# Finding Connected Components in Map-Reduce in Logarithmic Rounds

Vibhor Rastogi Ashwin Machanavajjhala Laukik Chitnis Anish Das Sarma
{vibhor.rastogi, ashwin.machanavajjhala, laukik, anish.dassarma}@gmail.com

***Abstract*—Given a large graph $G = (V, E)$ with millions of nodes and edges, how do we compute its connected components efficiently? Recent work addresses this problem in map-reduce, where a fundamental trade-off exists between the number of map-reduce rounds and the communication of each round. Denoting $d$ the diameter of the graph, and $n$ the number of nodes in the largest component, all prior techniques for map-reduce either require a linear, $\Theta(d)$, number of rounds, or a quadratic, $\Theta(n|V| + |E|)$, communication per round.**

**We propose here two efficient map-reduce algorithms: *(i)* Hash-Greater-to-Min, which is a randomized algorithm based on PRAM techniques, requiring $O(\log n)$ rounds and $O(|V| + |E|)$ communication per round, and *(ii)* Hash-to-Min, which is a novel algorithm, provably finishing in $O(\log n)$ iterations for path graphs. The proof technique used for Hash-to-Min is novel, but not tight, and it is actually faster than Hash-Greater-to-Min in practice. We conjecture that it requires $2 \log d$ rounds and $3(|V| + |E|)$ communication per round, as demonstrated in our experiments. Using secondary sorting, a standard map-reduce feature, we scale Hash-to-Min to graphs with very large connected components.**

**Our techniques for connected components can be applied to clustering as well. We propose a novel algorithm for agglomerative single linkage clustering in map-reduce. This is the first map-reduce algorithm for clustering in at most $O(\log n)$ rounds, where $n$ is the size of the largest cluster. We show the effectiveness of all our algorithms through detailed experiments on large synthetic as well as real-world datasets.**

## I. Introduction

Given a large graph $G = (V, E)$ with millions of nodes and edges, how do we compute its connected components efficiently? With the proliferation of large databases of linked data, it has become very important to scale to large graphs. Examples of large datasets include the graph of webpages, where edges are hyperlinks between documents, social networks that link entities like people, and Linked Open Data[1] that represents a collection of linked structured entities. The problem of finding connected components on such graphs, and the related problem of *undirected $s$-$t$ connectivity* (USTCON [14]) that checks whether two nodes $s$ and $t$ are connected, are fundamental as they are basic building blocks for more complex graph analyses, like clustering.

The number of vertices $|V|$ and edges $|E|$ in these graphs is very large. Moreover, such graphs often arise as intermediate outputs of large batch-processing tasks (e.g., clustering Web pages and entity resolution), thus requiring us to design algorithms in a distributed setting. Map-reduce [6] has become a very popular choice for distributed data processing. In map-reduce, there are two critical metrics to be optimized – number of map-reduce rounds, since each additional job incurs significant running time overhead because of synchronization and congestion issues, and communication per round, since this determines the size of the intermediate data.

There has been prior work on finding connected components iteratively in map-reduce, and a fundamental trade-off exists between the number of rounds and the communication per round. Starting from small clusters, these techniques iteratively expand existing clusters, by either adding adjacent one-hop graph neighbors, or by merging existing overlapping clusters. The former kind [5], [10], [17] require $\Theta(d)$ map-reduce rounds for a graph with diameter $d$, while the latter [1] require a larger, $\Theta(n|V| + |E|)$, computation per round, with $n$ being the number of nodes in the largest component.

More efficient $O(\log n)$ time PRAM algorithms have been proposed for computing connected components. While theoretical results simulating $O(\log n)$ PRAM algorithms in map-reduce using $O(\log n)$ rounds exist [12], the PRAM algorithms for connected components have not yet been ported to a practical and efficient map-reduce algorithm. (See Sec. II-A for a more detailed description)

In this paper, we present two new map-reduce algorithms for computing connected components. The first algorithm, called Hash-Greater-to-Min, is an efficient map-reduce implementation of existing PRAM algorithms [11], [21], and provably requires at most $3 \log n$ rounds with high probability, and a per round communication cost[2] of at most $2(|V| + |E|)$. The second algorithm, called Hash-to-Min, is novel, and provably finishes in $O(\log n)$ rounds for path graphs. The proof technique used for Hash-to-Min is novel, but not tight, and our experiments show that it requires at most $2 \log d$ rounds and $3(|V| + |E|)$ communication per rounds.

Both of our map-reduce algorithms iteratively merge overlapping clusters to compute connected components. Low communication cost is achieved by ensuring that a single cluster is replicated exactly once, using tricks like pointer-doubling, commonly used in the PRAM literature. The more intricate problem is processing graphs for which connected components are so big that either (i) they do not fit in the memory of a single machine, and hence cause failures, or (ii) they result in heavy data skew with some clusters being small, while others being large.

The above problems mean that we need to merge overlapping clusters, i.e. remove duplicate nodes occurring in multiple clusters, without materializing entire clusters in memory. Using Hash-to-Min, we solve this problem by maintaining each cluster as key-value pairs, where the key is a common cluster id and values are nodes. Moreover, the values are kept sorted (in lexicographic order), using a map-reduce capability

[1] http://linkeddata.org/

[2] Measured as total number of $c$-bit messages communicated, where $c$ is number of bits to represent a single node in the graph.

called secondary-sorting, which incurs no extra computation cost. Intuitively, when clusters are merged by the algorithm, mappers individually get values (i.e, nodes) for a key, sort them, and send them to the reducer for that key. Then the reducer gets the 'merge-sorted' list of all values, corresponding to nodes from all clusters needing to be merged. In a single scan, the reducer then removes any duplicates from the sorted list, without materializing entire clusters in memory.

We also present two novel map-reduce algorithms for single-linkage agglomerative clustering using similar ideas. One using Hash-to-All that provably completes in $O(\log n)$ map-reduce rounds, and at most $O(n|V| + |E|)$ communication per round, and other using Hash-to-Min that we conjecture completes in $O(\log d)$ map-reduce rounds, and at most $O(|V|+|E|)$ communication per round. We believe that these are the first Map-Reduce algorithm for single linkage clustering that finish in $o(n)$ rounds.

All our algorithms can be easily adapted to the Bulk Synchronous Parallel paradigm used by recent distributed graph processing systems like Pregel [16] and Giraph [4]. We choose to focus on the map-reduce setting, since it is more impacted by a reduction in number of iterations, thus more readily showing the gains brought by our algorithms (see Sections II and VI-C for a more detailed discussion).

**Contributions and Outline:**

- We propose two novel algorithms for connected components – *(i)* Hash-Greater-to-Min, which provably requires at most $3 \log n$ rounds with high probability, and at most $2(|V| + |E|)$ communication per round, and *(ii)* Hash-to-Min, which we prove requires at most $4 \log n$ rounds on path graphs, and requires $2 \log d$ rounds and $3(|V| + |E|)$ communication per round in practice. (Section III)
- While Hash-Greater-to-Min requires connected components to fit in a single machine's memory, we propose a robust implementation of Hash-to-Min that scales with arbitrarily large connected components. We also describe extensions to Hash-to-Min for load balancing, and show that on large social network graphs, for which Hash-Greater-to-Min runs out of memory, Hash-to-Min still works efficiently. (Section IV)
- We also present two algorithms for single linkage agglomerative clustering using our framework: one using Hash-to-All that provably finishes in $O(\log n)$ map-reduce rounds, and at most $O(n|V| + |E|)$ communication per round, and the other using Hash-to-Min that we again conjecture finishes in $O(\log d)$ map-reduce rounds, and at most $O(|V|+|E|)$ communication per round. (Section V)
- We present detailed experimental results evaluating our algorithms for connected components and clustering and compare them with previously proposed algorithms on multiple real-world datasets. (Section VI)

We present related work in Sec. II, followed by algorithm and experiment sections, and then conclude in Sec. VII.

## II. Related Work

The problems of finding connected components and undirected $s$-$t$ connectivity (USTCON) are fundamental and very well studied in many distributed settings including PRAM, MapReduce, and BSP. We discuss each of them below.

| Name | # of steps | Communication |
|---|---|---|
| Pegasus [10] | $O(d)$ | $O(\|V\| + \|E\|)$ |
| Zones [5] | $O(d)$ | $O(\|V\| + \|E\|)$ |
| L Datalog [1] | $O(d)$ | $O(n\|V\| + \|E\|)$ |
| NL Datalog [1] | $O(\log d)$ | $O(n\|V\| + \|E\|)$ |
| PRAM [21], [19], [8], [11], [12] | $O(\log n)$ | shared memory[3] |
| **Hash-Greater-to-Min** | $3 \log n$ | $2(\|V\| + \|E\|)$ |

TABLE I
Complexity comparison with related work: $n = \#$ of nodes in largest component, and $d =$ graph diameter

### A. Parallel Random Access Machine (PRAM)

The PRAM computation model allows several processors to compute in parallel using a common shared memory. PRAM can be classified as CRCW PRAM if concurrent writes to shared memory are permitted, and CREW PRAM if not. Although, map-reduce does not have a shared memory, PRAM algorithms are still relevant, due to two reasons: (i) some PRAM algorithms can been ported to map-reduce by case-to-case analyses, and (ii), a general theoretical result [12] shows that any $O(t)$ CREW PRAM algorithm can be simulated in $O(t)$ map-reduce steps.

For the CRCW PRAM model, Shiloach and Vishkin [21] proposed a deterministic $O(\log n)$ algorithm to compute connected components, with $n$ being the size of the largest component. Since then, several other $O(\log n)$ CRCW algorithms have been proposed in [8], [13], [19]. However, since they require concurrent writes, it is not obvious how to translate them to map-reduce efficiently, as the simulation result of [12] applies only to CREW PRAM.

