorithm decomposes the complete graph into a set of non-overlapped subgraphs, computes the intermediate sub-MSTs for each subgraph in parallel, and merges all the sub-MSTs to achieve the final solution. In addition, the same framework can treat the incremental data insertion as a separate data subset and integrate it nicely with the existing solution. We implement the unified algorithm by employing MapReduce framework.

Using both synthetic and real-world datasets containing up to millions of high-dimensional points, we show that the proposed algorithm achieves a scalable speedup up to 200 on 300 computer cores for the base dataset and a speedup up to 120 for the dataset with maximum 5% random insertion.

INTRODUCTION

Hierarchical clustering is one of the prominent and widely-used data mining techniques for its informative representation of clustering results. It organizes the relationships of clusters using a tree diagram (dendrogram), giving the idea of how each data point is positioned related to the overall cluster structure. Compared to non-parameter free clustering algorithms like partitioning clustering, hierarchical clustering does not require the number of clusters in advance, and can deterministically assign the cluster label to each data point. Moreover, it offers insight into the complete hierarchy of clusters. As a representative implementation of hierarchical clustering algorithm, single-linkage method has been used in numerous applications such as document classification, computational biology, and image segmentation [19, 22, 23, 28]. For example, [5] uses dendrogram to visualize hierarchical clustering of tissues and genes; [28] employs hierarchical information to help visitors navigate the articles on the websites.

HPC 2015 April 12-15, 2015, Alexandria, VA.
Copyright © Society for Modeling & Simulation International (SCS)

In the face of the ever-growing datasets, the single-machine performance of hierarchical clustering algorithm can no longer keep up the game, which creates an urgent demand for a parallel solution. However, the parallelization of hierarchical clustering algorithm is a non-trivial task. First, hierarchical clustering algorithm itself is highly computationally expensive. And, during the hierarchical tree construction procedure, it exhibits inherent data dependency. Second, in many recent applications, data are often received incrementally and merged to already huge datasets, it would be impractical to reapply the clustering algorithm on the updated dataset from the scratch whenever new data arrives. For instance, the data warehouses collect new data and apply updates periodically in a batch mode at the off-peak time; Facebook has more than 100 million photos uploaded per day and 90 billion photos in total currently [1]. It is impractical or even infeasible to apply the clustering algorithm on the entire dataset. Thus, efficiently updating the clustering solution incrementally on the explosive size of the original dataset without applying the algorithm from scratch is challenging but critical.

To cope with these two challenges gracefully without designing two different frameworks, a unified algorithm is required to not only cluster the large datasets efficiently, but also incorporate the newly added data incrementally. In this paper, we propose IncDiSC, an Incremental, Distributed Single-linkage hierarchical Clustering algorithm using MapReduce framework. The idea of our algorithm is to formulate the single-linkage hierarchical clustering algorithm using graph algorithmic concepts, which is equivalent to solve a Minimum Spanning Tree of the complete graph induced by the dataset [17]. The algorithm initially divides the base dataset into a number of non-overlapped subsets. Therefore, the complete graph induced by the base dataset can be decomposed into a set of complete subgraphs induced by data subset respectively as well a number of complete bipartite subgraphs by pairs of data subsets. In this way, any edge in the original complete graph is assigned to only one subgraph and all subgraphs are independent from one another. Such a divide-and-conquer strategy allows us to solve sub-MSTs in parallel and merge them to achieve the final MST.

When there is a new dataset come in, IncDiSC forms the complete bipartite subgraphs between the new dataset and each

initial data subset, as well as the complete subgraph induced solely by the new dataset. In the same vein, the algorithm calculates sub-MSTs for all the subgraphs and merge them with the existing MST to find the MST for the augmented dataset. This algorithm can be applied periodically at the data warehouse with relatively low cost. The algorithm provides both incremental and scalable solution to handle large-scale dynamic datasets. It is memory-efficient and can be scaled out linearly. In the experiment section, we present a data-dependent characterization of hardness and evaluate algorithm's scalability with up to 1 million data points. The k-way merge process of intermediate solutions is configurable by setting user-defined partitioning function in MapReduce framework. The empirical results show our algorithm can achieve an estimated speedup up to 200 on 300 computer cores for the base dataset and a speedup up to 120 for the dataset with maximum 5% random insertion.

The main contributions of this paper are summarized as follows:

- We propose a scalable algorithm for single-linkage hierarchical clustering using MapReduce framework which also support an incremental update on the original solution;
- We introduce a configurable merge process achieving reasonable trade-off between the number of MapReduce rounds and the degree of parallelism;
- And empirical evaluation demonstrates the linear scalability and speedup on both synthetic and real-world data.

The remainder of paper is organized as follows. We brief the related work in Section II. In Section III, we describe our proposed incremental, distributed single-linkage hierarchical clustering algorithm, and we detail the system design using MapReduce framework. In Section IV, the algorithm's scalability is evaluated on both synthetic datasets and real-world datasets. Finally, we draw the conclusions in Section V.

RELATED WORK

The goal of this paper is to develop a new parallel, incremental hierarchical clustering algorithm using MapReduce. Thus, our work is related to parallelization of hierarchical clustering. Since our approach essentially reformulates the hierarchical clustering problem as finding a Minimum Spanning Tree in the complete connected graph induced by the dataset, we have investigated literature on solving MST problems using MapReduce. On the other hand, some researchers have proposed incremental variations for hierarchical clustering algorithms in the context of sequential implementation. In the following, we discuss most recent research results from these perspectives.

PARALLELIZATION OF HIERARCHICAL ALGORITHM: In order to overcome the inefficiency of the sequential hierarchical clustering algorithm on large-scale, high dimensional dataset, Hendrix *et al.* present SHRINK [17], a parallel single-linkage hierarchical clustering algorithm based on SLINK [3]. SHRINK exhibits good scaling and communication behavior, and only keeps space complexity in $\mathcal{O}(n)$ with n being the number of data points. The algorithm trades duplicated computation for the independence of the subproblem, and leads to a good speedup. However, this work only evaluates SHRINK on up to 36 shared memory cores, achieving a speedup of roughly 19.

