# DiSC: A Distributed Single-Linkage Hierarchical Clustering Algorithm using MapReduce

Chen Jin, Md. Mostofa Ali Patwary, Ankit Agrawal, William Hendrix, Wei-keng Liao, Alok Choudhary
Northwestern University Evanston, IL 60208
⋆Corresponding author: chen.jin@eecs.northwestern.edu

## ABSTRACT

Hierarchical clustering has been widely used in numerous applications due to its informative representation of clustering results. But its higher computation cost and inherent data dependency prohibits it from performing on large datasets efficiently. In this paper, we present a distributed single-linkage hierarchical clustering algorithm (DiSC) based on MapReduce, one of the most popular programming models used for scalable data analysis. The main idea is to divide the original problem into a set of overlapped subproblems, solve each subproblem and then merge the sub-solutions into an overall solution. Further, our algorithm has sufficient flexibility to be used in practice since it runs in a fairly small number of MapReduce rounds through configurable parameters for data merge phase. In our experiments, we evaluate the DiSC algorithm using synthetic datasets with varied size and dimensionality, and find that DiSC provides a scalable speedup of up to 160 on 190 computer cores.

## 1. INTRODUCTION

Clustering is the process of grouping a set of objects in such a way that the intra-group similarity is maximized while the inter-group similarity is minimized. It is widely used in numerous applications: such as outlier detection [6], information retrieval [26], bioinformatics [19] and image segmentation [16]. However, as the data becomes much larger and higher dimensional, the traditional partitional clustering algorithms such as k-means, may not be able to find clustering results of high quality. Hierarchical clustering, on the other hand, provides a more informative way to represent the clustering results by returning a tree-based hierarchy (known as *dendrogram*), giving the idea of how each data point is positioned relative to the cluster structure. For example, hierarchical document clustering organizes documents into a binary-tree based taxonomy or dendrogram that facilitates browsing and navigation. The hierarchy offers insight about the relationships among clusters whereas the partitional clustering cannot. In addition, hierarchical clustering does not require the number of clusters as the algorithm input and cluster assignment for each data point is deterministic. However, these advantages of hierarchical clustering come at the cost of low efficiency. The need to parallelize the hierarchical clustering algorithm becomes even more imperative as the explosion in the size of the dataset. But the recursive representation of the tree structure poses high data dependency that makes it a very challenging problem to parallelize. Even though there are many existing parallel hierarchical clustering algorithms [5, 9–11, 13], few of these algorithms have been evaluated in a large scale. Furthermore, many realizations of such parallel algorithms store the distance matrix for a dataset explicitly, making the algorithm memory bound [9–11, 13].

In this paper, we present DiSC, a Distributed Single-linkage hierarchical Clustering algorithm using MapReduce 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. The algorithms are 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 500,000 data points. Also our algorithm can achieve an estimated speedup of up to 160 on 190 computer cores, which demonstrates its scalability. In addition, we closely examine the various merge schemes to optimize the algorithm's overall runtime by taking into consideration the overhead from each MapReduce iteration. The main contributions of this paper are:

- A scalable algorithm for Single-linkage hierarchical clustering
- A configurable merge process achieving reasonable trade-off between the number of MapReduce rounds and the degree of parallelism
- A tighter upper bound for the computational complexity in overall distance calculations
- Strong-scaling evaluation on synthetic datasets with varied size and dimensionality
- Seamless integration with MapReduce data pipeline and analytical tool chain

The rest of paper is organized as follows. Section II reviews the existing work related to the parallel single-linkage hierarchical clustering. Section III describes our proposed distributed algorithm, examines its theoretical bound and details the system design with MapReduce framework. Section IV gives experimental results on various datasets and section V draws the conclusion.

## 2. RELATED WORK

As mentioned earlier, 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 [20]. When the dataset is large and high-dimensional, the time complexity would be an obstacle to apply the algorithm. Moreover, the algorithms usually require storing the entire distance matrix in memory, which makes it more challenging for a single machine to compute. Several efforts were taken to parallelize hierarchical clustering algorithms. Due to the advance of modern computer architectures and large-scale system, there are various implementations using different kinds of platforms, including multi-core [15], GPU [10], MPI [9] as well as recently popularized MapReduce framework [23, 27].

SHRINK [15], proposed by Hendrix et al., is an parallel single-linkage hierarchical clustering algorithm based on SLINK [25]. 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 [20] and [15] are based on low communication-latency systems, Feng et al. [14] explore the design in PC cluster system with high communication cost. They propose a parallel hierarhical clustering (PARC), which implements CLAP [5] 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 high speedup, the authors apply asynchoronous MPI communiction to exchange the intermediate results. However, the algorithm is only evaluated with 8 computer nodes.

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 presents PARABLE [27], 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, the paper does not provide the formal theoretical proof on the correctness of the dendrogram alignment algorithm. The experiments only use 30 mappers and 30 reducers for the local clustering and a single reducer for the final global clustering.