For the CREW PRAM model, Johnson et. al. [9] provided a deterministic $O(\log^{3/2} n)$ time algorithm, which was subsequently improved to $O(\log n)$ by Karger et. al. [11]. These algorithms can be simulated in map-reduce using the result of [12]. However, they require computing all nodes at a distance 2 of each node, which would require $O(n^2)$ communication per map-reduce iteration on a star graph.

Conceptually, our algorithms are most similar to the CRCW PRAM algorithm of Shiloach and Vishkin [21]. That algorithm maintains a connected component as a forest of trees, and repeatedly applies either the operation of pointer doubling (pointing a node to its grand-parent in the tree), or of hooking a tree to another tree. Krishnamurthy et al [13] propose a more efficient implementation, similar to map-reduce, by interleaving local computation on local memory, and parallel computation on shared memory. However, pointer doubling and hooking require concurrent writes, which are hard to implement in map-reduce. Our Hash-to-Min algorithm does conceptually similar but, slightly different, operations in a single map-reduce step.

### B. Map-reduce Model

Google's map-reduce lecture series describes an iterative approach for computing connected components. In each iteration a series of map-reduce steps are used to find and include all nodes adjacent to current connected components. The number of iterations required for this method, and many of its improvements [5], [10], [17], is $O(d)$ where $d$ is the diameter of the largest connected component. These techniques do not scale well for large diameter graphs (such as graphical models where edges represent correlations between variables). Even

for moderate diameter graphs (with $d = 20$), our techniques outperform the $O(d)$ techniques, as shown in the experiments.

Afrati et al [1] propose map-reduce algorithms for computing transitive closure of a graph – a relation containing tuples of pairs of nodes that are in the same connected component. These techniques have a larger communication per iteration as the transitive closure relation itself is quadratic in the size of largest component. Recently, Seidl et al [20] have independently proposed map-reduce algorithms similar to ours, including the use of secondary sorting. However, they do not show the $O(\log n)$ bound on the number of map-reduce rounds.

Table I summarizes the related work comparison and shows that our Hash-Greater-to-Min algorithm is the first map-reduce technique with logarithmic number of iterations and linear communication per iteration.

### C. Bulk Synchronous Parallel (BSP)

In the BSP paradigm, computation is done in parallel by processors in between a series of synchronized point-to-point communication steps. The BSP paradigm is used by recent distributed graph processing systems like Pregel [16] and Giraph [4]. BSP is generally considered more efficient for graph processing than map-reduce as it has less setup and overhead costs for each new iteration. While the algorithmic improvements of reducing number of iterations presented in this paper are applicable to BSP as well, these improvements are of less significance in BSP due to lower overhead of additional iterations.

However, we show that BSP does not necessarily dominate map-reduce for large-scale graph processing (and thus our algorithmic improvements for map-reduce are still relevant and important). We show this by running an interesting experiment in shared grids having congested environments in Sec. VI-C.

The experiment shows that in congested clusters, map-reduce can have a better latency than BSP, since in the latter one needs to acquire and hold machines with a combined memory larger than the graph size. For instance, consider a graph with a billion nodes and ten billion edges. Suppose each node is associated with a state of 256 bytes (e.g., the contents of a web page, or recent updates by a user in a social network, etc.). Then the total memory required would be about 256 GB, say, 256 machines with 1G RAM. In a congested grid waiting for 256 machines could take much longer than running a map-reduce job, since the map-reduce jobs can work with a smaller number of mappers and reducers (say 50-100), and switch in between different MR jobs in the congested environment.

## III. Connected Components on Map-Reduce

In this section, we present map-reduce algorithms for computing connected components. All our algorithms are instantiations of a general map-reduce framework (Algorithm 1), which is parameterized by two functions – a *hashing* function $h$, and a *merging* function $m$ (see line 1 of Algorithm 1). Different choices for $h$ and $m$ (listed in Table II) result in algorithms having very different complexity.

Our algorithm framework maintains a tuple (key, value) for each node $v$ of the graph – key is the node identifier $v$, and the value is a cluster of nodes, denoted $C_v$. The value

1: **Input:** A graph $G = (V, E)$,
*hashing* function $h$
*merging* function $m$, and
*export* function EXPORT
2: **Output:** A set of connected components $C \subset 2^V$
3: Either Initialize $C_v = \{v\}$ Or $C_v = \{v\} \cup nbrs(v)$ depending on the algorithm.
4: **repeat**
5: **mapper for node $v$:**
6: Compute $h(C_v)$, which is a collection of key-value pairs $(u, C_u)$ for $u \in C_v$.
7: Emit all $(u, C_u) \in h(C_v)$.
8: **reducer for node $v$:**
9: Let $\{C_v^{(1)}, \ldots, C_v^{(K)}\}$ denote the set of values received from different mappers.
10: Set $C_v \leftarrow m(\{C_v^{(1)}, \ldots, C_v^{(K)}\})$
11: **until** $C_v$ does not change for all $v$
12: Return $C = \text{EXPORT}(\cup_v \{C_v\})$

**Algorithm 1:** General Map Reduce Algorithm

| |
|---|
| **Hash-Min** emits $(v, C_v)$, and $(u, \{v_{min}\})$ for all nodes $u \in nbrs(v)$. |
| **Hash-to-All** emits $(u, C_v)$ for all nodes $u \in C_v$. |
| **Hash-to-Min** emits $(v_{\min}, C_v)$, and $(u, \{v_{min}\})$ for all nodes $u \in C_v$. |
| **Hash-Greater-to-Min** computes $C_{\geq v}$, the set of nodes in $C_v$ not less than $v$. It emits $(v_{\min}, C_{\geq v})$, and $(u, \{v_{min}\})$ for all nodes $u \in C_{\geq v}$ |

TABLE II
HASHING FUNCTIONS: EACH STRATEGY DESCRIBES THE KEY-VALUE PAIRS EMITTED BY MAPPER WITH INPUT KEY $v$ AND VALUE $C_v$ ($v_{\min}$ DENOTES SMALLEST NODE IN $C_v$)

$C_v$ is initialized as either containing only the node $v$, or containing $v$ and all its neighbors $nbrs(v)$ in $G$, depending on the algorithm (see line 3 of Algorithm 1). The framework updates $C_v$ through multiple mapreduce iterations.

In the map stage of each iteration, the mapper for a key $v$ applies the hashing function $h$ on the value $C_v$ to emit a set of key-value pairs $(u, C_u)$, one for every node $u$ appearing in $C_v$ (see lines 6-7). The choice of hashing function governs the behavior of the algorithm, and we will discuss different instantiations shortly. In the reduce stage, each reducer for a key $v$ aggregates tuples $(v, C_v^{(1)}), \ldots, (v, C_v^{(K)})$ emitted by different mappers. The reducer applies the merging function $m$ over $C_v^{(i)}$ to compute a new value $C_v$ (see lines 9-10). This process is repeated until there is no change to any of the clusters $C_v$ (see line 11). Finally, an appropriate EXPORT function computes the connected components $C$ from the final clusters $C_v$ using one map-reduce round.

**Hash Functions** We describe four hashing strategies in Table II and their complexities in Table III. The first one, denoted Hash-Min, was used in [10]. In the mapper for key $v$, Hash-Min emits key-value pairs $(v, C_v)$ and $(u, \{v_{min}\})$ for all nodes $u \in nbrs(v)$. In other words, it sends the entire cluster $C_v$ to reducer $v$ again, and sends only the minimum node $v_{\min}$ of the cluster $C_v$ to all reducers for nodes $u \in nbrs(v)$. Thus communication is low, but so is rate of convergence, as information spreads only by propagating the minimum node.

On the other hand, Hash-to-All emits key-value pairs $(u, C_v)$ for all nodes $u \in C_v$. In other words, it sends the cluster $C_v$ to all reducers $u \in C_v$. Hence if clusters $C_v$ and $C'_v$ overlap on some node $u$, they will both be sent to reducer of $u$, where they can be merged, resulting in a faster convergence.

| Algorithm | MR Rounds | Communication (per MR step) |
|---|---|---|
| Hash-Min [10] | $d$ | $O(\|V\| + \|E\|)$ |
| Hash-to-All | $\log d$ | $O(n\|V\| + \|E\|)$ |
| Hash-to-Min | $O(\log n)^\star$ | $O(\log n\|V\| + \|E\|)^\star$ |
| Hash-Greater-to-Min | $3 \log n$ | $2(\|V\| + \|E\|)$ |

TABLE III
COMPLEXITY OF DIFFERENT ALGORITHMS ($^\star$ DENOTES RESULTS HOLD FOR ONLY PATH GRAPHS)

But, sending the entire cluster $C_v$ to all reducers $u \in C_v$ results in large quadratic communication cost. To overcome this, Hash-to-Min sends the entire cluster $C_v$ to only one reducer $v_{\min}$, while other reducers are just sent $\{v_{\min}\}$. This decreases the communication cost drastically, while still achieving fast convergence. Finally, the best theoretical complexity bounds can be shown for Hash-Greater-to-Min, which sends out a smaller subset $C_{\geq v}$ of $C_v$. We look at how these functions are used in specific algorithms next.

