2015 IEEE First International Conference on Big Data Computing Service and Applications

A Scalable Hierarchical Clustering Algorithm Using Spark

Chen Jin, Ruoqian Liu, William Hendrix
Ankit Agrawal and Alok Choudhary
Northwestern University, Evanston, Illinois, 60208

Zhengzhang Chen
NEC Laboratories America, INC., Princeton, NJ, 08540

***Abstract*—Clustering is often an essential first step in data mining intended to reduce redundancy, or define data categories. Hierarchical clustering, a widely used clustering technique, can offer a richer representation by suggesting the potential group structures. However, parallelization of such an algorithm is challenging as it exhibits inherent data dependency during the hierarchical tree construction. In this paper, we design a parallel implementation of Single-linkage Hierarchical Clustering by formulating it as a Minimum Spanning Tree problem. We further show that Spark is a natural fit for the parallelization of single-linkage clustering algorithm due to its natural expression of iterative process. Our algorithm can be deployed easily in Amazon's cloud environment. And a thorough performance evaluation in Amazon's EC2 verifies that the scalability of our algorithm sustains when the datasets scale up.**

I. Introduction

The data, features extracted from the data, and the model are the three major components of a machine learning solution. Over the last decade, it has shown that in real-world settings, the size of the dataset is the most important factor. A large body of literature has repeatedly shown that simple models trained over enormous quantities of data outperform more sophisticated models trained on less data [1]–[3]. This has led to the growing dominance of simple, data-driven solutions.

As one of simple and widely adopted machine learning models, clustering is often an essential first step in data mining to retrieve data's membership. Many applications face the demand to perform this operation over large amount of data. For example, Internet companies collect massive amount of data such as content produced by web crawlers or service logs. Frequently, it needs apply similarity queries to gain valuable understanding of the usage of their services. The understanding may include identifying customers with similar buying patterns, suggesting recommendations, performing correlation analysis, etc. Processing the ever-growing volume of data in a highly scalable and distributed fashion constitutes an answer to such a demand.

Fortunately, frameworks such as MapReduce [4] and Spark [5] can quickly process massive datasets by splitting them into independent chunks that are processed in parallel. Users are alleviated away from the system-level's programming details and only need to cope with the user-level APIs. However, clustering algorithms are generally difficult to parallelize effectively due to high data dependence. Relatively little work has been done. In this paper, we present SHAS, a Single-linkage Hierarchical clustering Algorithm using Spark framework. The key idea is to reduce the single-linkage hierarchical clustering problem to the minimum spanning tree (MST) problem in a complete graph constructed by the input dataset. The parallelization strategy naturally becomes to design an algorithm that can partition a large-scale dense graph and merge the intermediate solution efficiently and correctly. The algorithm we propose is memory-efficient and can be scaled out linearly. Given any practical memory size constraint, this framework guarantees the correct clustering solution without explicitly having all pair distances in the memory. This paper focuses on the design and implementation of Spark-based hierarchical clustering for numeric spaces, but the algorithm is general and applicable to any dataset. In the experiment section, we present a data-dependent characterization of hardness and evaluate clustering efficiency with up to $2,000,000$ data points. Also our algorithm can achieve an estimated speedup of up to 310 on 398 computer cores, which demonstrates its scalability. In addition, we closely examine the performance disparity in term of the total execution time between two frameworks: Spark and MapReduce. The main contributions of this paper are:

- We present SHAS, an efficient Spark-based single-linkage hierarchical clustering algorithm.
- The proposed algorithm is general enough to be used with any dataset that line in a metric space. The algorithm can be used with various distance functions and data types, e.g., numerical data, vector data, text, etc.
- We thoroughly evaluate the performance and scalability properties of the implemented algorithm with synthetic datasets. We show that SHAS performs significantly better than an adoption of state-of-the-art MapReduce-based Single-linkage Hierarchical Clustering (SHC) algorithm [6]. SHAS scales very well when important parameters K, data size, number of nodes, and number of dimensions increase.

The rest of paper is organized as follows. Section 2 describes the SHAS algorithm and examines its system design with Spark framework. Section 3 presents the performance evaluation. Section 4 reviews the existing work related to the parallel hierarchical clustering and comparison between disk-based and in-memory computing frameworks. Section 5 concludes the paper.

II. Introduction

The data, features extracted from the data, and the model are the three major components of a machine learning solution. Over the last decade, it has shown that in real-world settings, the size of the dataset is the most important factor. A large body of literature has repeatedly shown that simple models trained over enormous quantities of data outperform more sophisticated models trained on less data [1]–[3]. This has led to the growing dominance of simple, data-driven solutions.

As one of simple and widely adopted machine learning models, clustering is often an essential first step in data mining to retrieve data's membership. Many applications face the demand to perform this operation over large amount of data. For example, Internet companies collect massive amount of data such as content produced by web crawlers or service logs. Frequently, it needs apply similarity queries to gain valuable understanding of the usage of their services. The understanding may include identifying customers with similar buying patterns, suggesting recommendations, performing correlation analysis, etc. Processing the ever-growing volume of data in a highly scalable and distributed fashion constitutes an answer to such a demand.