Rastogi et al. [23] 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 $k$, and requires at most $2 \log d$ rounds and $3(|V|+|E|)$ communication cost per round in practice, with $n$ being the number of vertices in the largest component and $d$ is the graph diameter. In the same paper, the algorithm is applied to single-linkage hierarchical clustering by using Hash-to-Min or Hash-to-All function. The algorithm 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; then a MST algorithm is applied to return a set of cores by using the minimum core decomposition (MCD) procedure. A separate MapReduce job is required to determine the stop condition at each iteration. Different from our focus on high-dimensional large dataset, Rastogi et al. concentrate on general graphs.

Recently, Lattanzi et al. [18] present Filtering, a novel method for solving graph problem in MapReduce. It tackles dense graphs with large input data, where the data cannot be stored on a single machine and must be processed in parallel. 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.

## 3. SINGLE-LINKAGE HIERARCHICAL CLUSTERING ON MAPREDUCE

### 3.1 Hierarchical Agglomerative Clustering

Intuitively, there are two approaches to form the hierarchical structure of the entire dataset. One is to start with every data point as its own singleton cluster, each step the two closest clusters are merged until all the data points belong to the same cluster. This is called agglomerative hierarchical clustering. Reversely, a divisive approach works the process from top to bottom, by dividing a cluster into two most distant clusters, the procedure is repeated until all the clusters only contain one data member. Among the agglomerative approaches, single-linkage hierarchical clustering (SHC) is one of the most popular algorithms, using the distance between the closest data pair from two different clusters at each merge step. There are some other variations on the distance measure, such as complete linkage, median linkage, etc. 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 in various types of datasets.

### 3.2 Distributed Algorithm Design

Based on the theoretical finding [15] that calculating the SHC dendrogram of a dataset is equivalent to finding the 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 original problem can be formulated as: Given a complete undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i | v_i \in \mathcal{D}\}$ and $\mathcal{E} = \{e_{i,j} | e_{i,j} = d(v_i, v_j)\}$, design a parallel algorithm to find the MST in $\mathcal{G}$.

Not surprisingly, we follow the typical parallelization technique - divide and conquer. First we partition the dataset into $s$ splits, every two of these splits form a subgraph. In this way, any possible edge is assigned to some subgraph, and taking the union of these subgraphs would return us the original graph. However, this data-independent partitioning comes with the cost that some edges might be duplicated on multiple subgraphs. For example, we have a subgraph $\mathcal{G}_{i,j}$ resulting from a split pair ($\mathcal{D}_i$, $\mathcal{D}_j$) and a subgraph $\mathcal{G}_{i,k}$ from ($\mathcal{D}_i$, $\mathcal{D}_k$), then those edges that are exclusively formed by the data points in $\mathcal{D}_i$ are duplicated for both $\mathcal{G}_{i,j}$ and $\mathcal{G}_{i,k}$. However, we will show later that the duplication effect is bounded by at most twice as the original number of edges. Despite the duplication, the simple partitioning scheme lends us a neat implementation with MapReduce and linear scalability as the number of subgraphs increases. Algorithm 1 summarizes the dividing procedure in step 1-2.

**Algorithm 1** Outline of DiSC, a distributed SHC algorithm

```
INPUT: a dataset $D$, $K$
OUTPUT: a MST for $D$
1: Divide $D$ into $s$ roughly equal-sized splits: $D_1, D_2, ..., D_s$
2: Form $\mathcal{C}_s^2$ subgraphs containing the complete subgraph
   for every pair in $\{(D_i, D_j) | i < j$ and $i, j \in [1, s]\}$
3: Use Prim's algorithm to compute the local MST for each
   subgraph in parallel, and output the MST's edge list in
   increasing order of edge weight
4: repeat
5:     Merge the intermediate MSTs for every $K$ subgraphs
       using the idea of Kruskal's algorithm
6: until all vertices belong to the same MST
7: return the final MST
```

Once we get a much smaller subgraph with the number of vertices roughly being $2k$ and the number of edges being $\mathcal{C}_k^2$, where $k = \lceil \frac{n}{s} \rceil$, we can run a serial MST algorithm locally for each subgraph. There are three popular MST algorithms, namely Borůvka's [7], Kruskal's [17] and Prim's [22]. As the earliest known MST algorithm, Borůvka's algorithm proceeds in a sequence of Borůvka steps. It identifies the least weighted edge incident to each vertex, and then form the contracted graph for the next step. Each step takes linear time but the number of vertices is reduced by at least half for the next step, thus the algorithm takes $O(m \log n)$ time, where $m$ is the number of edges and $n$ is the number of vertices. Prim's algorithm starts with any random vertex, and grows the MST one edge at a time. The time complexity for the dense graph is $O(n^2)$ using adjacency matrix. Kruskal's algorithm starts with each vertex as a tree and iteratively picks the least weighted edge that doesn't create a cycle from the unused edge set until all the vertices belong to a single tree. Both Kruskal's and Borůvka's]s algorithms need to pick the eligible least weighted edge, which requires pre-calculating all the edge weights, while Prim's algorithm can use the local information for a vertex to proceed. Given the subgraph is complete and undirected, it avoids constructing the entire weight matrix and keep the space complexity linear with respect to the number of vertices.