### A. Hash-Min Algorithm

The Hash-Min algorithm is a version of the Pegasus algorithm [10].[4] In this algorithm each node $v$ is associated with a label $v_{min}$ (i.e., $C_v$ is a singleton set $\{v_{min}\}$) which corresponds to the smallest id amongst nodes that $v$ knows are in its connected component. Initially $v_{\min} = v$ and so $C_v = \{v\}$. It then uses Hash-Min hashing function to propagate its label $v_{min}$ in $C_v$ to all reducers $u \in nbrs(v)$ in every round. On receiving the messages, the merging function $m$ computes the smallest node $v_{\min}^{new}$ amongst the incoming messages and sets $C_v = \{v_{\min}^{new}\}$. Thus a node adopts the minimum label found in its neighborhood as its own label. On convergence, nodes that have the same label are in the same connected component. Finally, the connected components are computed by the following EXPORT function: return sets of nodes grouped by their label.

*Theorem 3.1 (Hash-Min [10]):* Algorithm Hash-Min correctly computes the connected components of $G = (V, E)$ using $O(|V| + |E|)$ communication and $O(d)$ map-reduce rounds.

### B. Hash-to-All Algorithm

The Hash-to-All algorithm initializes each cluster $C_v = \{v\} \cup \{nbrs(v)\}$. Then it uses Hash-to-All hashing function to send the entire cluster $C_v$ to all reducers $u \in C_v$. On receiving the messages, merge function $m$ updates the cluster by taking the union of all the clusters received by the node. More formally, if the reducer at $v$ receives clusters $C_v^{(1)}, \ldots, C_v^{(K)}$, then $C_v$ is updated to $\cup_{i=1}^{K} C_v^{(i)}$.

We can show that after $\log d$ map-reduce rounds, for every $v$, $C_v$ contains all the nodes in the connected component containing $v$. Hence, the EXPORT function just returns the distinct sets in $C$ (using one map-reduce step).

*Theorem 3.2 (Hash-to-All):* Algorithm Hash-to-All correctly computes the connected components of $G = (V, E)$ using $O(n|V|+|E|)$ communication per round and $\log d$ map-reduce rounds, where $n$ is the size of the largest component and $d$ the diameter of $G$.

[4][10] has additional optimizations that do not change the asymptotic complexity. We do not describe them here.

*Proof:* We can show using induction that after $k$ map-reduce steps, every node $u$ that is at a distance $\leq 2^k$ from $v$ is contained in $C_v$. Initially this is true, since all neighbors are part of $C_v$. Again, for the $k + 1^{st}$ step, $u \in C_w$ for some $w$ and $w \in C_v$ such that distance between $u, w$ and $w, v$ is at most $2^k$. Hence, for every node $u$ at a distance at most $2^{k+1}$ from $v$, $u \in C_v$ after $k + 1$ steps. Proof for communication complexity follows from the fact that each node is replicated at most $n$ times. ■

### C. Hash-to-Min Algorithm

While the Hash-to-All algorithm computes the connected components in a smaller number of map-reduce steps than Hash-Min, the size of the intermediate data (and hence the communication) can become prohibitively large for even sparse graphs having large connected components. We now present Hash-to-Min, a variation on Hash-to-All, that we show finishes in at most $4 \log n$ steps for path graphs. We also show that in practice it takes at most $2 \log d$ rounds and linear communication cost per round (see Section VI),where $d$ is the diameter of the graph.

The Hash-to-Min algorithm initializes each cluster $C_v = \{v\} \cup \{nbrs(v)\}$. Then it uses Hash-to-Min hash function to send the entire cluster $C_v$ to reducer $v_{min}$, where $v_{min}$ is the smallest node in the cluster $C_v$, and $\{v_{min}\}$ to all reducers $u \in C_v$. The merging function $m$ works exactly like in Hash-to-All: $C_v$ is the union of all the nodes appearing in the received messages. We explain how this algorithm works by an example.

*Example 3.3:* Consider an intermediate step where clusters $C_1 = \{1, 2, 4\}$ and $C_5 = \{3, 4, 5\}$ have been associated with keys 1 and 5. We will show how these clusters are merged in both Hash-to-All and Hash-to-Min algorithms.

In the Hash-to-All scheme, the mapper at 1 sends the entire cluster $C_1$ to reducers 1, 2, and 4, while mapper at 5 sends $C_5$ to reducers 3, 4, and 5. Therefore, on reducer 4, the entire cluster $C_4 = \{1, 2, 3, 4, 5\}$ is computed by the merge function. In the next step, this cluster $C_4$ is sent to all the five reducers.

In the Hash-to-Min scheme, the mapper at 1 sends $C_1$ to reducer 1, and $\{1\}$ to reducer 2 and 4. Similarly, the mapper at 5 sends $C_5$ to reducer 3, and $\{3\}$ to reducer 4 and 5. So reducer 4 gets $\{1\}$ and $\{3\}$, and therefore computes the cluster $C_4 = \{1, 3\}$ using the merge function.

Now, in the second round, the mapper at 4, has 1 as the minimum node of the cluster $C_4 = \{1, 3\}$. Thus, it sends $\{1\}$ to reducer 3, which already has the cluster $C_2 = \{3, 4, 5\}$. Thus after the second round, the cluster $C_3 = \{1, 3, 4, 5\}$ is formed on reducer 3. Since 1 is the minimum for $C_3$, the mapper at 3 sends $C_3$ to reducer 1 in the third round. Hence after the end of third round, reducer 1 gets the entire cluster $\{1, 2, 3, 4, 5\}$.

Note in this example that Hash-to-Min required three map-reduce steps; however, the intermediate data transmitted is lower since entire clusters $C_1$ and $C_2$ were only sent to their minimum element's reducer (1 and 3, resp).

As the example above shows, unlike Hash-to-All, at the end of Hash-to-Min, all reducers $v$ are not guaranteed to contain in $C_v$ the connected component they are part of. In fact, we can

show that the reducer at $v_{min}$ contains all the nodes in that component, where $v_{min}$ is the smallest node in a connected component. For other nodes $v$, $C_v = \{v_{\min}\}$. Hence, EXPORT outputs only those $C_v$ such that $v$ is the smallest node in $C_v$.

*Theorem 3.4 (Hash-to-Min Correctness):* At the end of algorithm Hash-to-Min, $C_v$ satisfies the following property: If $v_{\min}$ is the smallest node of a connected component $C$, then $C_{v_{\min}} = C$. For all other nodes $v$, $C_v = \{v_{\min}\}$.

*Proof:* Consider any node $v$ such that $C_v$ contains $v_{\min}$. Then in the next step, mapper at $v$ sends $C_v$ to $v_{\min}$, and only $\{v_{\min}\}$ to $v$. After this iteration, $C_v$ will always have $v_{min}$ as the minimum node, and the mapper at $v$ will always send its cluster $C_v$ to $v_{min}$. Now at some point of time, all nodes $v$ in the connected component $C$ will have $v_{\min} \in C_v$ (this follows from the fact that min will propagate at least one hop in every iteration just like in Theorem 3.1). Thus, every mapper for node $v$ sends its final cluster to $v_{\min}$, and only retains $v_{\min}$. Thus at convergence $C_{v_{min}} = C$ and $C_v = \{v_{min}\}$. ∎

*Theorem 3.5 (Hash-to-Min Communication):* Algorithm takes $O(k{\cdot}(|V|{+}|E|))$ expected communication per round, where $k$ is the total number of rounds. Here expectation is over the random choices of the node ordering.

*Proof:* Note that the total communication in any step equals the total size of all $C_v$ in the next round. Let $n_k$ denote the size of this intermediate after $k$ rounds. That is, $n_k = \sum_v C_v$. We show by induction that $n_k = O(k \cdot (|V| + |E|))$.

First, $n_0 = \sum_v C_v^0 \leq 2(|V|{+}|E|)$, since each node contains itself and all its neighbors. In each subsequent round, a node $v$ is present in $C_u$, for all $u \in C_v$. Then $v$ is sent to a different cluster in one of two ways:

- If $v$ is the smallest node in $C_u$, then $v$ is sent to all nodes in $C_u$. Due to this, $v$ gets replicated to $|C_u|$ different clusters. However, this happens with probability $1/|C_u|$.
- If $v$ is not the smallest node, then $v$ is sent to the smallest node of $C_u$. This happens with probability $1 - 1/|C_u|$. Moreover, once $v$ is not the smallest for a cluster, it will never become the smallest node; hence it will never be replicated more that once.

From the above two facts, on expectation after one round, the node $v$ is sent to $s_1 = |C_v^0|$ clusters as the smallest node and to $m_1 = |C_v^0|$ clusters as not the smallest node. After two rounds, the node $v$ is additionally sent to $s_2 = |C_v^0|$, $m_2 = |C_v^0|$, in addition to the $m_1$ clusters. Therefore, after $k$ rounds, $n_k = O(k \cdot (|V| + |E|))$. ∎

Next we show that on a path graph, Hash-to-Min finishes in $4 \log n$. The proof is rather long, and due to space constraints appears in Sec. A of the Appendix.

*Theorem 3.6 (Hash-to-Min Rounds):* Let $G = (V, E)$ be a path graph (i.e. a tree with only nodes of degree 2 or 1). Then, Hash-to-Min correctly computes the connected component of $G = (V, E)$ in $4 \log n$ map-reduce rounds.

Although, Theorem 3.6 works only for path graphs, we conjecture that Hash-to-Min finishes in $2 \log d$ rounds on all inputs, with $O(|V| + |E|)$ communication per round. Our experiments (Sec. VI) seem to validate this conjecture.

### D. Hash-Greater-to-Min Algorithm

Now we describe the Hash-Greater-to-Min algorithm that has the best theoretical bounds: $3 \log n$ map-reduce rounds with high probability and $2(|V|{+}|E|)$ communication complexity per round in the worst-case. In Hash-Greater-to-Min algorithm, the clusters $C_v$ are again initialized as $\{v\}$. Then Hash-Greater-to-Min algorithm runs two rounds using Hash-Min hash function, followed by a round using Hash-Greater-to-Min hash function, and keeps on repeating these three rounds until convergence.