As a powerful data processing tool, MapReduce is gaining significant momentum from both industry and academia. Some researchers have started to explore the possibility of implementing hierarchical clustering algorithm using MapReduce framework. For example, Wang and Dutta present PARABLE [29], a parallel hierarchical clustering algorithm using MapReduce. The algorithm is decomposed into two stages. In the first stage, the mappers randomly split the entire dataset into smaller partitions, on each of which the reducers perform the sequential hierarchical clustering algorithm. The intermediate dendrograms from all the small partitions are aligned and merged into a single dendrogram to suggest a final solution. However, this work does not provide the formal theoretical proof on the correctness of the dendrogram alignment algorithm. And the experiments only use 30 mappers and 30 reducers for the local clustering and a single reducer for the final global clustering. Hierarchical clustering has also been parallelized using other frameworks such as MPI [9]and GPU [10]. However, most of these methods explicitly compute and store the entire distance matrix containing all the pair-wise distance of the dataset.

SOLVING MST USING MAPREDUCE: Recently, Rastogi *et al.* [26] propose an efficient algorithm to find all the connected components in logarithmic number of MapReduce iterations for large-scale graphs. They present four different hashing schemes, among which Hash-to-Min proved to finish in $\mathcal{O}(\log n)$ iterations for path graphs and $\mathcal{O}(k(|V| + |E|))$ communication cost at round k. In the same paper, the algorithm is applied to single-linkage hierarchical clustering. It starts with each vertex and its neighbors as a starting connected component, all the components hashed to the same reducer are merged to a bigger component; a MST algorithm is then applied. However, a separate MapReduce job is required to determine the stop condition at each iteration, which might be acient. Besides, different from this work on general graphs, our work focuses on on complete connected graphs.

Lattanzi *et al.* [21] present Filtering, a novel method for solving large-scale dense graph problem in MapReduce. The authors present algorithms for MSTs as well as other fundamental graph problems with a constant number of MapReduce rounds even with machines having substantially sub-linear memory. However, the algorithms are mainly focused on meeting memory constraints rather than improving the scalability, and they only provide theoretical results.

SEQUENTIAL INCREMENTAL HIERARCHICAL CLUSTERING: Chen *et al.* [11] propose the incremental hierarchical clustering algorithm GRIN for numerical datasets. The algorithm is conducted in two phases in order to build the updated dendrogram. In the first phase, GRACE, a gravity-based agglomerative hierarchical clustering algorithm is applied to build a clustering dendrogram for the sets. Then the clustering dendrogram is reconstructed by flattering and pruning its bottom levels to generate a tentative dendrogram before

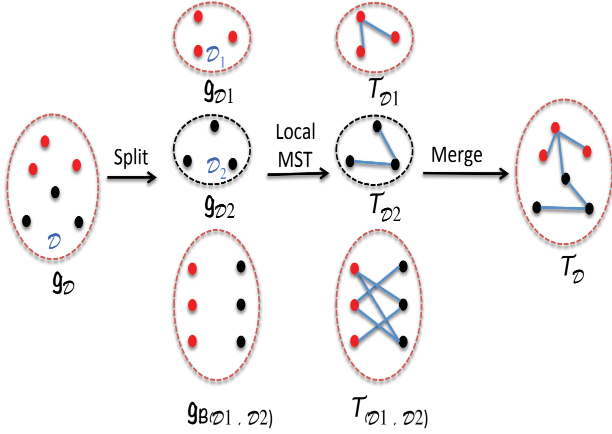


Figure 1: Illustration of parallel strategy on base dataset $\mathcal{D}$. We divide dataset $\mathcal{D}$ into two smaller parts, $\mathcal{D}_1$ and $\mathcal{D}_2$, calculate MSTs for $\mathcal{D}_1$, $\mathcal{D}_2$, and the complete bipartite bipartite graph between them, then merge the MSTs to find the MST for $\mathcal{D}$.

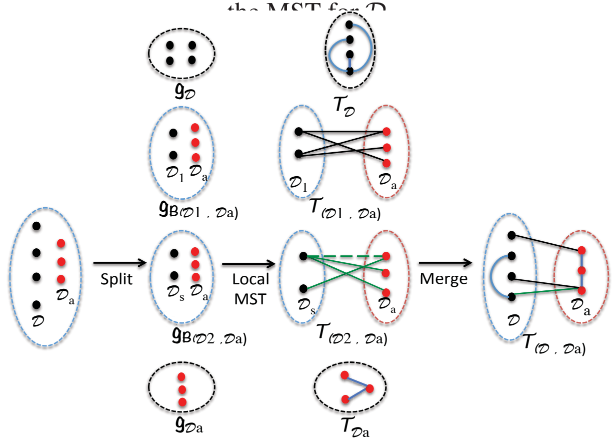


Figure 2: Illustration of parallel strategy on incremental data from $\mathcal{D}_a$. We divide dataset $\mathcal{D}$ into 2 smaller splits, $\mathcal{D}_1$ and $\mathcal{D}_2$, calculate the MSTs for $\mathcal{D}_a$ and the complete bipartite graphs between $\mathcal{D}_a$ and each split, then merge these intermediate MSTs along with the initial solution $\mathcal{T}_D$ to find the final MST for $\mathcal{D} \cup \mathcal{D}_a$.

adding new data points. In the second phase, GRIN examine each new data point to determine whether it belongs to leaf nodes of the tentative dendrogram. If it falls into exactly one leaf cluster, then it is labeled as that leaf cluster. Otherwise, the gravity theory is applied to determine which leaf cluster that the point belongs to. Some other techniques on incremental hierarchical clustering include [15, 27, 30].

INCREMENTAL, DISTRIBUTED SINGLE-LINKAGE HIERARCHICAL CLUSTERING USING MAPREDUCE

In this section, we describe our incremental, distributed algorithm for calculating single-linkage hierarchical clustering (SHC) dendrogram.

Hierarchical Clustering

First, we remind the reader about what the hierarchical clustering is. As an often used data mining technique, hierarchical clustering generally falls into two types: agglomerative and divisive. In the first type, each data point starts in its own singleton cluster, two closest clusters are merged at each iteration until all the data points belong to the same cluster. The divisive approach, however, works the process from top to bottom by performing splits recursively. As a typical example of agglomerative approach, Single-linkage hierarchical clustering merges the two clusters with the shortest distance, i.e. the link between the closest data pair (one in each cluster) at each step. Despite the fact that SHC can produce "chaining" effect where a sequence of close observations in different groups cause early merges of these groups, it is still a widely-used analysis tool to conduct early-stage knowledge discovery for its simplicity and quadratic time complexity.