Now we just focus on how to apply Prim's algorithm to a complete subgraph in which edge weights are determined by the distance between pairs of data points. To start, we arbitrarily select a vertex as the source of our subgraph, and populate the edge weight array by calculating distance of every other vertex to this vertex. At the same time we track the least weighted edge during the population process and emit the corresponding edge to a reducer. Afterwards, we update every other distance value with respect to the newly added vertex, calculating the minimum weighted edge. We continue expanding the vertex frontier until our subgraph spans all of the vertices. To further improve efficiency, we abort the weight computation that exceeds the current maximum weight to the subgraph for a vertex.

In steps 4-6 of algorithm 1, we iteratively merge the intermediate MSTs from subgraphs into larger MSTs until we have the MST for the entire graph. After step 3, we get the superset of the edges that will form the final MST. In order to filter out extra edges that not belong to the final solution, we apply Kruskal algorithm which greedily picks the least weighted edge that does not create any cycle.

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. A pseudo-code description of this procedure is later described in Algorithm 3. By iteratively conducting this merge procedure, we can quickly eliminate incorrect edges sooner. Recall that when 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 $K$ partial MSTs, we can detect and eliminate edges that create cycles 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.

## 3.3 Theoretical results

The original graph is decomposed into a series of overlapped subgraphs, which causes the edge weight between any two data points in the same partition to get recalculated multiple times. In the following, we are going to tighten the upper bound on the total number of distance calculations performed by DiSC, which is one of the major contributions in this paper. [15] proves the total number of edge weight calculations is no more than twice as that in the original graph when the number of data points is the exact multiple of the number of partitions. However, for the inexact multiple case, it only gives the upper bound of three times as the number of weight calculations. Here we provide the proof for a tighter upper bound for this case.

Theorem 3.1. *When $n$ is not exact multiple of the number of partitions, the total number of edge weight computa-*

*tion performed by Algorithm 1 is no more than $2\,\mathcal{C}_n^2$, where $n$ is the number of data points.*

Proof. Assume we divide $n$ data points into $a$ roughly equal-sized splits, $k$ is the greatest quotient such that the positive remainder $b$ is less than divisor $a$ ($b$ is positive as $n$ is not exact muliptle of $a$), we have

$$n = ak + b, \qquad (1)$$

where $0 < b < a, a \geq 1, n > 2$ and $a, b, k \in \mathcal{N}$

Without loss of generality, we add $b$ extra data points evenly into the first $b$ splits; each has $(k + 1)$ vertices as a result, while the other $(a - b)$ splits remain with $k$ vertices each. This setup leads to three types of subgraphs depending on how we select the split pairs: both splits from the first $b$ splits, or from the other $(a - b)$ splits, or one split from the first $b$ splits while the other from the other $(a - b)$ splits.

Let $\mathcal{D}(n, k)$ represent the total number of edge weight calculations performed by all the subgraphs, we have

$$\mathcal{D}(n,k) = \mathcal{C}_b^2\,\mathcal{C}_{2k+2}^2 + \mathcal{C}_{a-b}^2\,\mathcal{C}_{2k}^2 + (a-b)\,b\,\mathcal{C}_{2k+1}^2 \qquad (2)$$

Applying binomial coefficients' recursive formula that $\mathcal{C}_m^2 = \mathcal{C}_{m-1}^2 + \mathcal{C}_{m-1}^1$, Equation 2 can be rewritten as follows:

$$\begin{aligned}
\mathcal{D}(n,k) &= \mathcal{C}_{2k+1}^2\mathcal{C}_a^2 + (2k+1)\mathcal{C}_b^2 - 2k\,\mathcal{C}_{a-b}^2 \\
&= a(a-1)k^2 + [2ab + \frac{a}{2} + b - 1 - \frac{a^2}{2}]k + \mathcal{C}_b^2 \\
&\leq a(a-1)k^2 + (2ab - \frac{a^2}{2})k + \mathcal{C}_b^2 && (3) \\
&= (n-b)(n+b-k-a/2) + \mathcal{C}_b^2 && (4) \\
&\leq (n-b)(n+b-\sqrt{2(n-b)}) + \mathcal{C}_b^2 \\
&\leq (n^2 - b^2/2) - (n-b)\sqrt{2(n-b)} \\
&\leq n^2 - b^2/2 - (n-b) && (5) \\
&= n^2 - n - [(b - \frac{1}{2})^2 - \frac{1}{4}] \\
&\leq n^2 - n \\
&= 2\,\mathcal{C}_n^2
\end{aligned}$$

Note that $b$ is no less than 1 based on Equation 1, $\mathcal{D}(n, k)$ can be relaxed to expression (3). By substituting $ak$ with $(n - b)$, (3) is simplified to (4). After apply Cauchy-Schwarz inequality on (4), and use the fact that $\sqrt{n-b}$ is at least 1, we can derive expression (5), which eventually leads to the upper bound $2\,\mathcal{C}_n^2$. □

We conjecture $\mathcal{D}(n, k)$ is a quasi-convex function, due to the observation that the amount of overhead (in terms of repeated calculations) in fact decreases beyond a certain number of processes. In the extreme case, partitioning the dataset into sets of size 2 results in no repeated distance calculations; the number of splits that maximizes $\mathcal{D}(n, k)$ is approximately $\sqrt{2n}$ [15], which leads to $(n - \sqrt{n/2})$ subgraphs. In order to deploy a feasible number of subgraphs on a cluster, the number of splits should be much less than this critical value $\sqrt{2n}$, and the duplication effect is less than 2 accordingly.