In a round using Hash-Min hash function, the entire cluster $C_v$ is sent to reducer $v$ and $v_{\min}$ to all reducers $u \in nbrs(v)$. For, the merging function $m$ on machine $m(v)$, the algorithm first computes the minimum node among all incoming messages, and then adds it to the message $C(v)$ received from $m(v)$ itself. More formally, say $v_{\min}^{new}$ is the smallest nodes among all the messages received by $u$, then $C_{new}(v)$ is updated to $\{v_{\min}^{new}\} \cup \{C(v)\}$.

In a round using Hash-Greater-to-Min hash function, the set $C_{\geq v}$ is computed as all nodes in $C_v$ not less than $v$. This set is sent to reducer $v_{\min}$, where $v_{\min}$ is the smallest node in $C(v)$, and $\{v_{\min}\}$ is sent to all reducers $u \in C_{\geq v}$. The merging function $m$ works exactly like in Hash-to-All: $C(v)$ is the union of all the nodes appearing in the received messages. We explain this process by the following example.

*Example 3.7:* Consider a path graph with $n$ edges $(1, 2)$, $(2, 3)$, $(3, 4)$, and so on. We will now show three rounds of Hash-Greater-to-Min.

In Hash-Greater-to-Min algorithm, the clusters are initialized as $C_i = \{i\}$ for $i \in [1, n]$. In the first round, the Hash-Min function will send $\{i\}$ to reducers $i-1$, $i$, and $i+1$. So each reducer $i$ will receive messages $\{i-1\}$, $\{i\}$ and $\{i+1\}$, and aggregation function will add the incoming minimum, $i-1$, to the previous $C_i = \{i\}$.

Thus in the second round, the clusters are $C_1 = \{1\}$ and $C_i = \{i-1, i\}$ for $i \in [2, n]$. Again Hash-Min will send the minimum node $\{i-1\}$ of $C_i$ to reducers $i-1$, $i$, and $i+1$. Again merging function would be used. At the end of second step, the clusters are $C_1 = \{1\}$, $C_2 = \{1, 2\}$, $C_i = \{i-2, i-1, i,\}$ for $i \in [3, n]$.

In the third round, Hash-Greater-to-Min will be used. This is where interesting behavior is seen. Mapper at 2 will send its $C_{\geq(}2) = \{2\}$ to reducer 1. Mapper at 3 will send its $C_{\geq(}3) = \{3\}$ to reducer 1. Note that $C_{\geq(}3)$ does not include 2 even though it appears in $C_3$ as $2 < 3$. Thus we save on sending redundant messages from mapper 3 to reducer 1 as 2 has been sent to reducer 1 from mapper 2. Similarly, mapper at 4 sends $C_{\geq(}4) = \{4\}$ to reducer 2, and mapper 5 sends $C_{\geq(}5) = \{5\}$ to reducer 3, etc. Thus we get the sets, $C_1 = \{1, 2, 3\}$, $C_2 = \{1, 2, 4\}$, $C_3 = \{1, 3, 6\}$, $C_4 = \{4, 5, 6\}$, and so on.

The analysis of the Hash-Greater-to-Min algorithm relies on the following lemma.

*Lemma 3.8:* Let $v_{\min}$ be any node. Denote $GT(v_{\min})$ the set of all nodes $v$ for which $v_{\min}$ is the smallest node in $C_v$ after Hash-Greater-to-Min algorithm converges. Then $GT(v_{\min})$ is precisely the set $C_{\geq(}v_{\min})$.

Note that in the above example, after 3 rounds of Hash-Greater-to-Min, $GT(2)$ is $\{2, 4\}$ and $C_{\geq(}2)$ is also $\{2, 4\}$.

We now analyze the performance of this algorithm. The proof is based on techniques introduced in [8], [11], and

omitted here due to lack of space. The proof appears in the Appendix.

*Theorem 3.9 (Complexity):* Algorithm Hash-Greater-to-Min correctly computes the connected components of $G = (V, E)$ in expected $3 \log n$ map-reduce rounds (expectation is over the random choices of the node ordering) with $2(|V|+|E|)$ communication per round in the worst case.

## IV. Scaling the Hash-to-Min Algorithm

Hash-to-Min and Hash-Greater-to-Min complete in less number of rounds than Hash-Min, but as currently described, they require that every connected component of the graph fit in memory of a single reducer. We now describe a more robust implementation for Hash-to-Min, which allows handling arbitrarily large connected components. We also describe an extension to do load balancing. Using this implementation, we show in Section VI examples of social network graphs that have small diameter and extremely large connected components, for which Hash-Greater-to-Min runs out of memory, but Hash-to-Min still works efficiently.

### A. Large Connected Components

We address the problem of larger than memory connected components, by using secondary sorting in map-reduce, which allows a reducer to receive values for each key in a sorted order. Note this sorting is generally done in map-reduce to keep the keys in a reducer in sorted order, and can be extended to sort values as well, at no extra cost, using composite keys and custom partitioning [15].

To use secondary sorting, we represent a connected component as follows: if $C_v$ is the cluster at node $v$, then we represent $C_v$ as a graph with an edge from $v$ to each of the node in $C_v$. Recall that each iteration of Hash-to-Min is as follows: for hashing, denoting $v_{min}$ as the min node in $C_v$, the mapper at $v$ sends $C_v$ to reducer $v_{min}$, and $\{v_{min}\}$ to all reducers $u$ in $C_v$. For merging, we take the union of all incoming clusters at reducer $v$.

Hash-to-Min can be implemented in a single map-reduce step. The hashing step is implemented by emitting in the mapper, key-value pairs, with key as $v_{min}$, and values as each of the nodes in $C_v$, and conversely, with key as each node in $C_v$, and $v_{min}$ as the value. The merging step is implemented by collecting all the values for a key $v$ and removing duplicates.

To remove duplicates without loading all values in the memory, we use secondary sorting to guarantee that the values for a key are provided to the reducer in a sorted order. Then the reducer can just make a single pass through the values to remove duplicates, without ever loading all of them in memory, since duplicates are guaranteed to occur adjacent to each other. Furthermore, computing the minimum node $v_{min}$ is also trivial as it is simply the first value in the sorted order.

### B. Load Balancing Problem

Even though Hash-to-Min can handle arbitrarily large graphs without failure (unlike Hash-Greater-to-Min), it can still suffer from data skew problems if some connected components are large, while others are small. We handle this problem by tweaking the algorithm as follows. If a cluster $C_v$ at machine $v$ is larger than a predefined threshold, we send all nodes $u \leq v$ to reducer $v_{min}$ and $\{v_{min}\}$ to all reducers $u \leq v$, as done in Hash-to-Min. However, for nodes $u > v$, we send them to reducer $v$ and $\{v\}$ to reducer $u$. This ensures that reducer $v_{min}$ does not receive too many nodes, and some of the nodes go to reducer $v$ instead, ensuring balanced load.

This modified Hash-to-Min is guaranteed to converge in at most the number of steps as the standard Hash-to-Min converges. However, at convergence, all nodes in a connected component are not guaranteed to have the minimum node $v_{min}$ of the connected component. In fact, they can have as their minimum, a node $v$ if the cluster at $v$ was bigger than the specified threshold. We can then run standard Hash-to-Min, on the modified graph over nodes that correspond to cluster ids, and get the final output. Note that this increases the number of rounds by at most 2, as after load-balanced Hash-to-Min converges, we use the standard Hash-to-Min.

*Example 4.1:* If the specified threshold is 1, then the modified algorithm converges in exactly one step, returning clusters equal to one-hop neighbors. If the specified threshold is $\infty$, then the modified algorithm converges to the same output as the standard one, i.e. it returns connected components. If the specified threshold is somewhere in between (for our experiments we choose it to 100,000 nodes), then the output clusters are subsets of connected components, for which no cluster is larger than the threshold.

1: **Input:** Weighted graph $G = (V, E, w)$,
   stopping criterion $Stop$.
2: **Output:** A clustering $C \subseteq 2^V$.
3: Initialize clustering $C \leftarrow \{\{v\} | v \in V\}$;
4: **repeat**
5: Find the closest pair of clusters $C_1, C_2$ in $C$ (as per $d$);
6: Update $C \leftarrow C - \{C_1, C_2\} \cup \{C_1 \cup C_2)\}$;
7: **until** $C$ does not change or $Stop(C)$ is true
8: Return $C$

**Algorithm 2:** Centralized single linkage clustering

## V. Single Linkage Agglomerative Clustering

To the best of our knowledge, no map-reduce implementation exists for single linkage clustering that completes in $o(n)$ map-reduce steps, where $n$ is the size of the largest cluster. We now present two map-reduce implementations for the same, one using Hash-to-All that completes in $O(\log n)$ rounds, and another using Hash-to-Min that we conjecture to finish in $O(\log d)$ rounds.

For clustering, we take as input a weighted graph denoted as $G = (V, E, w)$, where $w : E \rightarrow [0, 1]$ is a weight function on edges. An output cluster $C$ is any set of nodes, and a clustering $\mathcal{C}$ of the graph is any set of clusters such that each node belongs to exactly one cluster in $C$.