Fortunately, frameworks such as MapReduce [4] and Spark [5] can quickly process massive datasets by splitting them into independent chunks that are processed in parallel. Users are alleviated away from the system-level's programming details and only need to cope with the user-level APIs. However, clustering algorithms are generally difficult to parallelize effectively due to high data dependence. Relatively little work has been done. In this paper, we present SHAS, a $\underline{\text{S}}$ingle-linkage $\underline{\text{H}}$ierarchical clustering $\underline{\text{A}}$lgorithm using $\underline{\text{S}}$park framework. The key idea is to reduce the single-linkage hierarchical clustering problem to the minimum spanning tree (MST) problem in a complete graph constructed by the input dataset. The parallelization strategy naturally becomes to design an algorithm that can partition a large-scale dense graph and merge the intermediate solution efficiently and correctly. The algorithm we propose is memory-efficient and can be scaled out linearly. Given any practical memory size constraint, this framework guarantees the correct clustering solution without explicitly having all pair distances in the memory. This paper focuses on the design and implementation of Spark-based hierarchical clustering for numeric spaces, but the algorithm is general and applicable to any dataset. In the experiment section, we present a data-dependent characterization of hardness and evaluate clustering efficiency with up to $2,000,000$ data points. Also our algorithm can achieve an estimated speedup of up to 310 on 398 computer cores, which demonstrates its scalability. In addition, we closely examine the performance disparity in term of the total execution time between two frameworks: Spark and MapReduce. The main contributions of this paper are:

- We present SHAS, an efficient Spark-based single-linkage hierarchical clustering algorithm.
- The proposed algorithm is general enough to be used with any dataset that line in a metric space. The algorithm can be used with various distance functions and data types, e.g., numerical data, vector data, text, etc.
- We thoroughly evaluate the performance and scalability properties of the implemented algorithm with synthetic datasets. We show that SHAS performs significantly better than an adoption of state-of-the-art MapReduce-based $\underline{\text{S}}$ingle-linkage $\underline{\text{H}}$ierarchical $\underline{\text{C}}$lustering (SHC) algorithm [6]. SHAS scales very well when important parameters K, data size, number of nodes, and number of dimensions increase.

The rest of paper is organized as follows. Section 2 describes the SHAS algorithm and examines its system design with Spark framework. Section 3 presents the performance evaluation. Section 4 reviews the existing work related to the parallel hierarchical clustering and comparison between disk-based and in-memory computing frameworks. Section 5 concludes the paper.

III. The SHAS Algorithm

In the section, we describe a parallel algorithm for calculating single-linkage hierarchical clustering (SHC) dendrogram, and show its implementation using Spark's programming model.

A. Hierarchical Clustering

Before dive into the details of the proposed algorithm, we first 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 down by performing splits recursively. As a typical example of agglomerative approach, single-linkage hierarchical clustering (SHC) [7] 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.

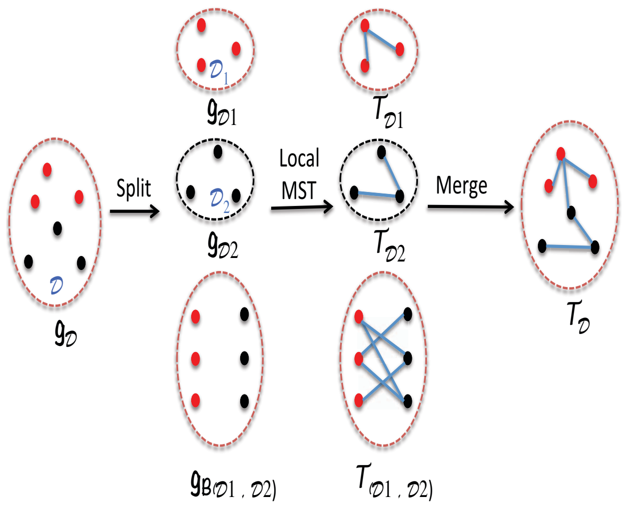
B. Problem Decomposition

Intuitively, we want 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. The absence of any inter-instance dependencies ensures that this strategy scales extremely well as we increases the degree of parallelism (i.e, the number of instances). In the following, we show how we convert the hierarchical clustering problem into a MST finding problem, and the original problem decomposition turns into the graph partitioning accordingly.

Based on the theoretical finding [8] 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 weights are the distances between any two points, the SHC problem with a base dataset $\mathcal{D}$ can be formulated as follows:

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

To show the process of problem decomposition or complete graph partition, a toy example is illustrated in Figure 1.



$\mathcal{D}_1$
$\mathcal{G}_{\mathcal{D}1}$
$\mathcal{T}_{\mathcal{D}1}$
Split
$\mathcal{D}_2$
Local MST
Merge
$\mathcal{D}$
$\mathcal{G}_{\mathcal{D}2}$
$\mathcal{T}_{\mathcal{D}2}$
$\mathcal{G}_{\mathcal{D}}$
$\mathcal{T}_{\mathcal{D}}$
$\mathcal{G}_{B(\mathcal{D}1,\mathcal{D}2)}$
$\mathcal{T}_{(\mathcal{D}1,\mathcal{D}2)}$

Fig. 1: Illustration of the divide-and-conquer strategy on input dataset $\mathcal{D}$. We divide dataset $\mathcal{D}$ into two smaller parts, $\mathcal{D}_1$ and $\mathcal{D}_2$, calculate MSTs for complete graphs induced by $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively, and the complete bipartite graph between them, then merge these three intermediate MSTs to find the MST for $\mathcal{D}$.

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 subgraphs 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.

C. Algorithm Design

The algorithm below desribes how we divide the problem into disjoint subproblems and how the sub-solutions are merged to form the final solution.

Algorithm 1 SHAS, a parallel SHC algorithm using Spark

```
INPUT: a base dataset $\mathcal{D}$, and a merging parameter $K$
OUTPUT: a MST $\mathcal{T}$ induced on $\mathcal{D}$
1: Divide $\mathcal{D}$ into $s$ roughly equal-sized splits:
       $\mathcal{D}_1, \mathcal{D}_1, \ldots, \mathcal{D}_s$
2: Form $\mathcal{C}_s^2$ complete bipartite subgraphs for each pair ($\mathcal{D}_i$,
   $\mathcal{D}_j$) and $s$ complete subgraphs for each split $\mathcal{D}_i$
3: Use Prim's algorithm to compute the sub-MST on
   each subgraph
4: repeat
5:     Taking the sub-MSTs, merge every $K$ of them using
       the idea of Kruskal's algorithm
6: until one MST remains
7: return the final MST $\mathcal{T}'$
```

Following the dividing steps described in step 1-3 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 [9]–[11]. 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 belong 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.

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, populate 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.

From Step 4 in Algorithm 1, we iteratively merge all the intermediate sub-MSTs and the pre-calculated $\mathcal{T}$ to obtain the overall 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.