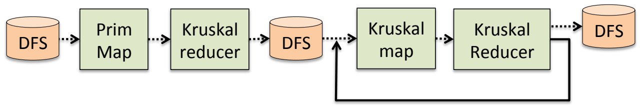


Figure 1: DiSC algorithm on MapReduce

### 3.4 DiSC on MapReduce (MR)

MapReduce, first introduced by Dean and Ghemawat [12], has become one of the most pervasive programming paradigms for large-scale data analysis. It alleviates users from distributed system's engineering work such as data partition, data locality, data replication, as well as task scheduling, data shuffle among tasks and fault tolerance. Programmers or data analysts only need to implement several MR primitives, including Mapper, Reducer, Combiner, RecordReader, and Partitioner. These stubs all provide easy-to-use interface and can be adapted to a wide range of applications.

A MapReduce job can split into 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.

Hadoop [1], the open source implementation of MapReduce framework, has become the de-facto standard. It is widely used in both industry and academia, and strongly supported by a large open source community. In the following, we present the detailed implementation of DiSC using Hadoop. As an integral component of Apache Hadoop framework, Hadoop Distributed File system (HDFS) is a distributed file system designed for commodity hardware. MapReduce computational engine retrieves data from HDFS and outputs the results back to it for the input of next available MR jobs in the data analytical pipeline. As illustrated in Figure 1, DiSC algorithm is decomposed into two types of MR jobs. The first one is called Prim-MR which consists of PrimMapper and KruskalReducer. It is executed only once and followed by a series of Kruskal-MR jobs which repetitively do Kruskal merging process until the final MST is complete.

Figure 2 sketches the detailed implementation of Prim-MR. The original dataset is divided into $s$ smaller splits. Each split is stored as the built-in SequenceFile format provided by Hadoop. SequenceFile is a flat binary file for key-value pairs and extensively used in Hadoop for input/output formats. In our case, the files 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 generate $\mathcal{C}_s^2$ input files, each of which contains a single integer value *gid* between 1 and $\mathcal{C}_s^2$ to represent the subgraph id. This value can uniquely represent pair $(i, j)$, and $i$ and $j$ can be

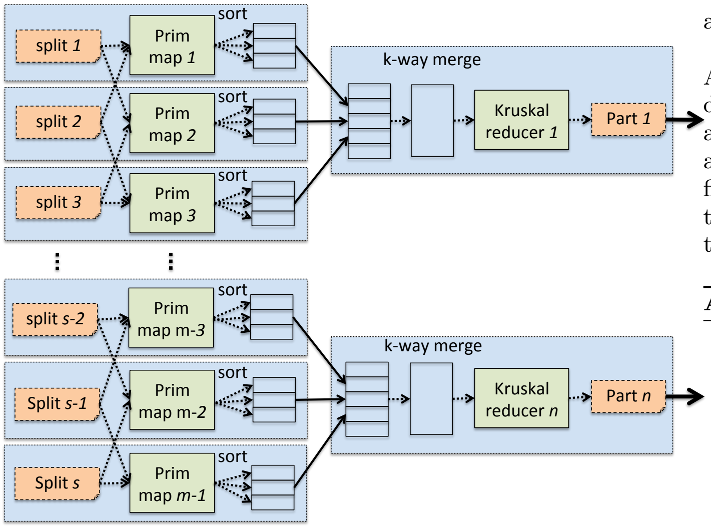


Figure 2: Prim-MR Diagram: Prim-MR is the first MapReduce job in DiSC algorithm. It consists of $m$ Prim maps and $n$ Kruskal reducers, where $n = \lceil m/\text{K} \rceil$

easily calculated from *gid* using simple algebra.

Each mapper runs the local sequential Prim's algorithm to identify the intermediate MST on subgraph *gid*. The PrimMapper is given in Algorithm 2, where the input key is *gid* and input value is data point. Once the total number of vertices exceeds the aggregated number of vertices from the two data splits, we start to calculate the MST. As described in the earlier section, given a complete graph, Prim's algorithm starts with a single-node tree, and then augments the tree one vertex at a time by the least weighted edge from all the links between this vertex and all the identified vertices in the tree. Different from the algorithm in [15], we can emit the newly added edge immediately with no need to wait until all the MST edges are resolved.

**Algorithm 2** PrimMapper

```
map(Int $gid$, Point $value$)
    if ($cnts < numLocalVertices$) do
        $dataset[cnts++] = value$
    end
    if ($cnts >= numLocalVertices$) do
        PrimLocal $mst$ = new PrimLocal($dataset$)
        $mst$.emitMST()
    end
```

While PrimMapper 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. Since Hadoop sorts key by default, we leverage this implementation specifics by using edge weight as key. Moreover, as illustrated in Figure 2, there is a built-in merge property at the reducer, we can design the partitioning or hash function simply as $gid/K$, where *gid* is the subgraph id, and $K$ is the number of subgraphs for the merge procedure.