### A. Centralized Algorithm:

Algorithm 2 shows the typical bottom up centralized algorithm for single linkage clustering. Initially, each node is its own cluster. Define the distance between two clusters $C_1, C_2$ to be the minimum weight of an edge between the two clusters;

1: **Input:** Weighted graph $G = (V, E, w)$, stopping criterion $Stop$.
2: **Output:** A clustering $C \subseteq 2^V$.
3: Initialize $C = \{\{v\} \cup nbrs(v) | v \in V\}$.
4: **repeat**
5: **Map:** Use Hash-to-All or Hash-to-Min to hash clusters.
6: **Reduce:** Merge incoming clusters.
7: **until** $C$ does not change or $Stop(C)$ is true
8: Split clusters in $C$ merged incorrectly in the final iteration.
9: Return $C$

**Algorithm 3:** Distributed single linkage clustering

i.e.,

$$d(C_1, C_2) \ = \ \min_{e=(u,v), u \in C_1, v \in C_2} w(e)$$

In each step, the algorithm picks the two closest clusters and merges them by taking their union. The algorithm terminates either when the clustering does not change, or when a *stopping* condition, $Stop$, is reached. Typical stopping conditions are *threshold* stopping, where the clustering stops when the closest distance between any pair of clusters is larger than a threshold, and *cluster size* stopping condition, where the clustering stops when the merged cluster in the most recent step is too large.

Next we describe a map-reduce algorithm that simulates the centralized algorithm, i.e., outputs the same clustering. If there are two edges in the graph having the exact same weight, then single linkage clustering might not be unique, making it impossible to prove our claim. Thus, we assume that ties have been broken arbitrarily, perhaps, by perturbing the weights slightly, and thus no two edges in the graph have the same weight. Note that this step is required only for simplifying our proofs, but not for the correctness of our algorithm.

### B. Map-Reduce Algorithm

Our Map-Reduce algorithm is shown in Algorithm 3. Intuitively, we can compute single-linkage clustering by first computing the connected components (since no cluster can lie across multiple connected components), and then splitting the connected components into appropriate clusters. Thus, Algorithm 3 has the same map-reduce steps as Algorithm 1, and is implemented either using Hash-to-All or Hash-to-Min.

However, in general, clusters, defined by the stopping criteria, $Stop$, may be much smaller than the connected components. In the extreme case, the graph might be just one giant connected component, but the clusters are often small enough that they individually fit in the memory of a single machine. Thus we need a way to check and stop execution as soon as clusters have been computed. We do this by evaluating $Stop(C)$ after each iteration of map-reduce. If $Stop(C)$ is false, a new iteration of map-reduce clustering is started. If $Stop(C)$ is true, then we stop iterations.

While the central algorithm can implement any stopping condition, checking an arbitrary predicate in a distributed setting can be difficult. Furthermore, while the central algorithm merges one cluster at a time, and then evaluates the stopping condition, the distributed algorithm evaluates stopping condition only at the end of a map-reduce iteration. This means that some reducers can merge clusters incorrectly in the last map-reduce iteration. We describe next how to stop the map-reduce clustering algorithm, and split incorrectly merged clusters.

### C. Stopping and Splitting Clusters

It is difficult to evaluate an arbitrary stopping predicate in a distributed fashion using map-reduce. We restrict our attention to a restricted yet frequently used class of local monotonic stopping criteria, which is defined below.

*Definition 5.1 (Monotonic Criterion):* $Stop$ is monotone if for every clusterings $\mathcal{C}$, $\mathcal{C}'$, if $\mathcal{C}'$ refines $\mathcal{C}$ (i.e, $\forall C \in \mathcal{C} \Rightarrow \exists C' \in \mathcal{C}', \ C \subseteq C'$), then $Stop(\mathcal{C}) = 1 \Rightarrow Stop(\mathcal{C}') = 1$.

Thus monotonicity implies that stopping predicate continues to remain true if some clusters are made smaller. Virtually every stopping criterion used in practice is monotonic. Next we define the assumption of locality, which states that stopping criterion can be evaluated locally on each cluster individually.

*Definition 5.2 (Local Criterion):* $Stop$ is local if there exists a function $Stop_{local} : 2^V \to \{0, 1\}$ such that $Stop(\mathcal{C}) = 1$ iff $Stop_{local}(C) = 1$ for all $C \in \mathcal{C}$.

Examples of local stopping criteria include distance-threshold (stop merging clusters if their distance becomes too large) and size-threshold (stop merging if the size of a cluster becomes too large). Example of *non*-local stopping criterion is to stop when the total number of clusters becomes too high.

If the stopping condition is local and monotonic, then we can compute it efficiently in a single map-reduce step. To explain how, we first define some notations. Given a cluster $C \subseteq V$, denote $G_C$ the subgraph of $G$ induced over nodes $C$. Since $C$ is a cluster, we know $G_C$ is connected. We denote $tree(C)$ as the[5] minimum weight spanning tree of $G_C$, and $split(C)$ as the pair of clusters $C_L, C_R$ obtained by removing the edge with the maximum weight in $tree(C)$. Intuitively, $C_L$ and $C_R$ are the clusters that get merged to get $C$ in the centralized single linkage clustering algorithm. Finally, denote $nbrs(C)$ the set of clusters closest to $C$ by the distance metric $d$, i.e. if $C_1 \in nbrs(C)$, then for every other cluster $C_2$, $d(C, C_2) > d(C, C_1)$.

We also define the notion of core and minimal core decomposition as follows.

*Definition 5.3 (Core):* A singleton cluster is always a core. Furthermore, any cluster $C \subseteq V$ is a core if its split $C_L, C_R$ are both cores and closest to each other, i.e. $C_L \in nbrs(C_R)$ and $C_R \in nbrs(C_L)$.

*Definition 5.4 (Minimal core decomposition):* Given a cluster $C$ its minimal core decomposition, $MCD(C)$, is a set of cores $\{C_1, C_2, \ldots, C_l\}$ such that $\cup_i C_i = C$ and for every core $C' \subseteq C$ there exists a core $C_i$ in the decomposition for which $C' \subseteq C_i$.

Intuitively, a cluster $C$ is a core, if it is a valid, i.e., it is a subset of some cluster $C'$ in the output of the centralized single linkage clustering algorithm, and $MCD(C)$ finds the largest cores in $C$, i.e. cores that cannot be merged with any other node in $C$ and still be cores.

**Computing MCD** We give in Algorithm 4, a method to find the minimal core decomposition of a cluster. It checks whether the input cluster is a core. Otherwise it computes cluster splits $C_l$, and $C_r$ and computes their MCD recursively.

[5]The tree is unique because of unique edge weights

1: **Input:** Cluster $C \subseteq V$.
2: **Output:** A set of cores $\{C_1, C_2, \ldots, C_l\}$ corresponding to $MCD(C)$.
3: If $C$ is a core, return $\{C\}$.
4: Construct the spanning tree $T_C$ of $C$, and compute $C_L, C_R$ to be the cluster split of $C$.
5: Recursively compute $MCD(C_L)$ and $MCD(C_R)$.
6: Return $MCD(C_L) \cup MCD(C_R)$

**Algorithm 4:** Minimal Core Decomposition $MCD$

1: **Input:** Stopping predicate $Stop$, Clustering $C$.
2: **Output:** $Stop(C)$
3: For each cluster $C \in C$, compute $MCD(C)$. (performed in reduce of calling of Algo. 3)
4: **Map:** Run Hash-to-All on cores, i.e, hash each core $C_i \in MCD(C)$ to all machines $m(u)$ for $u \in C_i$
5: **Reducer for node v:** Of all incoming cores, pick the largest core, say, $C_v$, and compute $Stop_{local}(C_v)$.
6: Return $\wedge_{v \in V} Stop_{local}(C_v)$

**Algorithm 5:** Stopping Algorithm

Note that this algorithm is centralized and takes as input a single cluster, which we assume fits in the memory of a single machine (unlike connected components, graph clusters are rather small).

**Stopping Algorithm** Our stopping algorithm, shown in Algorithm 5, is run after each map-reduce iteration of Algorithm 3. It takes as input the clustering $C$ obtained after the map-reduce iteration of Algorithm 3 . It starts by computing the minimal core decomposition, $MCD(C)$, of each cluster $C$ in $C$. This computation can be performed during the reduce step of the pervious map-reduce iteration of Algorithm 3. Then, it runs a map-reduce iteration of its own. In the map step, using Hash-to-All, each core $C_i$ is hashed to all machines $m(u)$ for $u \in C_i$. In reducer, for machine $m(v)$, we pick the incoming core with largest size, say $C_v$. Since $Stop$ is local, there exists a local function $Stop_{local}$. We compute $Stop_{local}(C_v)$ to determine whether to stop processing this core further. Finally, the algorithm stops if all the cores for nodes $v$ in the graph are stopped.

**Splitting Clusters** If the stopping algorithm (Algorithm 5) returns true, then clustering is complete. However, some clusters could have merged incorrectly in the final map-reduce iteration done before the stopping condition was checked. Our recursive splitting algorithm, Algorithm 6, correctly splits such a cluster $C$ by first computing the minimal core decomposition, $MCD(C)$. Then it checks for each core $C_i \in MCD(C)$ that its cluster splits $C_l$ and $C_r$ could have been merged by ensuring that both $Stop_{local}(C_l)$ and $Stop_{local}(C_r)$ are false. If that is the case, then core $C_i$ is valid and added to the output, otherwise the clusters $C_l$ and $C_r$ should not have been merged, and $C_i$ is split further.

### D. Correctness & Complexity Results

We first show the correctness of Algorithm 3. For that we first show the following lemma about the validity of cores.