Problem Decomposition

Our goal is to design a scalable and incremental algorithm such that it can not only scale to the large dataset but also accommodate the newly added data incrementally without carrying out clustering from scratch.

Based on the theoretical finding [17] that calculating the SHC dendrogram of a dataset is equivalent to finding the Minimum Spanning Tree (MST) of a complete weighted graph, where the vertices are the data points and the edge weight are the distance between any two points, the incremental SHC clustering problem with a base dataset $\mathcal{D}$ and newly added dataset $\mathcal{D}_a$ can be formulated as follows:

Given a complete weighted graph $\mathcal{G}(\mathcal{D} \cup \mathcal{D}_a)$ induced by the distances between points in $\mathcal{D} \cup \mathcal{D}_a$, design a parallel algorithm to find the MST in the complete weighted graph $\mathcal{G}(\mathcal{D})$ and incrementally solve the MST in $\mathcal{G}(\mathcal{D} \cup \mathcal{D}_a)$.

In this section, we describe how the clustering problem can be decomposed in these two settings. Intuitively, we propose to employ the multi-level paradigm via divide-and-conquer parallelization strategy. The multilevel paradigm is known for its effectiveness when solving very large-scale scientific problems including the top-down divisive clustering (e.g., PDDP [16]) or spectral graph partitioning techniques (e.g., [18]). The main idea of our algorithm is to divide the original problem into a set of non-overlapped subproblems, solve each subproblem and then merge the sub-solutions into an overall solution.

To show the process of problem decomposition, a toy example is illustrated in Figure 1. Given an original dataset $\mathcal{D}$, we first divided it into two disjoint subsets: $\mathcal{D}_1$ and $\mathcal{D}_2$, thus the complete graph $\mathcal{G}(D)$ is decomposed into to three subgraphs: $\mathcal{G}(\mathcal{D}_1)$, $\mathcal{G}(\mathcal{D}_2)$ and $\mathcal{G}_B(\mathcal{D}_1, \mathcal{D}_2)$, where $\mathcal{G}_B(\mathcal{D}_1, \mathcal{D}_2)$ is the complete bipartite graph on datasets $\mathcal{D}_1$ and $\mathcal{D}_2$. In this way, any possible edge is assigned to some subgraph, and taking the union of these subgraphs would return us the original graph. This approach can be easily extended to s splits, and leads to multiple subproblems of two different types: s complete subgraph on each split and C_s^2 complete bipartite subgraphs on each pair of splits. Once we complete the dividing procedure and form a set of subproblems, we distribute

these subproblems among multiple processes and apply a local MST algorithm on each of them, the calculated sub-MSTs are then combined to obtain the final solution for the original problem.

One of the main challenges in the design of modern clustering algorithms is that, in many applications, new datasets are continuously added into an already large dataset. As a result, it is impractical to reapply the clustering algorithm on the updated datasets. One way to tackle this challenge is to design a new parallel clustering algorithm that accommodate the update incrementally. Let's denote the newly arrived batch of as $\mathcal{D}_a$ and the original dataset as $\mathcal{D}$. By adding $\mathcal{D}_a$ into $\mathcal{D}$, new edges are formed for the complete bipartite on pair ($\mathcal{D}_a$, $\mathcal{D}$), and the complete subgraph within $\mathcal{D}_a$, therefore the complete graph on the updated dataset $\mathcal{G}(\mathcal{D} \cup \mathcal{D}_a)$ composes of three subgraphs: $\mathcal{G}_B(\mathcal{D}, \mathcal{D}_a)$, $\mathcal{G}(\mathcal{D}_a)$ and $\mathcal{G}(\mathcal{D})$. Since we already have the MST result on $\mathcal{G}(\mathcal{D})$, only MSTs on $\mathcal{G}_B(\mathcal{D}, \mathcal{D}_a)$ and $\mathcal{G}(\mathcal{D}_a)$ need to be calculated. However, the newly arrived data is usually much smaller than the original dataset, it is aciently to only use one process to calculate the MST on $\mathcal{G}_B(\mathcal{D}, \mathcal{D}_a)$. As shown in Figure 2, one approach is to divide the the original dataset into multiple splits and calculated the sub-MSTs on the complete bipartite subgraphs on each pair of $\mathcal{D}_a$ and the split and the complete subgraph induced on $\mathcal{D}_a$.

In Figure 1 and 2, we observe that except for the problem decomposition, both cases can be solved by the same local MST algorithm followed by a merge procedure, which leads to the final solution. Therefore, a generalized framework can be designed to handle large datasets with incremental updates.

IncDiSC Algorithm Design

In this section, we present IncDiSC, an incremental, distributed single-linkage hierarchical clustering algorithm.

Algorithm Design

Following the dividing steps described in step 1-8 of Algorithm 1, we break the original problem into multiple much smaller subproblems, a serial MST algorithm can be applied locally on each of them. For a weighted graph, there are three frequently used MST algorithms, namely Borůvka's, Kruskal's and Prim's [7, 20, 25]. Borůvka's algorithm was published back in 1920s. At each iteration, it identifies the cheapest edge incident to each vertex, and then forms the contracted graph which reduces the number of vertices by at least half. Thus, the algorithm takes $O(E \log V)$ time, where E is the number of edges and V is the number of vertices. Kruskal's algorithm initially creates a forest with each vertex as a separate tree, and iteratively selects the cheapest edge that doesn't create a cycle from the unused edge set to merge two trees at a time until all vertices belongs to a single tree. Both of these two algorithms require all the edge weights available in order to select the cheapest edge either for every vertex in the entire graph at each iteration. By contrast, Prim's algorithm starts with an arbitrary vertex as a MST root and then grows one vertex at a time until it spans all the vertices in the graph. At each iteration, it only needs one vertex's local information to proceed. Moreover, given a complete weighted graph, Prim's algorithm only takes $O(V^2)$ time and $O(V)$ space complexity, lending itself a good choice for the local MST algorithm.