D. The Main Algorithm

1) Spark: As an in-memory cluster computing framework for iterative and interactive applications, Spark [5] has attracted a lot of attention and become one of the most active Apache open-source projects with 20+ companies as contributors. In particular, Spark is a parallel dataflow system implemented in Scala and centered around the concept of Resilient Distributed Datasets (RDDs) [13]. RDD is essentially an immutable collection partitioned across cluster that can be rebuilt if a partition is lost. When RDD is set to be cached or persisted in memory, each node caches its respective slices from local computation and reuses them in other operations on that RDD. This is the key that Spark can achieve much higher performance than disk-based MapReduce.

RDD abstraction supports two kinds of operations: transformations, which form a new dataset from a base dataset by using functions such as *map*, and actions, which return the final results to the driver program (e.g. *collect*) or a distributed dataset (e.g. *reduceByKey*) after running a series of operations on the dataset. Such an abstraction is provided through a language-integrated APIs in Scala (a statically typed functional programming language for Java VM). Each RDD dataset is represented as a Scala Object, and the transformations to be applied on the dataset are invoked as the methods on those objects.

A Spark cluster consists of masters and workers. Mutiple masters mode can be provided by using Apache ZooKeeper [14] along with some kind of clusters managers such as Yarn [15], Mesos [16] or Spark's "standalone" cluster manager. A master's lifetime can span over several queries. The workers are long-lived processes that can store dataset partitions in memory across operations. When the user runs a driver program, it starts with a master, which defines RDDs for the workers and invokes operations on them.

Spark's programming model is well suited for bulk iterative algorithms because RDDs are cached in memory and the dataflow is created lazily which means the computation is taken place only when RDDs are actually needed. It accepts iterative programs, which create and consume RDDs in a loop. By using Spark's Java APIs, Algorithm 1 can be implemented as a driver program naturally. The main snippet is listed below:

```
1  JavaRDD<String> subGraphIdRDD = sc
2        .textFile(idFileLoc,numGraphs);
3
4  JavaPairRDD<Integer, Edge> subMSTs =
       subGraphIdRDD.flatMapToPair(
5         new LocalMST(filesLoc, numSplits));
6
7  numGraphs = numSplits * numSplits / 2;
8
9  numGraphs = (numGraphs + (K - 1)) / K;
10
11 JavaPairRDD<Integer, Iterable<Edge>>
       mstToBeMerged = subMSTs
12        .combineByKey(
13              new CreateCombiner(),
14              new Merger(),
15              new KruskalReducer(numPoints),
16              numGraphs);
17
18 while (numGraphs > 1) {
19    numGraphs = (numGraphs + (K - 1)) / K;
20    mstToBeMerged = mstToBeMerged
21        .mapToPair(new SetPartitionId(K))
22        .reduceByKey(
23           new KruskalReducer(numPoints),
24           numGraphs);
25 }
```

Listing 1 : The Snippet of SHAS's driver program in Java.

2) Partition phase: In order for a worker to know which two splits to be read, we initially produce $(\mathcal{C}_s^2 + \lceil \frac{s}{2} \rceil)$ input files, each of which contains a single integer gid between 0 and $(\mathcal{C}_s^2 + \lceil \frac{s}{2} \rceil)$ representing the subgraph id. Without loss of generality, the subgraphs with id less than $(\mathcal{C}_s^2)$ are complete bipartite graphs while the rest are the regular complete ones. Given a certain graph type, we apply the corresponding Prim's algorithm accordingly. 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.

3) Local computation phase: Different from [6], where the subMSTs need to be materialized to the disk at the Map side and then shuffle to the Reduce side, Spark are lazy and just logs the transformations such as LocalMST() in Map operator on the base dataset at line 4 in Listing 1. Only when an action operation takes place, in our case, when reduceBy function gets called, the recorded transformations then start to be realized and use as input for reduce operation. Thanks to the Spark's location aware schedule, if all the K inputs of reducers are on the same node, KruskalReducer can be processed right away without waiting for the input shuffling through the wire, otherwise, it will not start until the missing input shuffled through the wire. The data shuffle stage is similar to MapReduce frame, where Map 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.

4) Merge phase: We remap the subgraph id gid using a simple hash function below:

$$gid = gid/K \tag{1}$$

The reassignment of subgraph ids guarantees that K consecutive subgraphs are processed in the same reduce procedure. However, this also implies that the number of parallelism decreases by K per iteration. The reduceFunction combines all the intermediate MSTs using K-way merge iteratively until one MST remains.

IV. Experimental Results

Cloud computing attracts a significant amount of attention from industry, academia, and media because of its on-demand, pay-as-you-go characteristics, etc. As a representative and one of the most widely adopted public cloud platforms, Amazon Elastic Compute Cloud (Amazon EC2) has been used for a host of small and medium sized enterprises (SMEs) for various applications. It comes as no surprise that our experiments are also conducted on Amazon Web Service (AWS). We employ "m2.4xlarge" instance [17] with a 64-bit architecture, featuring 8 virtual CPUs (Intel's Xeon Family), 64.8 GB memory, and two 840 GB ephemeral stores. Among Amazon EC2 instance types, m2 and m3 instance types are optimized for memory-intensive applications and provide low cost per GiB of RAM as well as high network performance. The cluster is set up in the US West (Oregon) region, the AWS's newest infrastructure location in the United States. In our experiments, we vary the size of cluster from 7 to 50 m2.4xlarge instances, and the number of cores ranges from 56 to 400 accordingly. In order to make a fair comparison, we install the latest versions for both frameworks: Spark 1.0.0 [18] and MapReduce 2.0.0 [19].

A. Data Sets

The data sets underlying this analysis are generated synthetically using the IBM synthetic data generator [20]. Considering different data distributions, we synthesize two categories of datasets: *synthetic-cluster* and *synthetic-random*. In *synthetic-cluster* datasets (clust100k, clust500k, and clust2m), a certain number of seed points are selected first as the centroids of clusters, the rest of points are then added randomly to these clusters, while in the *synthetic-random* datasets (rand100k, rand500k, and rand2m), points in each dataset are drawn with a uniform distribution.