*Lemma 5.5 (Cores are valid):* Let $C_{central}$ be the output of Algorithm 2, and $C$ be any core (defined according to Def. 5.3)

1: **Input:** Incorrectly merged cluster $C$ w.r.t $Stop_{local}$.
2: **Output:** Set $S$ of correctly split clusters in $C$.
3: Initialize $S = \{\}$.
4: **for** $C_i$ in $MCD(C)$ **do**
5: Let $C_l$ and $C_r$ be the cluster splits of $C_i$.
6: **if** $Stop_{local}(C_l)$ and $Stop_{local}(C_r)$ are false **then**
7: $S = S \cup C_i$.
8: **else**
9: $S = S \cup Split(C_i)$
10: **end if**
11: **end for**
12: Return $S$.

**Algorithm 6:** Recursive Splitting Algorithm Split

such that its clusters splits $C_l$, $C_r$ have both $Stop_{local}(C_l)$ and $Stop_{local}(C_r)$ as false. Then $C$ is valid, i.e. Algorithm 2 does compute $C$ some time during its execution, and there exists a cluster $C_{central}$ in $C_{central}$ such that $C \subseteq C_{central}$.

*Proof:* The proof uses induction. For the base case, note that any singleton core is obviously valid. Now assume that $C$ has cluster splits $C_l$ and $C_r$, which by induction hypothesis, are valid. Then we show that $C$ is also valid. Since $Stop_{local}(C_l)$ and $Stop_{local}(C_r)$ are false for the cluster splits of $C$, they do get merged with some clusters in Algorithm 2. Furthermore, by definition of a core, $C_l, C_r$ are closest to each other, hence they actually get merged with each other.Thus $C = C_l \cup C_r$ is constructed some during execution of Algorithm 2, and there exists a cluster $C_{central}$ in its output that contains $C_l \cup C_r = C$, completing the proof. ■

Next we show the correctness of Algorithm 3. Due to lack of space, the proof is omitted and appears in the Appendix.

*Theorem 5.6 (Correctness):* The distributed Algorithm 3 simulates the centralized Algorithm 2, i.e., it outputs the same clustering as Algorithm 2.

Next we state the complexity result for single linkage clustering. We omit the proof as it is very similar to that of the complexity result for connected components.

*Theorem 5.7 (Single-linkage Runtime):* If Hash-to-All is used in Algorithm 3, then it finishes in $O(\log n)$ map-reduce iterations and $O(n|V| + |E|)$ communication per iteration, where $n$ denotes the size of the largest cluster.

We also conjecture that if Hash-to-Min is used in Algorithm 3, then it finishes in $O(\log d)$ steps.

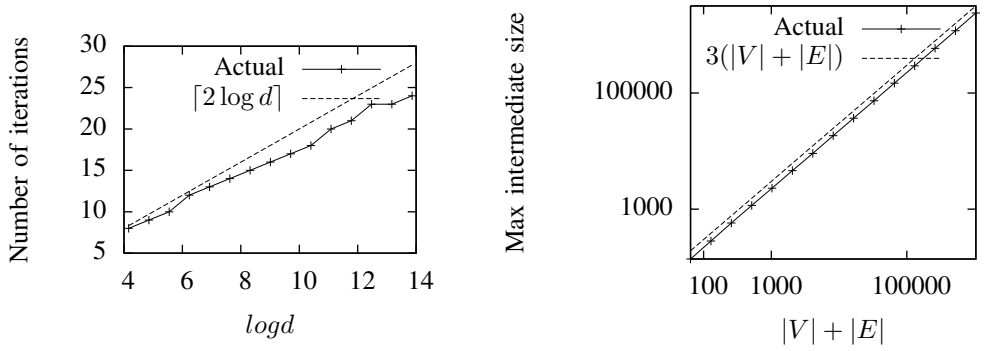
## VI. Experiments

In this section, we experimentally analyze the performance of the proposed algorithms for computing connected components of a graph. We also evaluate the performance of our agglomerative clustering algorithms.

**Datasets:** To illustrate the properties of our algorithms we use both synthetic and real datasets.

- *Movie*: The movie dataset has movie listings collected from Y! Movies[6] and DBpedia[7]. Edges between two listings correspond to movies that are duplicates of one another; these edges are output by a pairwise matcher algorithm. Listings maybe duplicates from the same or

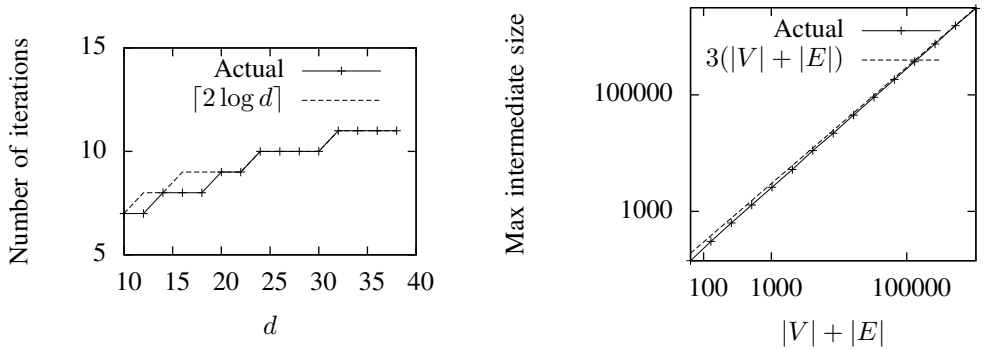
[6]http://movies.yahoo.com/movie/*/info
[7]http://dbpedia.org/




(a) # of Iterations (max over 10 runs)

(b) Largest intermediate data size (average 10 runs)

Fig. 1. Analysis of Hash-to-Min on a path graph with random node orderings




(a) # of Iterations (max over 10 runs)

(b) Largest intermediate data size (average 10 runs)

Fig. 2. Analysis of Hash-to-Min on a tree graph with random node orderings

different source, hence the sizes of connected components vary. The number of nodes $|V| = 431,221$ (nearly 430K) and the number of edges $|E| = 889,205$ (nearly 890K). We also have a sample of this graph (238K nodes and 459K edges) with weighted edges, denoted *MovieW*, which we use for agglomerative clustering experiments.

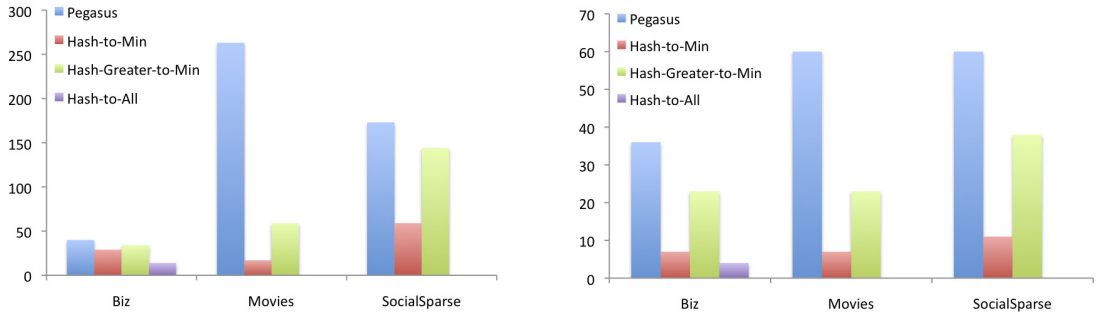
- *Biz*: The biz dataset has business listings coming from two overlapping feeds that are licensed by a large internet company. Again edges between two businesses correspond to business listings that are duplicates of one another. Here, $|V| = 10,802,777$ (nearly 10.8M) and $|E| = 10,197,043$ (nearly 10.2M). We also have a version of this graph with weighted edges, denoted *BizW*, which we use for agglomerative clustering experiments.
- *Social*: The *Social* dataset has social network edges between users of a large internet company. *Social* has $|V| = 58552777$ (nearly 58M) and $|E| = 156355406$ (nearly 156M). Since social network graphs have low diameter, we remove a random sample of edges, and generate *SocialSparse*. With $|E| = 15,638,853$ (nearly 15M), *SocialSparse* graph is more sparse, but has much higher diameter than *Social*.
- *Twitter*: The *Twitter* dataset (collected by Cha et al [3]) has follower relationship between twitter users. *Twitter* has $|V| = 42069704$ (nearly 42M) and $|E| = 1423194279$ (nearly 1423M). Again we remove a random sample of edges, and generate a more sparse graph, *TwitterSparse*, with $|E| = 142308452$ (nearly 142M).
- *Synth*: We also synthetically generate graphs of a varying diameter and sizes in order to better understand the properties of the algorithms.

### A. Connected Components

**Algorithms:** We compare Hash-Min, Hash-Greater-to-Min, Hash-to-All, Hash-to-Min and its load-balanced version Hash-to-Min* (Section IV). For Hash-Min, we use the open-source Pegasus implementation[8], which has several optimizations over the Hash-Min algorithm. We implemented all other algorithms in Pig[9] on Hadoop[10]. There is no native support for iterative computation on Pig or Hadoop map-reduce. We implement one iteration of our algorithm in Pig and drive a loop using a python script. Implementing the algorithms on iterative map-reduce platforms like HaLoop [2] and Twister [7] is an interesting avenue for future work.