Algorithm 1 IncDiSC, an incremental, distributed SHC algorithm

INPUT: a base dataset $\mathcal{D}$, a newly added dataset $\mathcal{D}_a$, a minimum spanning Tree $\mathcal{T}$ induced on $\mathcal{D}$, a merging parameter K
OUTPUT: a MST $\mathcal{T}'$ for $\mathcal{D} \cup \mathcal{D}_a$

```
1: if T == ∅ then
2:     Divide D into s roughly equal-sized splits,
       D_1, D_1, ..., D_s
3:     Form C_s^2 complete bipartite subgraphs and s complete
   subgraphs
4: else
5:     Divide D into t roughly equal-sized splits,
       D_1, D_1, ..., D_t
6:     Form t complete bipartite subgraphs and a complete
       subgraph
7: end if
8: Use Prim's algorithm to compute the sub-MST on
   each subgraph
9: repeat
10:    Taking the sub-MSTs and T, merge every K of
       them using the idea of Kruskal's algorithm
11: until one MST remains
12: return the final MST T'
```

As mentioned earlier, we have two types of subproblems: complete weighted graph and complete bipartite graph. For the first type of subproblem, we start with the first vertex v_0 in the vertex list just for convenience. While we populate all its edge weights by calculating distance from v_0 to every other vertex, we track the cheapest edge and emit the corresponding edge to the reducer in MapReduce framework (in this way, we don't need to store the MST explicitly). v_0 is then removed from the vertex list and the other endpoint of the emitted edge is selected to be the next starting vertex. This process is repeated until all the vertices are added to the tree. Thus, our algorithm maintains quadratic time complexity and linear space complexity.

The other type of subproblem is the complete bipartite subgraph between two disjoint data splits, denoted as the left and right split. Different from the complete subgraph case, we need to maintain an edge weight array for each split respectively. To start, we select the first vertex v_0 in the left split, populates an edge weight array from v_0 to every vertex in the right split, record the cheapest edge (v_0, v_t). In the next iteration, we populate another edge weight array from v_t to every vertex in the left split except for v_0. Then the cheapest edge is selected from both edge weight arrays. The endpoint of the cheapest edge (which is neither v_0 nor v_t) is selected as the next starting vertex, and the same process can be iterated until the tree spans all the vertices. The procedure takes $O(mn)$ time complexity and $O(m+n)$ space complexity, where m, n are the sizes of the two disjoint sets.

Step 10 in Algorithm 1 then iteratively merges all the intermediate sub-MSTs and the precalculated $\mathcal{T}$ to obtain the over-

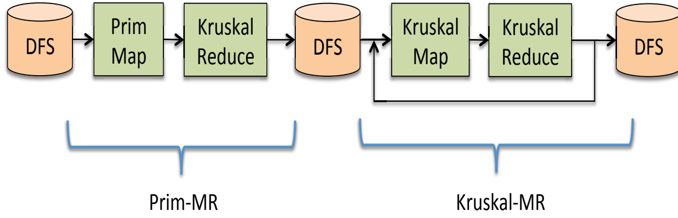


Figure 3: IncDiSC framework using MapReduce.

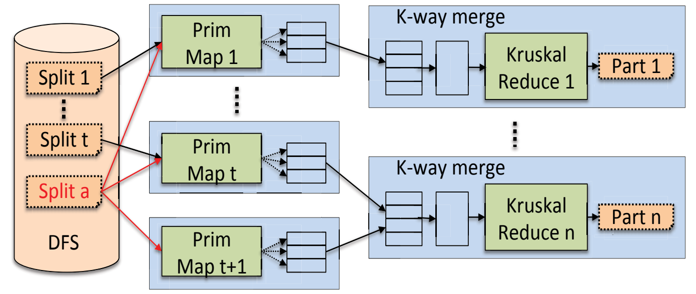


Figure 4: Prim-MR Diagram: Prim-MR is the first MapReduce job in IncDiSC algorithm. For incremental data insertion case, it consists of $t + 1$ Prim maps and n Kruskal reducers, where $n = \lceil (t + 1)/\mathrm{K} \rceil$.

all solution. In extreme case, all the sub-MSTs and $\mathcal{T}$ can be combined all at once using one process, however, this incurs huge communication contention and computational load; rather, we extend the merge procedure into multiple iterations by introducing configurable parameter K such that every K intermediate MSTs are merged at each iteration and it terminates when only one MST remains.

In order to efficiently combine these partial MSTs, we use union-find (disjoint set) data structure to keep track of the component to which each vertex belongs [12]. Recall the way we form subgraphs, most neighboring subgraphs share half of the data points. Every K consecutive subgraphs more likely have a fairly large portion of overlapping vertices. Thus, by combining every K sub-MSTs, we can detect and eliminate incorrect edges at an early stage, and reduce the overall communication cost for the algorithm. The communication cost can be further optimized by choosing the right K value with respect to the size of dataset, which we will discuss in the next section.

IncDiSC bcpReduce

MapReduce has emerged as one of the most frequently used parallel programming model for processing large-scale datasets since it was first proposed in 2004 by Dean and Ghemawat [14]. Its open source implementation, Hadoop, has become the de facto standard for both industry and academia. In the following, we describe in details the implementation of our proposed algorithm using Hadoop's MapReduce Framework.

A MapReduce job comprises three consecutive phases: map, shuffle and reduce. The input, output as well as intermediate data, is formated in $(key, value)$ pairs. In the map phase the input is processed one tuple at a time. All $(key, value)$ pairs emitted by the map phase which have the same key are then aggregated by the MapReduce system during the shuffle phase and sent to the reducer. At the reduce phase, each key, along with all the values associated with it, are processed together. In order for all the values with the same key end up on the same reducer, a partitioning or hash function need to be provided for the shuffle phase. The system then makes sure that all of the $(key, value)$ pairs with the same key are collected on the same reducer.

Figure 3 provides an overall MapReduce diagram for Algorithm 1. The algorithm can be implemented by two types of MapReduce jobs: Prim-MR and Kruskal-MR. The Prim-MR job is executed only once while the Kruskal-MR job can be repeated for multiple rounds until the final MST is completed. The aggregated results after a MR job is stored on the distributed file system, e.g. Hadoop distributed file system, as the input for the next available MR job. Prim-MR consists of PrimMap and KruskalReduce. Its detailed implementation is illustrated in Figure 4. The setup in the diagram also can handle the incremental case where the new arrived data split $\mathcal{D}_a$ is inserted. The original dataset is divided into t smaller splits. Each split is stored in the built-in SequenceFile format provided by Hadoop. SequenceFile is a flat binary record file with each record being a key-value pair. It is extensively used as Hadoop's input/output formats. In our case, the input data splits are keyed on the data point's id and valued on the corresponding feature vector.