Algorithm 3 outlines the implementation of Kruskal Reducer. It takes edge as key which contains edge weight and a pair of endpoint ids, the list of values consists of *gid*s that are associated with a given edge key. We define a union-find data structure to keep track of the membership of all the connected components and filter out the incorrect edges that create cycles.

**Algorithm 3** KruskalReducer

```
configure()
    UnionFind $uf$ = new UnionFind($numVertices$);
reduce(Edge $edge$, Iterator<Int> $values$)
    while ($values$.hasNext()) do
        $gid = values$.next()
        if $uf$.union($edge$.getLeft(), edge.getRight()) then
            emit($gid/K$, $edge$)
        end
    end
```

In Kruskal-MR job, the mapper is an Identity Mapper which just passes through (key, value) pairs as they are and reuse the Kruskal Reducer. The same job is repeated until we reach the final MST solution.

In the next section, we will evaluate the performance of these two types of MR jobs respectively, and argue that as the number of splits increases, the number of MapReduce rounds also increases accordingly. The overhead incurred due to setting up and tearing down each MR job can thus no longer be negligible [8, 24]. Reducing the number of iterations also needs to be considered in order to optimize the total run time, and our implementation provides the natural mechanism to exploit $K$-way merge.

# 4. EXPERIMENTAL RESULTS

Our experiments are conducted on Jesup [2], a Hadoop testbed at NERSC. Jesup is a 80-node cluster where each compute node is quad-core Intel Xeon X5550 ”Nehalem” 2.67 GHz processors (eight cores/node) with 24 GB of memory per node. It is a part of Carver cluster [3], a liquid-cooled IBM iDataPlex System with 1202 computer nodes. It is featured with relatively high memory per core and a batch system that support long running jobs. All nodes are interconnected by 4X QDR InfiniBand technology, providing 32 Gb/s of point-to-point bandwidth for high-performance message passing and I/O. However, different from most commodity cluster, all compute nodes in Jesup are ”diskless”, which implies that the data is always resident in RAM. Therefore no data locality can be leveraged in our experiments. Despite this downside caused by the system specifics, our experiments still show the real behavior for both computation and data communication as in many other clusters.

## 4.1 Datasets

We evaluate our MapReduce implementation against 8 datasets, which are divided into two categories, each with four datasets, respectively. These two categories, *synthetic-cluster* and *synthetic-random*, have been generated synthetically using the IBM synthetic data generator [4,21]. In synthetic-cluster datasets (*clust20k*, *clust100k*, and *clust500k*), first a specific number of random points are taken as different clusters, points are then added randomly to these clusters. In the synthetic-random datasets (*rand20k*, *rand100k*, and *rand500k*), points in each dataset have been generated uniformly at random. Our testbed contains up to **500,000** data points and each data point is a vector of up to **10** dimensions.

## 4.2 Performance

We first evaluate the scalability of our algorithm on the twelve synthetic datasets described in Table **??**. Figure 3 illustrates timing results of our algorithm on synthetic-cluster datasets. "Total" is the entire execution time, and it breaks down into two components: "Prim-MR" measuring the runtime for the first MapReduce job with Prim Mapper and Kruskal Reducer, and "Kruskal-MRs" measuring the runtime for a series of Kruskal-MR jobs until the algorithm completes. Figure 3(a) and (b) show that the runtime actually turns worse for small datasets when increasing the number of computer cores. This makes sense since small datasets can fit in one or a few machines' memory, the overhead introduced by the duplicated edge weight calculations and multiple MapReduce rounds would offset the computational gain from data parallelism. Figure 3 (f) demonstrates the nice decreasing trend for both "Total" and "Prim-MR". As "Prim-MR" is the dominating component in the entire runtime, the increase of "Kruskal-MRs" runtime for more computer cores does not affect the overall performance. Figure 3 (c), (d) and (e) exhibit the decreasing trend up to a certain number of computer cores and then it flats out or even increases afterwards. Therefore, adding more machines does not necessarily improve the performance. We also need to take into consideration the algorithm's constraints (duplication effect) and the framework overhead (MapReduce job setup, tear-down, data shuffle, etc.).

In addition, we measure the speedup on $p$ cores as $S = \frac{p_0 t_p}{t_{p_0}}$, where $p_0$ is the minimum computer cores we conduct our experiments, and $t_p$ is the DiSC's execution time on $p$ cores. Since we already discussed the speedup for small datasets, Figure 4 only presents the speedup results for medium and large datasets from 10 to 190 computer cores with different $K$ values, where $K$ is the number of subgraphs that can be merged at one reducer. For both medium and large datasets, $K = 4$ 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 4 seems to be a sweet spot achieving a reasonable trade-off between these two affects.

Figure 5 summarizes the speedup results on these twelve datasets with different size and dimensionality. 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 datasets 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.

## 4.3 I/O and data shuffle

In this section, we evaluate the data patterns with respect to MapReduce metrics, including file read/write and data shuffle from mapper to reducer per iteration. Recall that when we form the subgraphs, each split need actually be paired with other $s - 1$ splits. Therefore, the amount of data read from the disk is $\sqrt{p}\mathcal{O}(|\mathcal{V}|)$ $(p = \mathcal{C}_s^2)$. In Figure 6, each bar represents a MapReduce round, and bars in the same color represent a series of MR rounds that DiSC algorithm requires to find the MST given a certain number of computer cores. For example, the first bar represents Prim-MR round in DiSC algorithm. Figure 6 (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 Kruskal reducer's merging process. Figure 6 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.