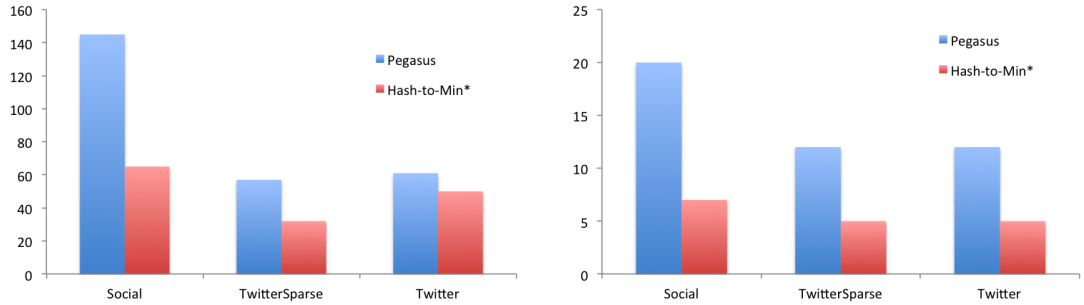
*1) Analyzing Hash-to-Min on Synthetic Data:* We start by experimentally analyzing the rounds complexity and space requirements of Hash-to-Min. We run it on two kinds of synthetic graphs: paths and complete binary trees. We use synthetic data for this experiment so that we have explicit

[8]http://www.cs.cmu.edu/~pegasus/
[9]http://pig.apache.org/
[10]http://hadoop.apache.org




(a) Group I:Runtimes (in minutes)

(b) Group I: # of Map-Reduce jobs




(c) Group II: Runtimes (in minutes)

(d) Group II: # of Map-Reduce jobs

Fig. 3. Comparison of Pegasus and our algorithms on real datasets.

| Input | $\|V\|$ | $\|E\|$ | $n$ | Pegasus | | Hash-to-Min | | Hash-Greater-to-Min | | Hash-to-All | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | # MR jobs | Time | # MR jobs | Time | # MR jobs | Time | # MR jobs | Time |
| *Biz* | 10.8M | 10.1M | 93 | 36 | 40 | 7 | 29 | 23 | 34 | 4 | **14** |
| *Movie* | 430K | 890K | 17K | 60 | 263 | 7 | **17** | 23 | 59 | DNF | DNF |
| *SocialSparse* | 58M | 15M | 2.9M | 60 | 173 | 11 | **59** | 38 | 144 | DNF | DNF |

TABLE IV
COMPARISON OF PEGASUS, HASH-TO-MIN, HASH-GREATER-TO-MIN, AND HASH-TO-ALL ON THE GROUP I DATASETS. TIME IS AVERAGED OVER 4 RUNS AND ROUNDED TO MINUTES. OPTIMAL TIMES APPEAR IN BOLD: IN ALL CASES EITHER HASH-TO-MIN OR HASH-TO-ALL IS OPTIMAL.

| Input | $\|V\|$ | $\|E\|$ | $n$ | Pegasus | | Hash-to-Min* | |
|---|---|---|---|---|---|---|---|
| | | | | # MR jobs | Time | # MR jobs | Time |
| *Social* | 58M | 156M | 36M | 20 | 145 | 7 | **65** |
| *TwitterSparse* | 42M | 142M | 24M | 12 | 57 | 5 | **32** |
| *Twitter* | 42M | 1423M | 42M | 12 | 61 | 5 | **50** |

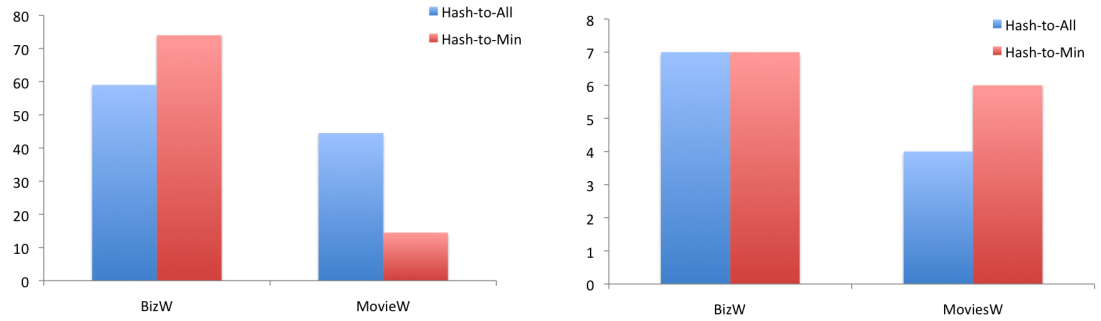
TABLE V
COMPARISON OF PEGASUS AND THE HASH-TO-MIN* ALGORITHM ON GROUP II DATASETS. TIME IS AVERAGED OVER 4 RUNS AND ROUNDED TO MINUTES. OPTIMAL TIMES APPEAR IN BOLD: IN ALL CASES, HM* IS OPTIMAL.

control over parameters $d$, $|V|$, and $|E|$. Later we report the performance of Hash-to-Min on real data as well. We use path graphs since they have largest $d$ for a given $|V|$ and complete binary trees since they give a very small $d = \log |V|$.

For measuring space requirement, we measure the largest intermediate data size in any iteration of Hash-to-Min. Since the performance of Hash-to-Min depends on the random ordering of the nodes chosen, we choose 10 different random orderings for each input. For number of iterations, we report the worst-case among runs on all random orderings, while for intermediates data size we average the maximum intermediate data size over all runs of Hash-to-Min. This is to verify our conjecture that number of iterations is $2 \log d$ in the worst-case (independent of node ordering) and intermediate space complexity is $O(|V|+|E|)$ in expectation (over possible node orderings).

For path graphs, we vary the number of nodes from 32 ($2^5$) to $524,288$ ($2^{19}$). In Figure 1(a), we plot the number of iterations (worst-case over 10 runs on random orderings) with respect to $\log d$. Since the diameter of a path graph is equal to number of nodes, $d$ varies from 32 to $524,288$ as well. As conjectured the plot is linear and always lies below the line corresponding to $2 \log d$. In Figure 1(b), we plot the largest intermediate data size (averaged over 10 runs on random orderings) with respect to $|V| + |E|$. Note that both x-axis and y-axis are in log-scale. Again as conjectured, the plot is linear and always lies below $3(|V| + |E|)$.

For complete binary trees, we again vary the number of nodes from 32 ($2^5$) to $524,288$ ($2^{19}$). The main difference from the path case is that for a complete binary tree, diameter




(a) Runtimes (in minutes)

(b) # of Map-Reduce jobs

Fig. 4. Comparison of Hash-to-All and Hash-to-Min for single linkage clustering on *BizW* and *MovieW*.

is $2\log(|V|)$ and hence the diameter varies only from 10 to 38. Again in Figure 2(a), we see that the rounds complexity still lies below the curve for $2\log d$ supporting our conjecture even for trees. In Figure 2(b), we again see that space complexity grows linearly and is bounded by $3(|V| + |E|)$.

*2) Analysis on Real Data:* We next compared Hash-to-Min, Hash-Greater-to-Min, and Hash-to-All algorithms on real datasets against Pegasus [10]. To the best of our knowledge, Pegasus is the fastest technique on MapReduce for computing connected components. Although all datasets are sparse (have average degree less than 3), each dataset has very different distribution on the size $n$ of the largest connected components and graph diameter $d$. We partition our datasets into two groups – *group I* with $d >= 20$ and relatively small $n$, and *group II* with $d < 20$ and very large $n$.

**Group I: Graphs with large $d$ and small $n$:** This group includes *Biz*, *Movie*, and *SocialSparse* datasets that have large diameters ranging from 20 to 80. On account of large diameters, these graphs requires more MR jobs and hence longer time, even though they are somewhat smaller than the graphs in the other group. These graphs have small connected components that fit in memory.

Each connected component in the *Biz* dataset represents the number of duplicates of a real-world entity. Since there are only two feeds creating this dataset, and each of the two feeds is almost void of duplicates, the size of most connected components is 2. In some extreme cases, there are more duplicates, and the largest connected component we saw had size 93. The *Movie* dataset has more number of sources, and consequently significantly more number of duplicates. Hence the size of some of the connected components for it is significantly larger, with the largest containing 17,213 nodes. Finally, the *SocialSparse* dataset has the largest connected component in this group, with the largest having 2,945,644 nodes. Table IV summarizes the input graph parameters. It also includes the number of map-reduce jobs and the total runtime for all of the four techniques.

Differences in the connected component sizes has a very interesting effect on the run-times of the algorithm as shown in Figures 3(a) and 3(b). On account of the extremely small size of connected components, runtime for all algorithms is fastest for the *Biz* dataset, even though the number of nodes and edges in *Biz* is larger than the *Movie* dataset. Hash-to-All has the best performance for this dataset, almost 3 times faster than Pegasus. This is to be expected as Hash-to-All just takes 4 iterations (in general it takes $\log d$ iterations) to converge. Since connected components are small, the replication of components, and the large intermediate data size does not affect its performance that much. We believe that Hash-to-All is the fastest algorithm whenever the intermediate data size is not a bottleneck. Hash-to-Min takes twice as many iterations ($2\log d$ in general) and hence takes almost twice the time. Finally, Pegasus takes even more time because of a larger number of iterations, and a larger number of map-reduce jobs.

For the *Movie* and *SocialSparse* datasets, connected components are much larger. Hence Hash-to-All does not finish on this dataset due to large intermediate data sizes. However, Hash-to-Min beats Pegasus by a factor of nearly 3 in the *SocialSparse* dataset since it requires a fewer number of iterations. On movies, the difference is the most stark: Hash-to-Min has 15 times faster runtime than Pegasus again due to significant difference in the number of iterations.

**Group II: Graphs with small $d$ and large $n$:** This group includes *Social*, *TwitterSparse*, and *Twitter* dataset that have a small diameter of less than 20, and results are shown in Figures 3(c) and 3(d) and Table V. Unlike Group I, these datasets have very large connected components, such that even a single connected component does not fit into memory of a single mapper. Thus we apply our robust implementation of Hash-to-Min (denoted Hash-to-Min$^*$) described in Sec. IV.

The Hash-to-Min$^*$ algorithm is nearly twice as fast as pegasus, owing to reduction in the number of MR rounds. Only exception is the *Twitter* graph, where reduction in times is only 18%. This is because the *Twitter* graph has some nodes with very high degree, which makes load-balancing a problem for all algorithms.