To summarize, our testbed contains up to **2,000,000** data points and each data point comprises a numeric vector with up to **10** dimensions. Before the experiments, each data set is copied to the cluster's ephemeral Hadoop File System [21] as a single binary file. The structural properties of the dataset are provided in the table below:

B. Performance

In each experiment, the data is split into a certain number of partitions evenly without any assumption of the data distribution.

TABLE I: Structural properties of the synthetic-cluster and synthetic-random testbed

Name	Points	dimensions	size (**MByte**)
clus100k	100k	5, 10	5, 10
clus500k	500k	5, 10	20, 40
clus2m	2m	5, 10	80, 160
rand100k	100k	5, 10	5, 10
rand500k	500k	5, 10	20, 40
rand2m	2m	5, 10	80, 160

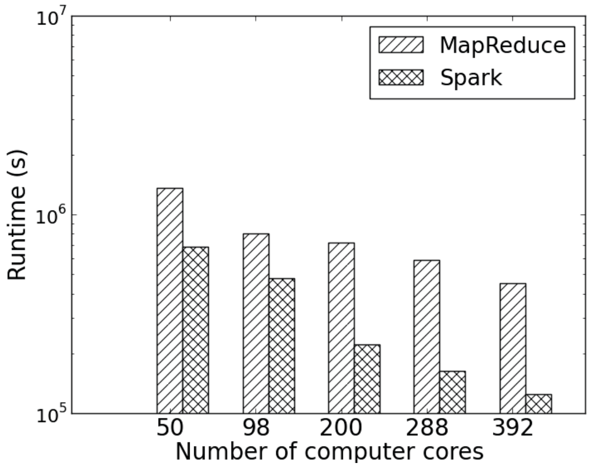


Fig. 2: The execution time comparison between Spark and MapReduce.

1) Total Execution Time: We compare SHAS's performance with the equivalent implementation in MapReduce, both of which are written in Java. Since Spark and MapReduce have a significant difference in their implementation, we only take the total execution time into consideration. Other system metrics such as cpu load and memory usage are out of scope of this paper.

We first evaluate the algorithm on the twelve synthetic datasets described in Table I. Figure 2 illustrates the total execution time of our algorithm on synthetic-cluster datasets. Without any doubt, memory-based Spark greatly outperforms disk-based MapReduce for all the datasets. More importantly, the execution time using Spark decreases much more quickly than using MapReduce. In other words, Spark shows much stronger scalability as the number of cores increases. One reason is that Spark keeps RDDs in memory which reduces the amount of data to be materialized. The other is that per iteration Spark has no framework overhead such as job setup and tear-down as MapReduce.

2) The Speedup: In order to illustrate how SHAS algorithm sustains the speedup as the size of cluster scales up and the amount of data to process increases, we measure the speedup on p cores as $Speedup = \frac{p_0 t_{p_0}}{t_p}$, where p_0 is the minimum computer cores we conduct our experiments, which is 50 in our experiments, and t_p is the SHAS's execution time on p cores. Figure 3 summarizes the speedup results on these twelve datasets with different sizes and dimensionalities. As expected,

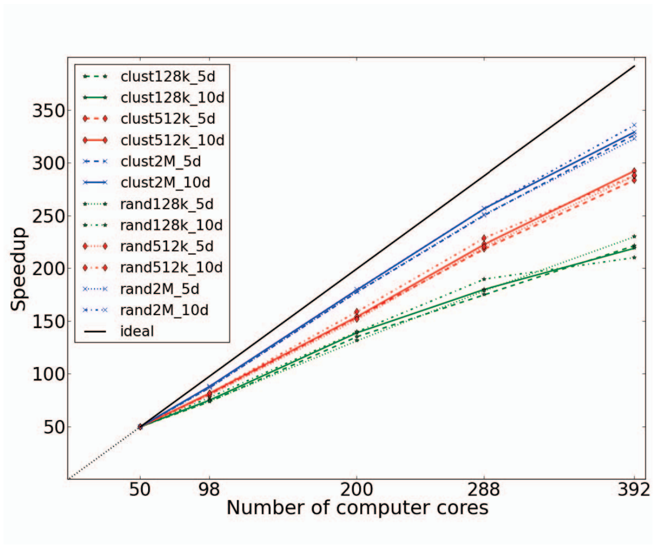


Fig. 3: Speedup on synthetic datasets using 50-392 computer cores.

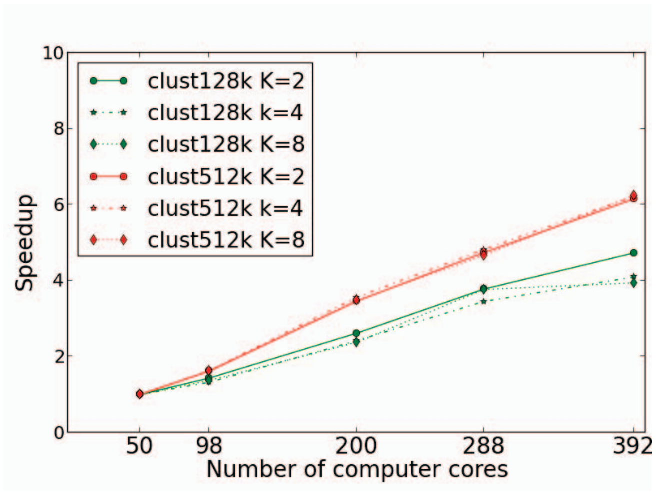


Fig. 4: Speedup with the merge factor K.