# 5. CONCLUSION

In this paper, we have presented DiSC, a distributed algorithm for single-linkage hierarchical clustering. We provided a theoretical analysis of the algorithm, including a upper bound on the computation cost. We also evaluated DiSC empirically using both synthetic datasets with varied size and dimensionality, and observed that it achieves speedup of up to 160 on 190 computer cores. The parallelization technique employed by DiSC may be extended to other types of problems, particularly those that can be modeled as dense graph problems. Future work on DiSC may involve efforts to reduce the duplicated work by using better graph partition scheme and achieve higher speedups.

# 6. REFERENCES

[1] `http://hadoop.apache.org`. [Online].

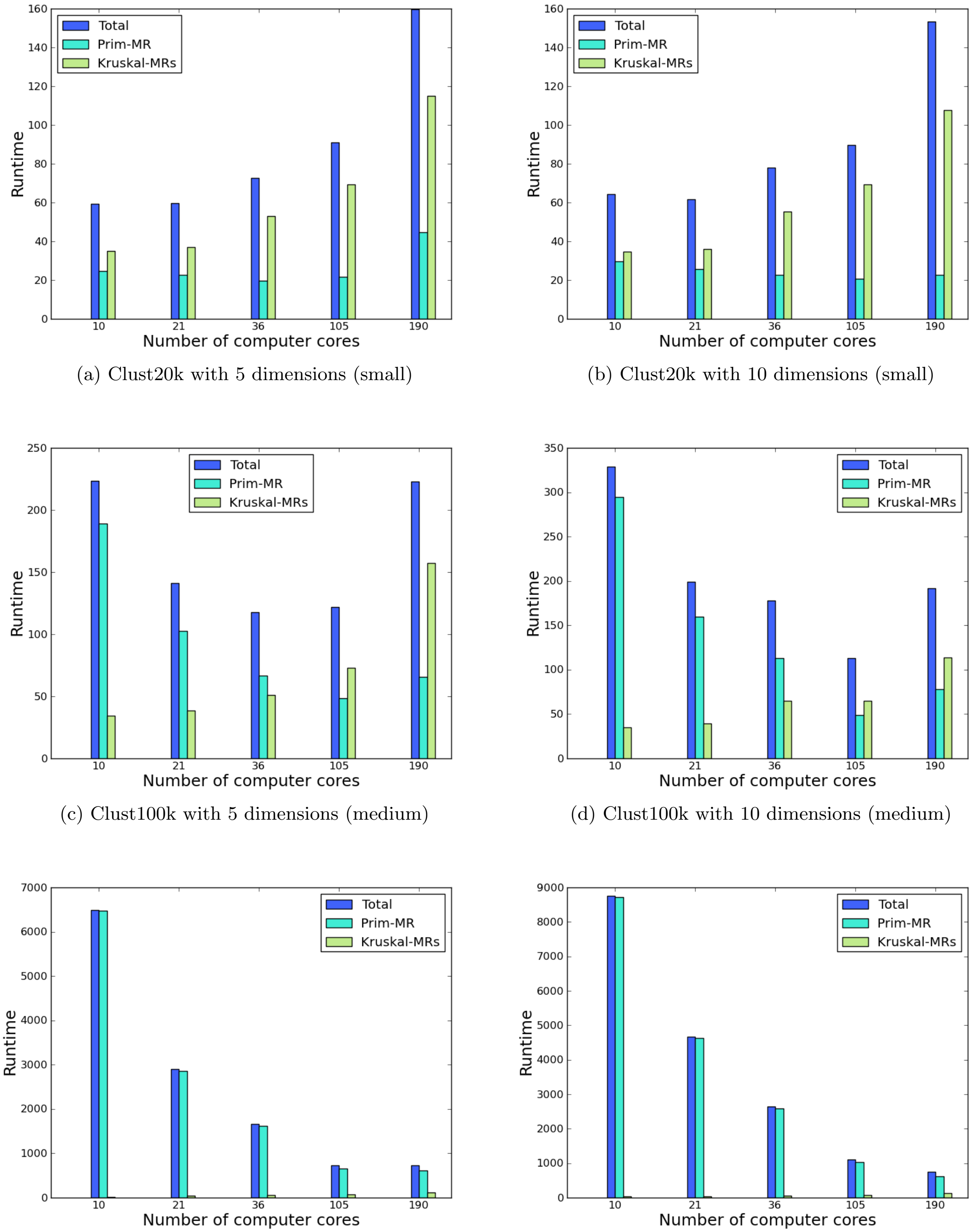
[2] `http://www.nersc.gov/users/computational-systems/testbeds/jesup`. [Online].

[3] `http://www.nersc.gov/users/computational-systems/carver`. [Online].

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

[5] Z. Bing, S. Junyi, and P. Qinke. Clap: clustering algoirthm based on random-sampling and cluster-feature. *J. Xi'an Jiaotong Univ*, 37(12):1234–1237, 2003.