In order for a mapper to know which two splits to be read, we initially produce $t + 1$ input files, each of which contains a single integer value gid between 0 and t to represent the subgraph id. Without loss of generality, the subgraph $(t + 1)$ is a complete subgraph while the others are complete bipartite subgraphs. Given a certain graph type, we apply the corresponding Prim's algorithm accordingly. For the case where there is no precalculated MST available for the original dataset, the decomposition in Figure 1 is adopted in the Prim-MR framework.

Each complete bipartite subgraph is assigned to a mapper. However, the complete subgraph only has half as many vertices as the complete bipartite subgraph. In order to keep the load balance, we assign two complete subgraphs to one mapper. On each mapper, a local sequential MST algorithm is implemented to calculate the MST on subgraph gid. As described in Algorithm 2, PrimMap's input is keyed on gid and valued on data point. The algorithm first populate two arrays S_1 and S_2. S_2 may be empty if the number of complete subgraphs is odd. Once both arrays get populated, we start to calculate the MST. As described previously, given a complete graph, the local MST algorithm starts with a single-node tree, and then augments the tree one vertex at a time by greedily selecting the cheapest edge among all the edges we have calculated so far. Instead of releasing all the edges after the MST is complete, the edges can be emitted one by one as they are generated, reducing I/O and network contention.

Algorithm 2 PrimMap: calculate the MST for a complete bipartite subgraph or a complete subgraph

```
1:  map(Int gid, Point value)
2:    Point[] S1, S2;
3:    cnts = 0;
4:    if (cnts < |S1|) do
5:      S1.append(value)
6:      cnt++; return;
7:    else if (cnts < (|S1| + |S2|)) do
8:      S2.append(value)
9:      cnt++; return;
10:   end
11:   if (isBiPartite) do
12:     PrimBipartite mst = new PrimBipartite(S1, S2)
13:     mst.emitMST()
14:   else do
15:     PrimComplete mst = new PrimComplete(S1)
16:     mst.emitMST()
17:     if (S2 is not NULL) do
18:       mst = new PrimComplete(S2)
19:       mst.emitMST()
20:     end
21:   end
```

As PrimMap emits the edge one at a time, the output is spilled into multiple temporary files on the local machine in a sorted order, and transferred to the designated reducer based on the partitioner. Before passing to the reducer, the files are concatenated in the sorted order and merged into a single input file. This is called data shuffle or sort-and-merge stage. Hadoop sorts keys by default and provides a secondary sorting mechanism, with which we can design a composite key containing graph id and the edge weight. In this way, K-way can be implicitly implemented in Hadoop under the hood without actual realization in the KruskalReduce.

In Kruskal-MR job, the map function is essentially an identity map which just passes through (key, value) pairs as they are. The reduce function reuses KruskalReduce to combine any intermediate MSTs and precalculate $\mathcal{T}$. The Kruskal-MR job is iterated until one MST remains.

EXPERIMENTAL RESULTS

Here, we discuss the experimental results on the implementation of IncDiSC algorithm using both synthetic and real-word datasets. We evaluate the scalability of the algorithm on the dataset with various size and dimensions as well as the data shuffle and I/O patterns in each MapReduce round.

We conduct the experiments on Jesup [2], a Hadoop cluster at NERSC. Jesup has 80 compute nodes each of which is quad-core Intel Xeon X5550 "Nehalem" 2.67 GHz processors (eight cores per node) with 24 GB of memory per node. All the nodes are interconnected by 4X QDR InfiniBand technology that provides 32 Gb/s of point-to-point bandwidth for data communication and I/O. However, computer nodes in Jesup have no local disks, which implies no data locality can be leveraged from Hadoop distributed file system. Despite this downside caused by the system specifics, our experiments still show the real computation and communication behaviors as in many other clusters.

Table 1: Structural properties of the synthetic-cluster, synthetic-random, and millennium-run-simulation testbed. The data size is measured in SequenceFile format which is compressed by GZIP.

Name	Points	Dimensions	Size (**MByte**)
clust20k	20k	5, 10	1.3, 2.1
clust100k	100k	5, 10	6.6, 10
clust500k	500k	5, 10	32, 49
rand20k	20k	5, 10	1.3, 2.1
rand100k	100k	5, 10	6.6, 10
rand500k	500k	5, 10	32, 49
DGalaxiesBower2006a (db)	1m	3	51
MPAHaloTreesMhalo (mm)	1m	3	51
MPAGalaxiesBertone2007 (mb)	1m	3	51
MPAGalaxiesDeLucia2006a (md)	1m	3	51

Datasets

We evaluate IncDiSC using sixteen datasets, which are divided into three categories: *synthetic-cluster*, *synthetic-random* and *millennium-run-simulation*. The first two categories are synthesized by using the IBM synthetic data generator [4,24]. With three different numbers of data points and two different dimensions, we generate six *synthetic-cluster* datasets, in which a random number of centroids are selected first and data points are added randomly to these centroids. And in six *synthetic-random* datasets, points in each dataset are uniformly distributed.

The third category *millennium-run-simulation* consists of four real-world datasets, MPAGalaxiesBertone2007 (mb) [6], MPAGalaxiesDeLucia2006a (md) [13], DGalaxiesBower2006a (db) [8], and MPAHaloTreesMhalo (mm) [6] which are taken from the Galaxy and Halo databases (as the name specified). Because we only take the first 3 dimensions (particle coordinates) from each dataset, we have randomly selected **1 million** points from these datasets. Our testbed contains up to **1 million** data points and each data point is a vector of up to **10** dimensions. Table 1 shows structural properties of the dataset.