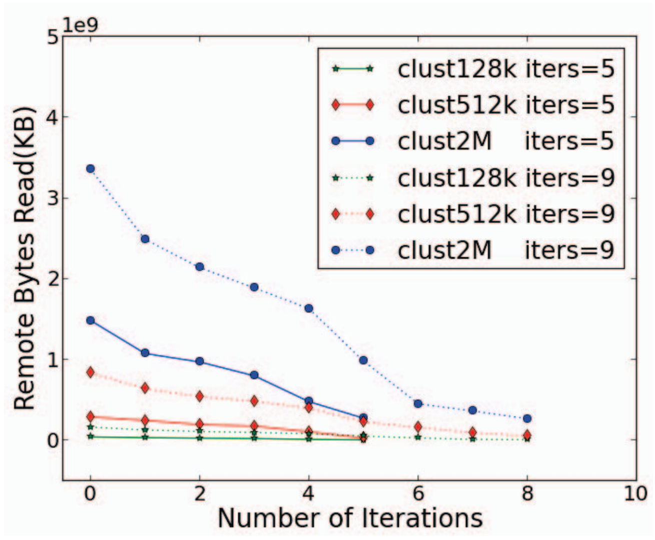


Fig. 5: Total Remote Bytes Read Per Iteration.

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 categories of datasets hardly makes any difference in our algorithm in that 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.

3) The Merge Factor K: The motivation to have the configurable merge factor is to offer a tradeoff between the number of iterations and the number of parallelism. As discussed in Equation (1), larger K leads to fewer iterations. Unfortunately, larger K also implies less number of reducers and smaller degree of parallelism. Therefore, finding a right value of K is very important for overall performance. Figure 4 shows speedup for datasets clust100K and clust500k with K equals to 2, 4, 8, and both datasets achieve better speedup when $K = 2$. It appears that having a larger number of parallelism is more important because if we have more reducers, essentially we avoid shuffling the data which already reside on those reducers and each iteration can finish much quicker. Therefore larger K will deteriorate the performance.

4) The Data Shuffle: In this section, we discuss the cost of I/O to load the partitions as well as the data shuffle patterns in the context of Spark, which includes remote bytes read and bytes written during the data shuffle stage of each iteration.

Recall that when we form the subgraphs, each split need actually be paired with other $s-1$ splits. The size of MST in a bipartite subgraph is $2\frac{|\mathcal{E}|}{s} - 1$, while one in a regular complete subgraph is $\frac{|\mathcal{E}|}{s}$. Therefore, the total number of edges produced by the local prim algorithm is $s|\mathcal{V}|$, where $|\mathcal{V}|$ is the total number of vertices, which decreases drastically from the number of edges in the original graph. The amount of data shuffle is linearly proportional to the number of vertices residing in the merged subgraphs.

The data shuffle patterns are illustrated in Figure 5, in which x axis represents Spark iterations, and y axis shows the aggregated amount of bytes remotely read by all the reducers per iteration. SHAS algorithm requires to find the MST given a certain number of computer cores. The plot depicts the increasing trend of the amount of data shuffle as the number of splits increases. Notably, as we scale up the number of processes, the number of Spark iterations increases. However, the data is dramatically reduced after the first iteration by almost 25%, which verifies our claim that incorrect edges are pruned at a very early stage. The same trend is observed for bytes written at the data shuffle stage. And the amount of vertices decreases by approximately K times due to the deduplication effect at the Kruskal reducer's merging process.

5) The Load Balance: Spark provides a Ganglia web portal to monitor the entire cluster's performance. Ganglia [22], as a cluster-wide monitoring tool, can provide insight into overall cluster utilization and resource bottlenecks. Figure 6 shows the cluster's snapshot at the first iteration. In our algorithm, the dataset is partitioned evenly and induced subgraphs are distributed based on the size of vertices, thus, the workload is balanced among Spark workers. This is verified by the heap map in Figure 6(a). The entire algorithm is CPU bound due to the quadratic time complexity of Prim algorithm. Figure 6 (b) shows one of the later iterations. The edges are pruned by KruskalReducer in the manner of K-way merge. As we can see, the degree of parallelism runs down as the number of reducers decreases, but the reduce operation has much lighter CPU load compared with the Prim algorithm.

V. Related Work

Hierarchical clustering provides a rich representation about the structure of the data points without predetermining the number of clusters. However, the complexity is at least quadratic in the number of data points [23], which is not tolerable with large-scale and high-dimension applications. Several efforts were taken to parallelize hierarchical clustering algorithm, relying on the advance of modern computer architectures and large-scale systems. Different platforms, including multi-core [8], GPU [24], MPI [25] as well as recently popularized MapReduce framework [26], [27], have all seen its implementation.

Clustering using the single-linkage algorithm is closely related to finding the Euclidean minimal spanning tree of a set of points. Sequential algorithms with a runtime of $O(n^2)$ are known [23]. Parallel implementations of single-linkage dates back to late 1980s with the Rasmussen and Willett's implementation on a SIMD array processor [28]. SHRINK [8], proposed by Hendrix *et al.*, is a parallel single-linkage hierarchical clustering algorithm based on SLINK [29]. 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 good speedup. However, the authors only evaluate SHRINK on up to 36 shared memory cores, achieving a speedup of roughly 19.

While both [23] and [8] are based on low communication-latency systems, Feng *et al.* [30] explore the design in PC cluster system with high communication cost. They propose a parallel hierarchical clustering (PARC), which implements CLAP [31] algorithm in a distributed fashion. The algorithm includes sample clustering phase and global clustering phase. The main idea is to form a fuzzy global clustering pattern by exchanging the sample clustering results from each computer node and then refine the initial global clustering with the entire dataset. In order to achieve a high speedup, the authors apply asynchronous MPI communication to exchange the intermediate results. However, the algorithm is only evaluated with 8 computer nodes.

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper, focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analytical tools.

In the new cluster computing framework, Spark, however, users can construct complex directed acyclic graphs (DAGs), even cyclic graphs, each of which defines the dataflow of the application. Separate DAGs can be executed all at once. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39GB dataset with sub-second response time. The following two subsections describe each of these advanced distributed frameworks.