[6] D. Birant and A. Kut. St-dbscan: An algorithm for clustering spatial-temporal data. *Data Knowl. Eng.*, 60(1):208–221, Jan. 2007.



(a) Clust20k with 5 dimensions (small)

(b) Clust20k with 10 dimensions (small)

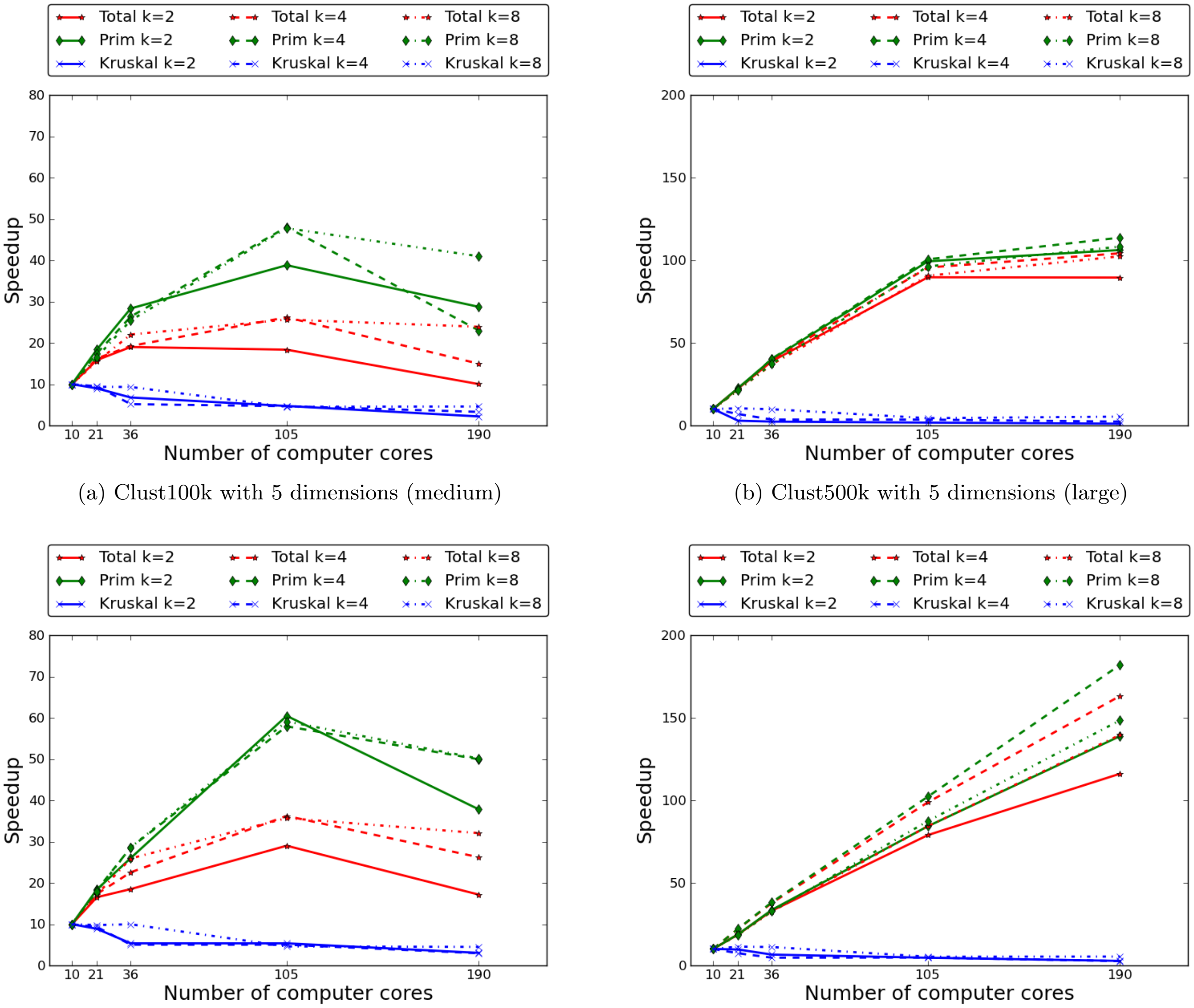
(c) Clust100k with 5 dimensions (medium)

(d) Clust100k with 10 dimensions (medium)

(e) Clust500k with 5 dimensions (large)

(f) Clust500k with 10 dimensions (large)

Figure 3: Result on the synthetic dataset using 2-way merging on 10-190 computer cores. The I/O and communication cost is relatively low. Overall, the MR implementation achieves almost linear speedup in the number of processes.



(a) Clust100k with 5 dimensions (medium)

(b) Clust500k with 5 dimensions (large)

(c) Clust100k with 10 dimensions (medium)

(d) Clust500k with 10 dimensions (large)

Figure 4: Speedup on the synthetic datasets using 10-190 computer cores.