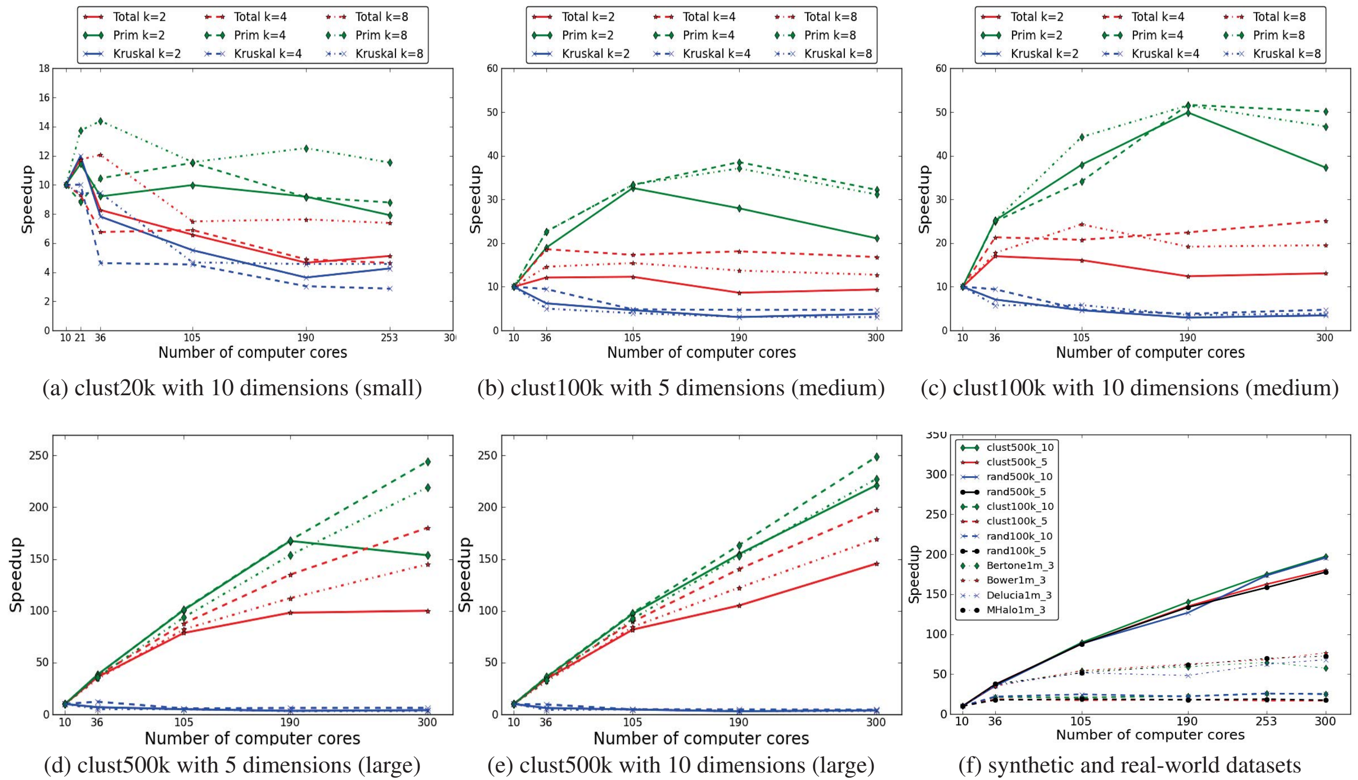
Performance

Scalability

We first evaluate how well our algorithm scales from 10 to 300 computer cores on twelve synthetic datasets with different K values, where K is the number of subgraphs that can be merged at one reducer.

The speedup on p cores is defined as $S = p_0 \frac{t_{p_0}}{t_p}$, with p_0 being the minimum computer cores that we conduct our experiments with, and t_p being the DiSC's execution time on p cores. Figure 5 (a) presents the speedup result for *clust20k* with 10 dimensions. "Total" measures the algorithm's entire execution time. It comprises "Prim" measuring the runtime for Prim-MR job, and "Kruskal" measuring the runtime for a series of Kruskal-MR jobs until the algorithm completes.

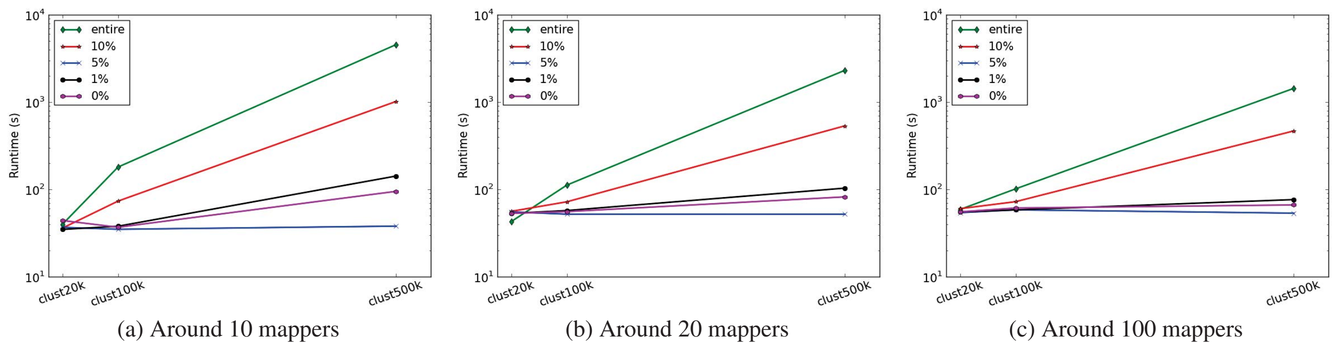
The performance has no improvement or even turns worse as we scale out for *clust20k_10*. This makes sense since small



(a) clust20k with 10 dimensions (small) (b) clust100k with 5 dimensions (medium) (c) clust100k with 10 dimensions (medium)

(d) clust500k with 5 dimensions (large) (e) clust500k with 10 dimensions (large) (f) synthetic and real-world datasets

Figure 5: Speedup on the synthetic datasets using 10-300 computer cores.