A. Disk-based Computing Framework

MapReduce has emerged as one of the most frequently used parallel programming models for processing large-scale datasets since it was first proposed in 2004 by Dean and Ghemawat [4]. Its open source implementation, Hadoop, has become the de facto standard for both industry and academia. There are many research efforts inspired by MapReduce ranging from applying MapReduce to support Online Analysis, improving MapRedcue's pipeline performance, building high-level languages on top of MapReduce (e.g. DryadLINQ [32], Hive [33], Pig [34]).

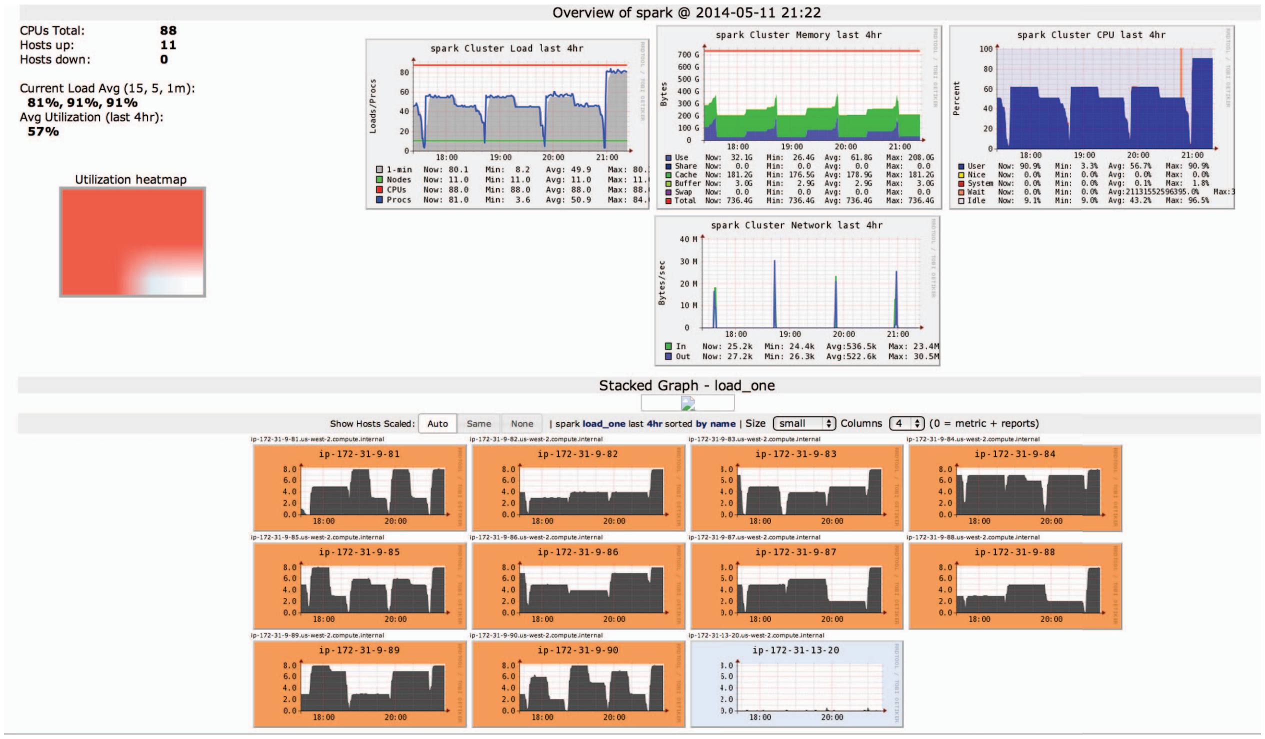
The programming models of MapReduce [4] and Dryad [35] are instances of stream processing, or data-flow models. Because of MapReduce's popularity, programmers start using it to build in-memory iterative application such as PageRank, even though data-flow model is not a natural fit for these applications. Zaharia and *et al.* [13] proposes to add distributed read-only in-memory cache to improve the performance of MapReduce-based iterative computations.

MapReduce executes jobs in a simple but inflexible map-shuffle-reduce structure. Such a structure has so far been sufficient for one-pass batch processing, however, when there are complicatedly cross-dependent, multi-stage jobs, one often has to string together a series of MapReduce jobs and have them executed sequentially in time. This leads to high latency. Another limitation is that data is shared among parallel operations in MapReduce by writing it to a distributed file system, where replication and disk I/O cause substantial overhead [36].

B. In-memory Computing Framework

Designing scalable systems for analyzing, processing and mining huge real-world datasets has become one of the most timely problems facing systems researchers. For example, high-dimensional metric space are particularly challenging to handle, because they cannot be readily decomposed into small parts that could be processed in parallel. This lack of data parallelism renders MapReduce inefficient for computing on such problem, as has been argued by many researchers (for example, [36]). Consequently, in recent years several in-memory based abstraction has been proposed, most notably, Spark [5].

Spark is a cluster computing framework that allows users to define distributed datasets that can be cached in memory across the cluster for applications that require frequent passes through them. One of the advantages of such a programming model is that you can write a single program, similar to DryadLINQ [32]. Due to RDD's immutability, data consistency is very



CPUs Total: **88**
Hosts up: **11**
Hosts down: **0**

Current Load Avg (15, 5, 1m):
81%, 91%, 91%
Avg Utilization (last 4hr):
57%

```
1-min   Now: 80.1   Min:  8.2   Avg: 49.9   Max: 80.
Nodes   Now: 11.0   Min: 11.0   Avg: 11.0   Max: 11.
CPUs    Now: 88.0   Min: 88.0   Avg: 88.0   Max: 88.
Procs   Now: 81.0   Min:  3.6   Avg: 50.9   Max: 84.
```