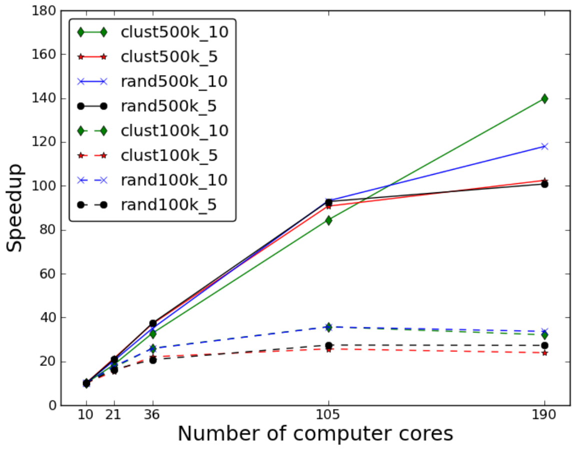


Figure 5: Speedup on synthetic datasets using 10-190 computer cores.

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

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

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

[10] D.-J. Chang, M. M. Kantardzic, and M. Ouyang. Hierarchical clustering with cuda/gpu. In J. H. Graham and A. Skjellum, editors, *ISCA PDCCS*, pages 7–12. ISCA, 2009.

[11] M. Dash and P. Scheuermann. Efficient parallel hierarchical clustering. In *In International Europar Conference (EURO-PARâĂŹ04*, 2004.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[13] Z. Du and F. Lin. A novel parallelization approach for hierarchical clustering. *Parallel Computing*, 31(5):523–527, 2005.

[14] Z. Feng, B. Zhou, and J. Shen. A parallel hierarchical clustering algorithm for pcs cluster system. *Neurocomput.*, 70(4-6):809–818, Jan. 2007.

[15] W. Hendrix, M. M. A. Patwary, A. Agrawal, W. keng Liao, and A. N. Choudhary. Parallel hierarchical clustering on shared memory platforms. In *HiPC*, pages 1–9, 2012.

[16] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review, 1999.

[17] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, Feb. 1956.

[18] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM.

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

[20] C. F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21:1313–1325, 1995.

[21] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 3.0. Technical report, Technical Report CUCIS-2005-08-01, Northwestern University, 2010.

[22] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technology Journal*, 36:1389–1401, 1957.

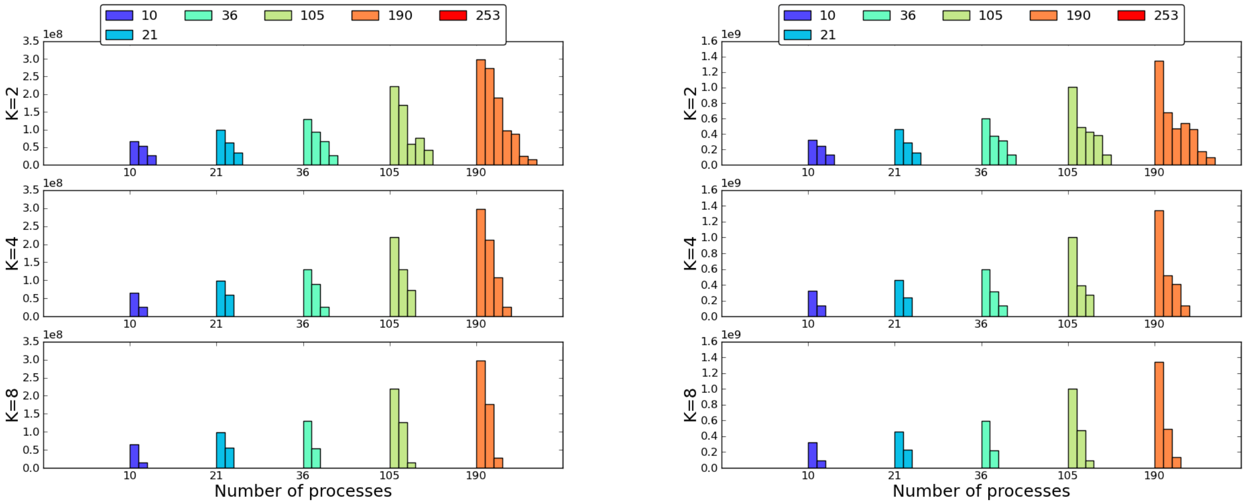
[23] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components on map-reduce in logarithmic rounds. *CoRR*, abs/1203.5387, 2012.

[24] J. Rosen, N. Polyzotis, V. R. Borkar, Y. Bu, M. J. Carey, M. Weimer, T. Condie, and R. Ramakrishnan. Iterative mapreduce for large scale machine learning. *CoRR*, abs/1303.3517, 2013.

[25] R. Sibson. SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, Jan. 1973.

[26] M. Surdeanu, J. Turmo, and A. Ageno. A hybrid unsupervised approach for document clustering. In *INT. CONF. 11H ACM SIGKDD ON KNOWLEDGE DISCOVERY IN DATA MINING*, pages 685–690, 2005.

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



(a) Data shuffle

(b) File read

Figure 6: Data patterns for DiSC algorithm, including the amount of bytes in the stages of data shuffle, file read. (file write is omitted here due to lack of space and it exhibits the same trend as file read. Each bar represents a MR round, bars in the same color represent a series of MR rounds that DiSC algorithm requires to find the MST using a certain number of computer cores.)