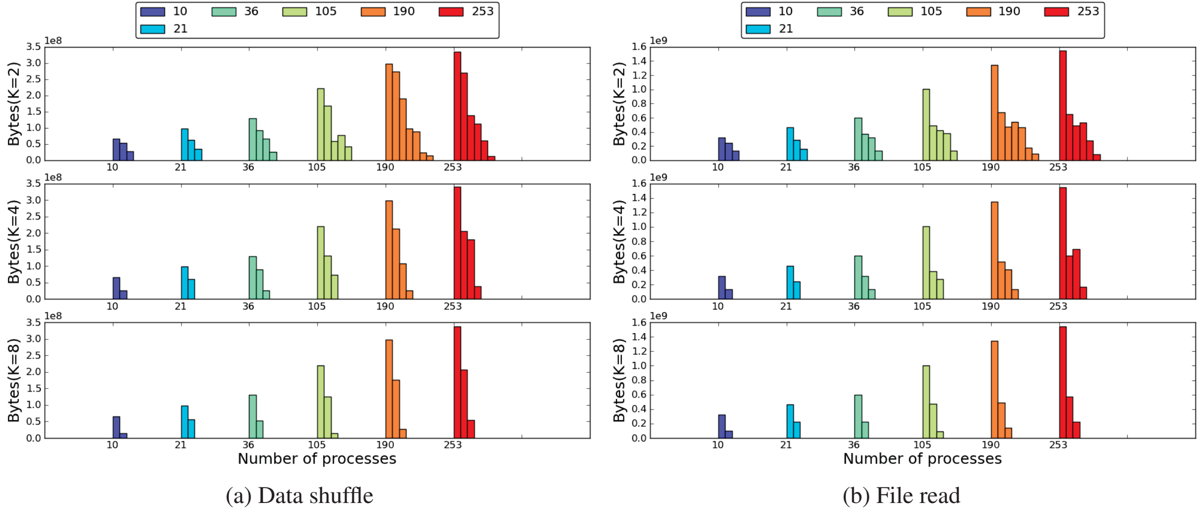


(a) Around 10 mappers (b) Around 20 mappers (c) Around 100 mappers

Figure 6: Runtime of DiSC vs. runtime of IncDiSC.

datasets can fit in one or a few machines' memory, the overhead introduced by unnecessary multiple MapReduce iterations would offset the computational gain from data parallelism.

For both medium and large datasets, Figure 5 (b) – (e) demonstrates a nice linear speedup until the number of compute cores increases beyond a certain number. For the medium-sized dataset, Figure 5 (c) shows that even with high dimensionality, the speedup starts to drop regardless of the number of dimensions, when the number of processes is beyond 190, which corresponds to 20 splits on the original data, each with $5k$ data points. However, with $K = 2$, we approximately need 7 MR iterations for the entire algorithm, thus the communication cost is no longer negligible. As illustrated in Figure 5 (e) the linear speedup sustains up to 300 compute cores for large datasets with high dimensionality. Among all four plots, $K = 4$ consistently outperforms $K = 2$ or 8. This is because K not only affects the number of MapReduce rounds, but also the number of reducers at the merge phase at each round. Increasing K value leads to a smaller number of iterations and smaller degree of parallelism. The value of



(a) Data shuffle

(b) File read

Figure 7: Data Pattern for IncDiSC algorithm: the amount of data bytes in the stages of data shuffle and file read. (File write is omitted here and it exhibits the same trend as file read.)

4 seems to be a sweet spot achieving a reasonable trade-off between these two effects.

Figure 5 (f) summarizes the speedup results on both synthetic and real-world datasets. As expected, the number of objects in the dataset significantly influences the speedups (bigger datasets show better scalability), and the dimensionality is another factor that affects the performance. The type of dataset hardly makes any difference in our algorithm as we use Euclidean distance as the edge weight measure, and the distribution of data points has no impact on the computational complexity in calculating distances.

Incremental Insertion

We evaluate the efficiency of IncDISC using *synthetic-cluster*'s three datasets with 10 dimensions. On each synthetic dataset, we perform $0.5\%, 0.1\%, 5\%$, and 10% random insertions and compare the runtime of IncDiSC algorithm on the entire updated dataset from the scratch with the average runtime of IncDiSC solely on the random updates. The results with different granularity of problem decomposition are depicted in Figure 6. We first divide the original dataset into 10 splits, each pairing with the insertion split to form 10 complete bipartite subgraphs in addition to a complete subgraph on the insertion alone. To be consistent with the case in which we reapply IncDiSC on the entire updated dataset, we accordingly partition the accumulated dataset into 5 splits which leads to 10 complete bipartite subgraphs and 5 complete subgraphs. Given this setup, we conduct experiments with 10 mappers as shown in Figure 6, and observe the speedup factors of about 1, 5, 120 on datasets of $20k, 100k, 500k$ data points respectively for small insertions up to 5%, and the speedup degrades to 5 when the insertion is 10% of the original data. Such a large percentage of addition usually would not appear in the real-world because the base dataset is already very large while the incremental update is batched and processed over small time interval. A similar speedup observation can be made when evaluating performance on finer decomposition on 20 and 100 mappers with about 45X and 27X speedup for 5% insertion respectively. The speedup factor doesn't increase as the number of mappers increases. A possible reason is that each complete bipartite subgraph is already fairly small when we split the original dataset into 5 partitions. Making finer decomposition would not buy us more data parallelism, it leads to more Kruskal-MR iterations which comes with MapReduce framework overhead, such as job setup, tear-down, data shuffle, etc. For large-scale and high-dimensional datasets, IncDiSC saves the runtime by two orders of magnitude over the naive implementation that works on the entire updated dataset without incorporating the priori clustering results.

I/O and Data Shuffle

In this section, we evaluate the data patterns with respect to MapReduce metrics, including file read, file write and data shuffle from mapper to reducer per iteration. Recall when we form the complete bipartite subgraphs, each split need be paired with every other split. Therefore, the amount of data read from the disk is linear to the number of splits, which is approximate the square root of the number of mappers. In Figure 7, each bar represents a MapReduce round, and bars in the same color represent a series of MR rounds that IncDiSC algorithm requires to find the MST given a certain number of computer cores. For example, the first bar represents Prim-MR job in IncDiSC algorithm. Figure 7 (a) illustrates the increasing trend of the amount of data shuffle from mapper to reducer. Notably, as we scale up the number of processes, the number of MapReduce rounds increases. However, the data is dramatically reduced after the first Prim-MR job by almost 2 orders of magnitude, which verifies our claim that incorrect edges are pruned at a very early stage. The same trend is observed for file read at mapper's input and file write at reducer's output. After the first iteration, the amount of

data shuffle and I/O is proportional to the number of vertices residing in the merged subgraphs, and the amount of vertices decreases by approximately K times due to the deduplication effect at the KruskalReduce's merging process. Figure 7 reveals that the number of iterations decreases with large K, so does the amount of the data. This finding also corresponds with speedup chart that 4-way merge outperforms 2- and 8-way merges because it provides a good trade-off between the number of MR rounds and the degree of parallelism per round.