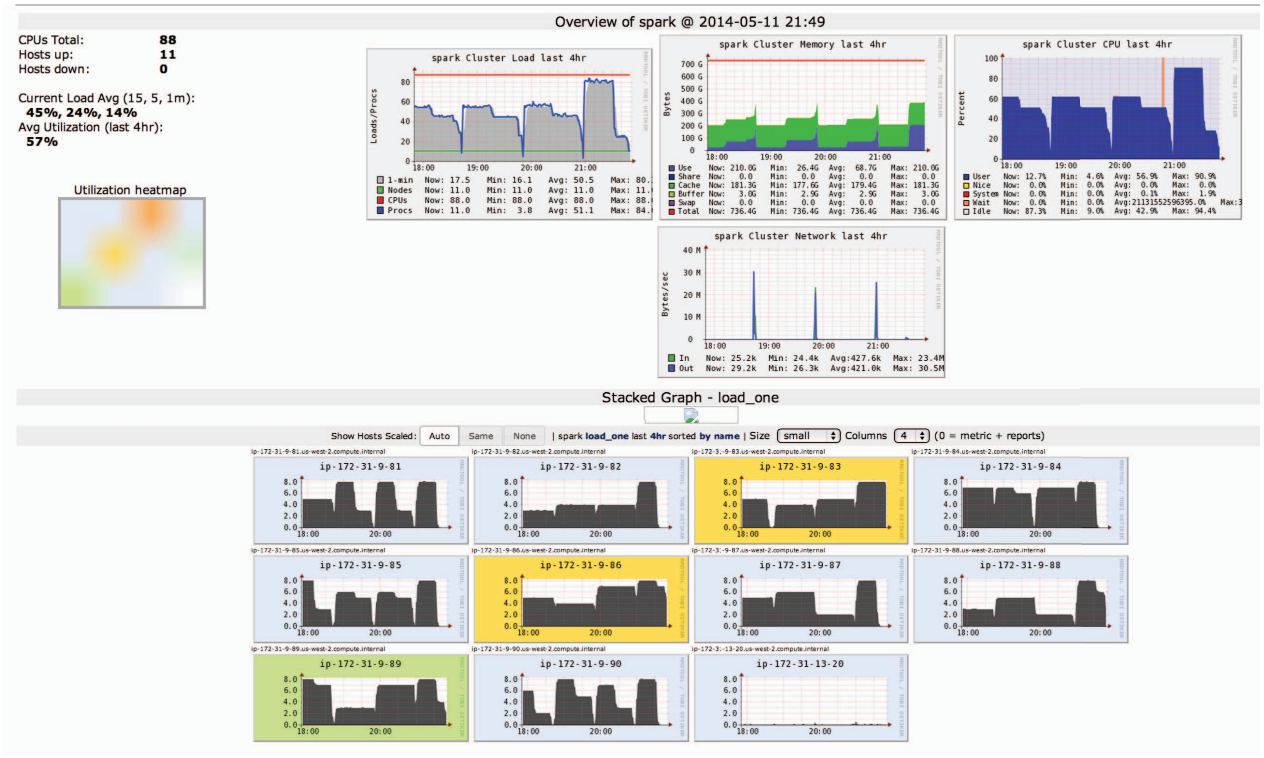
```
Use     Now:  32.1G   Min:  26.4G   Avg:  61.8G   Max: 208.0G
Share   Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
Cache   Now: 181.2G   Min: 176.5G   Avg: 178.9G   Max: 181.2G
Buffer  Now:   3.0G   Min:   2.9G   Avg:   2.9G   Max:   3.0G
Swap    Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
Total   Now: 736.4G   Min: 736.4G   Avg: 736.4G   Max: 736.4G
```

```
User    Now: 90.9%   Min:  3.3%   Avg: 56.7%   Max: 90.9%
Nice    Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
System  Now:  0.0%   Min:  0.0%   Avg:  0.1%   Max:  1.8%
Wait    Now:  0.0%   Min:  0.0%   Avg:2113155259639 5.0%   Max:3
Idle    Now:  9.1%   Min:  9.0%   Avg: 43.2%   Max: 96.5%
```

```
In   Now: 25.2k   Min: 24.4k   Avg:536.5k   Max: 23.4M
Out  Now: 27.2k   Min: 26.3k   Avg:522.6k   Max: 30.5M
```

Show Hosts Scaled: Auto Same None | spark **load_one** last **4hr** sorted **by name** | Size small Columns 4 (0 = metric + reports)

(a) The first iteration



CPUs Total: **88**
Hosts up: **11**
Hosts down: **0**

Current Load Avg (15, 5, 1m):
45%, 24%, 14%
Avg Utilization (last 4hr):
57%

```
1-min   Now: 17.5   Min: 16.1   Avg: 50.5   Max: 80.
Nodes   Now: 11.0   Min: 11.0   Avg: 11.0   Max: 11.
CPUs    Now: 88.0   Min: 88.0   Avg: 88.0   Max: 88.
Procs   Now: 11.0   Min:  3.8   Avg: 51.1   Max: 84.
```

```
Use     Now: 210.0G   Min:  26.4G   Avg:  68.7G   Max: 210.0G
Share   Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
Cache   Now: 181.3G   Min: 177.6G   Avg: 179.4G   Max: 181.3G
Buffer  Now:   3.0G   Min:   2.9G   Avg:   2.9G   Max:   3.0G
Swap    Now:   0.0    Min:   0.0    Avg:   0.0    Max:   0.0
Total   Now: 736.4G   Min: 736.4G   Avg: 736.4G   Max: 736.4G
```

```
User    Now: 12.7%   Min:  4.6%   Avg: 56.9%   Max: 90.9%
Nice    Now:  0.0%   Min:  0.0%   Avg:  0.0%   Max:  0.0%
System  Now:  0.0%   Min:  0.0%   Avg:  0.1%   Max:  1.9%
Wait    Now:  0.0%   Min:  0.0%   Avg:2113155259639 5.0%   Max:3
Idle    Now: 87.3%   Min:  9.0%   Avg: 42.9%   Max: 94.4%
```

```
In   Now: 25.2k   Min: 24.4k   Avg:427.6k   Max: 23.4M
Out  Now: 29.2k   Min: 26.3k   Avg:421.0k   Max: 30.5M
```

Show Hosts Scaled: Auto Same None | spark **load_one** last **4hr** sorted **by name** | Size small Columns 4 (0 = metric + reports)

(b) One of later iterations

Fig. 6: Snapshot of cluster utilization at the first iteration and one of later iterations.

easy to achieve. Users are able to store data in memory across DAGs. The fault tolerance is also inexpensive by only logging lineage rather than replicating or check-pointing data. Even though the model seems being restricted, it is still applicable to a broad variety of applications.