CONCLUSION

In this paper, we present IncDiSC, an incremental, distributed algorithm for single-linkage hierarchical clustering algorithm to overcome the data dependency and algorithm complexity challenges in a unified framework. IncDiSC can not only scale to the large dataset, but also incorporate the incremental accumulation of the new data. We evaluated IncDiSC empirically using both synthetic and real-world datasets, and observed that it achieves a speedup up to 200 on 300 computer cores and two orders of magnitude speedup for up to 5% the insertion update. Experimental results have shown that IncDiSC reduces the amount of data shuttle dramatically at early iterations. Future work on IncDiSC may involve efforts to tackle the deletion update such that the algorithm can be adapted to much wide modern applications.

Acknowledgment

This work is supported in part by the following grants: NSF awards CCF-1029166, ACI-1144061, IIS-1343639, and CCF-1409601; DOE award DESC0007456; AFOSR award FA9550-12-1-0458; NIST award 70NANB14H012.

REFERENCES

1. `http://www.facebook.com/blog/blog.php?topic_id=185341929641`. [Online].
2. `http://www.nersc.gov/users/computational-systems/testbeds/jesup`. [Online].
3. SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 1 (1973).
4. Agrawal, R., and Srikant, R. Quest Synthetic Data Generator. *IBM Almaden Research Center* (1994).
5. Alizadeh, A., Eisen, M., Davis, R., Ma, C., Lossos, I., Rosenwald, A., Boldrick, J., Sabet, H., Tran, T., Yu, X., Ji, P., Yang, L., Ge, M., Moore, T., Hudson, J., Lu, L., Tibshirani, R., Sherlock, G., Chan, W., Greiner, T., Weisenburger, D., Armitage, J., Warnke, R., Levy, R., Wilson, W., Grever, M., Byrd, J., Botstein, D., Brown, P., and Staudt, L. Distinct types of diffuse large B-cell lymphoma identified by gene expression profiling. *Nature* (2000).
6. Bertone, S., De Lucia, G., and Thomas, P. The recycling of gas and metals in galaxy formation: predictions of a dynamical feedback model. *Monthly Notices of the Royal Astronomical Society 379*, 3 (2007), 1143–1154.
7. Boruvka, O. O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary). *Práce Mor. Prírodoved. Spol. v Brne III 3* (1926).
8. Bower, R., Benson, A., Malbon, R., Helly, J., Frenk, C., Baugh, C., Cole, S., and Lacey, C. Breaking the hierarchy of galaxy formation. *Monthly Notices of the Royal Astronomical Society 370*, 2 (2006), 645–655.
9. Cathey, R. J., Jensen, E. C., Beitzel, S. M., Frieder, O., and Grossman, D. Exploiting parallelism to support scalable hierarchical clustering. *Journal of the American Society for Information Science and Technology 58*, 8 (2007), 1207–1221.
10. Chang, D.-J., Kantardzic, M. M., and Ouyang, M. Hierarchical clustering with CUDA/GPU. In *ISCA PDCCS*, J. H. Graham and A. Skjellum, Eds., ISCA (2009), 7–12.
11. Chen, C., Hwang, S.-C., and Oyang, Y.-J. An incremental hierarchical data clustering algorithm based on gravity theory. In *Proceedings of the 6th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, PAKDD (London, UK, 2002), 237–250.
12. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.
13. De Lucia, G., and Blaizot, J. The hierarchical formation of the brightest cluster galaxies. *Monthly Notices of the Royal Astronomical Society 375*, 1 (2007), 2–14.
14. Dean, J., and Ghemawat, S. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, USENIX Association (Berkeley, CA, USA, 2004), 10–10.
15. Gurrutxaga, I., Arbelaitz, O., Martín, J. I., Muguerza, J., Pérez, J. M., and Perona, I. n. SIHC: A Stable Incremental Hierarchical Clustering Algorithm. In *ICEIS (2)'09* (2009), 300–304.
16. Heisele, B., Serre, T., Mukherjee, S., and Poggio, T. Feature reduction and hierarchy of classifiers for fast object detection in video images. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 2* (2001), 18.
17. Hendrix, W., Patwary, M. M. A., Agrawal, A., keng Liao, W., and Choudhary, A. Parallel hierarchical clustering on shared memory platforms. In *HiPC* (2012), 1–9.
18. Hofmeyr, S. A., Forrest, S., and Somayaji, A. Intrusion detection using sequences of system calls. *Journal of Computer Security 6* (1998), 151–180.
19. Jain, A. K., Murty, M. N., and Flynn, P. J. Data clustering: A review, 1999.

20. Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society 7*, 1 (Feb. 1956), 48–50.

21. Lattanzi, S., Moseley, B., Suri, S., and Vassilvitskii, S. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA (2011), 85–94.

22. Liu, L., Rui, Y., Sun, L., Yang, B., Zhang, J., and Yang, S.-Q. Topic mining on web-shared videos. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)* (2008), 2145–2148.

23. Madeira, S. C., and Oliveira, A. L. Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM Trans. Comput. Biol. Bioinformatics 1*, 1 (Jan. 2004), 24–45.

24. Pisharath, J., Liu, Y., keng Liao, W., Memik, G., Choudhary, A., and Dubey, P. NU-MineBench 3.0.

25. Prim, R. C. Shortest connection networks and some generalizations. *Bell System Technology Journal* (1957).

26. Rastogi, V., Machanavajjhala, A., Chitnis, L., and Sarma, A. D. Finding connected components on map-reduce in logarithmic rounds. *CoRR abs/1203.5387* (2012).

27. Sahoo, N., Callan, J., Krishnan, R., Duncan, G., and Padman, R. Incremental hierarchical clustering of text documents. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM ’06 (New York, NY, USA, 2006), 357–366.

28. Surdeanu, M., Turmo, J., and Ageno, A. A hybrid unsupervised approach for document clustering. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (2005), 685–690.

29. Wang, S., and Dutta, H. Parable: A parallel random-partition based hierarchical clustering algorithm for the mapreduce framework. *Technical Report, CCLS-11-04* (2011).

30. Widyantoro, D. H., Ioerger, T. R., and Yen, J. An incremental approach to building a cluster hierarchy. In *IEEE International Conference on Data Mining* (2002), 705–708.