VI. Conclusion

In this paper, we have presented SHAS, a new parallel algorithm for single-linkage hierarchical clustering. We demonstrated how the algorithm scaled using Spark in contrast with MapReduce. By evaluating SHAS empirically with two synthetic datasets generated from different distributions, we observed that it achieved a speedup of up to 300 on 392 computer cores. The parallelization technique employed by SHAS can be extended to other types of problems, particularly those that can be modeled as dense graph problems. Future work on SHAS may involve efforts to take data set distribution into consideration and use better graph partition scheme to reduce the cost of the data shuffling between iterations.

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] A. Halevy, P. Norvig, and F. Pereira, “The unreasonable effectiveness of data,” *IEEE Intelligent Systems*, vol. 24, no. 2, pp. 8–12, Mar. 2009. [Online]. Available: http://dx.doi.org/10.1109/MIS.2009.36

[2] J. Lin and C. Dyer, “Data-intensive text processing with MapReduce,” in *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, ser. NAACL-Tutorials ’09. Stroudsburg, PA, USA: Association for Computational Linguistics, 2009, pp. 1–2.

[3] J. Lin and A. Kolcz, “Large-scale machine learning at twitter,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’12. New York, NY, USA: ACM, 2012, pp. 793–804. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213958

[4] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-53, May 2010.

[6] C. Jin, M. M. A. Patwary, A. Agrawal, W. Hendrix, W. Liao, and A. Choudhary, “Disc: A distributed single-linkage hierarchical clustering algorithm using MapReduce,” in *Proceedings of the 4th International SC Workshop on Data Intensive Computing in the Clouds (DataCloud)*, 2013.

[7] S. Johnson, “Hierarchical clustering schemes,” *Psychometrika*, vol. 32, no. 3, pp. 241–254, Sep. 1967. [Online]. Available: http://dx.doi.org/10.1007/bf02289588

[8] W. Hendrix, M. M. A. Patwary, A. Agrawal, W. Liao, and A. N. Choudhary, “Parallel hierarchical clustering on shared memory platforms,” in *HiPC*, 2012, pp. 1–9.

[9] O. Boruvka, “ O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary),” *Práce Mor. Prírodoved. Spol. v Brne III*, vol. 3, 1926.

[10] J. B. Kruskal, “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem,” *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, feb 1956. [Online]. Available: http://www.jstor.org/stable/2033241

[11] R. C. Prim, “Shortest connection networks and some generalizations,” *Bell System Technology Journal*, vol. 36, pp. 1389–1401, 1957.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. The MIT Press, 2009.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28.

[14] http://zookeeper.apache.org, [Online].

[15] http://hadoop.apache.org/docs/r2.3.0/index.html, [Online].

[16] https://mesos.apache.org, [Online].

[17] Amazon, https://aws.amazon.com/ec2/instance-types, [Online].

[18] Spark, https://spark.apache.org/downloads.html, [Online].

[19] Cloudera, http://www.cloudera.com/content/cloudera-content/cloudera-docs/CDH4/latest/CDH-Version-and-Packaging-Information/CDH-Version-and-Packaging-Information.html, [Online].

[20] R. Agrawal and R. Srikant, “Quest synthetic data generator,” *IBM Almaden Research Center*, 1994.

[21] Amazon, https://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-plan-file-systems.html, [Online].

[22] http://ganglia.sourceforge.net, [Online].

[23] C. F. Olson, “Parallel algorithms for hierarchical clustering,” *Parallel Computing*, vol. 21, pp. 1313–1325, 1995.

[24] D.-J. Chang, M. M. Kantardzic, and M. Ouyang, “Hierarchical clustering with cuda/gpu.” in *ISCA PDCCS*, J. H. Graham and A. Skjellum, Eds. ISCA, 2009, pp. 7–12. [Online]. Available: http://dblp.uni-trier.de/db/conf/ISCApdcs/pdccs2009.html#ChangKO09

[25] R. Cathey, E. C. Jensen, S. M. Beitzel, O. Frieder, and D. Grossman, “Exploiting parallelism to support scalable hierarchical clustering.”

[26] S. Wang and H. Dutta, “Parable: A parallel random-partition based hierarchical clustering algorithm for the MapReduce framework,” *Technical Report, CCLS-11-04*, 2011.

[27] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma, “Finding connected components on map-reduce in logarithmic rounds,” *CoRR*, vol. abs/1203.5387, 2012.

[28] E. Rasmussen and P. Willett, “Efficiency of hierarchic agglomerative clustering using the icl distributed array processor,” *Journal of Documentation*, vol. 45, no. 1, pp. 1–24, 1989.

[29] R. Sibson, “SLINK: An optimally efficient algorithm for the single-link cluster method,” *The Computer Journal*, vol. 16, no. 1, pp. 30–34, jan 1973. [Online]. Available: http://dx.doi.org/10.1093/comjnl/16.1.30

[30] Z. Feng, B. Zhou, and J. Shen, “A parallel hierarchical clustering algorithm for pcs cluster system,” *Neurocomput.*, vol. 70, no. 4-6, pp. 809–818, jan 2007.

[31] Z. Bing, S. Junyi, and P. Qinke, “Clap: clustering algoirthm based on random-sampling and cluster-feature,” *J. Xi’an Jiaotong Univ*, vol. 37, no. 12, pp. 1234–1237, 2003.

[32] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, “Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, “Hive: A warehousing solution over a map-reduce framework,” *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, aug 2009.

[34] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: A not-so-foreign language for data processing,” in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’08. New York, NY, USA: ACM, 2008, pp. 1099–1110. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376726

[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys ’07. New

York, NY, USA: ACM, 2007, pp. 59–72. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273005

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica, “Fast and interactive analytics over hadoop data with spark.” USENIX, 2012.