### B. Single Linkage Clustering

We implemented single linkage clustering on map-reduce using both Hash-to-All and Hash-to-Min hashing strategies. We used these algorithms to cluster the *MovieW* and *BizW* datasets. Figures 4(a) and 4(b) shows the runtime and number of map-reduce iterations for both these algorithms, respectively. Analogous to our results for connected components, for the *MovieW* dataset, we find that Hash-to-Min outperforms Hash-to-All both in terms of total time as well as number of rounds. On the *BizW* dataset, we find that both Hash-to-Min and Hash-to-All take exactly the same number of rounds. Nevertheless, Hash-to-All takes lesser time to complete that Hash-to-Min. This is because some clusters (with small $n$)

| k | MR-k Completion Time | | BSP-k Completion Time | |
|---|---|---|---|---|
| | maximum | median | maximum | median |
| 1 | 10:45 | 10:45 | 7:40 | 7:40 |
| 5 | 17:03 | 11:30 | 11:20 | 8:05 |
| 10 | 19:10 | 12:46 | 23:39 | 15:49 |
| 15 | 28:17 | 26:07 | 64:51 | 43:37 |

TABLE VI
MEDIAN AND MAXIMUM COMPLETION TIMES (IN MIN:SEC) FOR $k$ CONNECTED COMPONENT JOBS DEPLOYED SIMULTANEOUSLY USING MAP-REDUCE (MR-K) AND GIRAPH (BSP-K)

finish much earlier in Hash-to-All; finished clusters reduce the amount of communication required in further iterations.

### C. Comparison to Bulk Synchronous Parallel Algorithms

Bulk synchronous parallel (BSP) paradigm is generally considered more efficient for graph processing than map-reduce as it has less setup and overhead costs for each new iteration. While the algorithmic improvements of reducing number of iterations presented in this paper are important independent of the underlying system used, these improvements are of less significance in BSP due to low overhead of additional iterations.

In this section, we show that BSP does not necessarily dominate Map-Reduce for large-scale graph processing (and thus our algorithmic improvements for Map-Reduce are still relevant and important). We show this by running an interesting experiment in shared grids having congested environments. We took the *Movie* graph and computed connected components using Hash-to-Min (map-reduce, with 50 reducers) and using Hash-Min[11] on Giraph [4] (BSP, with 100 mappers), an open source implementation of Pregel [16] for Hadoop. We deployed $k = 1$, 5, 10, and 15 copies of each algorithm (denoted by MR-k and BSP-k), and tracked the maximum and median completion times of the jobs. The jobs were deployed on a shared Hadoop cluster with 454 map slots and 151 reduce slots, and the cluster experienced normal and equal load from other unrelated tasks.

Table VI summarizes our results. As expected, BSP-1 outperforms MR-1;[12] unlike map-reduce, the BSP paradigm does not have the per-iteration overheads. However, as $k$ increases from 1 to 15, we can see that the maximum and median completion times for jobs increases at a faster rate for BSP-k than for MR-k. This is because the BSP implementation needs to hold all 100 mappers till the job completes. On the other hand, the map-reduce implementation can naturally parallelize the map and reduce rounds of different jobs, thus eliminating the impact of per round overheads. So it is not surprising that while all jobs in MR-15 completed in about 20 minutes, it took an hour for jobs in BSP-15 to complete. Note that the cluster configurations favor BSP implementations since the reducer capacity (which limits the map-reduce implementation of Hash-to-Min) is much smaller ($< 34\%$) than the mapper capacity (which limits the BSP implementation of Hash-Min). We also ran the experiments on clusters with higher ratios of reducers to mappers, and we observe similar results (not included due to space constraints) showing that map-reduce handles congestion better than BSP implementations.

[11]Hash-Min is used as it is easier to implement and not much different than Hash-to-Min in terms of runtime for BSP environment

[12]The time taken for MR-1 is different in Tables IV and VI since they were run on different clusters with different number of reducers (100 and 50 resp.).

## VII. CONCLUSIONS AND FUTURE WORK

In this paper we considered the problem of find connected components in a large graph. We proposed the first map-reduce algorithms that can find the connected components in logarithmic number of iterations – *(i)* Hash-Greater-to-Min, which provably requires at most $3 \log n$ iterations with high probability, and at most $2(|V| + |E|)$ communication per iteration, and *(ii)* Hash-to-Min, which has a worse theoretical complexity, but in practice completes in at most $2 \log d$ iterations and $3(|V| + |E|)$ communication per iteration; $n$ is the size of the largest component and $d$ is the diameter of the graph. We showed how to extend our techniques to the problem of single linkage clustering, and proposed the first algorithm that computes a clustering in provably $O(\log n)$ iterations.

## REFERENCES


[1] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *EDBT*, 2011.

[2] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. In *VLDB*, 2010.

[3] M. Cha, H. Haddadi, F. Benevenutoz, and K. P. Gummadi. Measuring user influence in twitter: The million follower fallacy. In *ICWSM*, 2010.

[4] A. Ching and C. Kunz. Giraph : Large-scale graph processing on hadoop. In *Hadoop Summit*, 2010.

[5] J. Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, 11(4):29–41, July 2009.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51, January 2008.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *MAPREDUCE*, 2010.

[8] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, 1991.

[9] D. B. Johnson and P. Metaxas. Connected components in o(log3/2n) parallel time for the crew pram. *J. Comput. Syst. Sci.*, 54(2):227–242, 1997.

[10] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System- Implementation and Observations. 2009.

[11] D. R. Karger, N. Nisan, and M. Parnas. Fast connected components algorithms for the erew pram. *SIAM J. Comput.*, 28(3):1021–1034, 1999.

[12] H. J. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *SODA*, 2010.

[13] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge*, 1994.

[14] H. R. Lewis and C. H. Papadimitriou. Symmetric space-bounded computation. *Theor. Comput. Sci.*, 19:161–187, 1982.

[15] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[17] S. J. Plimpton and K. D. Devine. MapReduce in MPI for Large-scale Graph Algorithms. *Special issue of Parallel Computing*, 2011.

[18] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce in logarithmic rounds. http://www.cs.duke.edu/ ashwin/pubs/cc-icde13-full.pdf, 2012.

[19] J. Reif. Optimal parallel algorithms for interger sorting and graph connectivity. In *Technical report*, 1985.

[20] T. Seidl, B. Boden, and S. Fries. CC-MR - finding connected components in huge graphs with mapreduce. In *ECML/PKDD (1)*, 2012.

[21] Y. Shiloach and U. Vishkin. An O( $\log n$ ) parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.


## APPENDIX

### A. Proof of Theorem 3.6

We first restate Theorem 3.6 below.

*Theorem A.1 (3.6):* Let $G = (V, E)$ be a path graph (i.e. a tree with only nodes of degree 2 or 1). Then, Hash-to-Min correctly computes the connected component of $G = (V, E)$ in $4 \log n$ map-reduce rounds.

To prove the above theorem, we first consider path graphs when node ids increase from left to right. Then we show the result for path graphs with arbitrary ordering.

*Lemma A.2:* Consider a path with node ids increasing from the left to right. Then after $k$ iterations of the Hash-to-min algorithm,

- For every node $j$ within a distance of $2^k$ from the minimum node $m$, $m$ knows $j$ and $j$ knows $m$.
- For every pair of nodes $i, j$ that are a distance $2^k$ apart, $i$ knows $j$ and $j$ knows $k$.
- Node $j$ is not known to and does not know any node $i$ that is at a distance $> 2^k$.

*Proof:* The proof is by induction.

*Base Case:* After 1 iterations, each node knows about its 1-hop and 2-hop neighbors (on either side).

*Induction Hypothesis:* Suppose the claim holds after $k - 1$ iterations.

*Induction Step:*

In the $k^{th}$ iteration, consider a node $j$ that is at a distance $d$ from the min node $m$, where $2^{k-1} < d \leq 2^k$. From the induction hypothesis, there is some node $i$ that is $2^{k-1}$ away from $j$ that knows $j$. Since, $m$ is known to $i$ (from induction hypothesis), the Hash-to-min algorithm would send $j$ to $m$ and $m$ to $j$ in the current iteration. Therefore, $m$ knows $j$ and $m$ is known to $j$.

Consider a node $j$ that is $> 2^k$ distance from the min node. At the end of the previous iteration, $j$ knew (and was known to) $i$, and $i$ knew and was known to $i'$ – where $i$ and $i'$ are at distance $2^{k-1}$ from $j$ and $i$ respectively. Moreover, $i$ did not know any node $i''$ smaller than $i'$. Therefore, in the current iteration, $i$ sends $i'$ to $j$ and $j$ to $i'$. Therefore, $j$ knows and is known to a node that is $2^k$ distance away.

Finally, we can show that a node does not know (and is not known to) any node that is distance $> 2^k$ as follows. Node $j$ can only get a smaller node $i'$ if $i'$ is a minimum at some node $i$. Since in the previous step no one knows a node at distance $> 2^{k-1}$, $j$ cannot know a node at distance $> 2^k$. ■

Now we extend the proof for arbitrary path graphs. Denote $min_k(u)$ the minimum node after $k$ iterations that $u$ knows that also knows $u$. Also denote $\Delta(u, v)$ the distance between node $u$ and $v$.

*Definition A.3 (Local Minima):* A node $v$ is local minimum if all its neighbors have id larger than $v$'s id.

For a path graph, we define the notion of levels below.

*Definition A.4 (Levels):* Given a path, level 0 consists of all nodes in the path. Level $i$ is then defined recursively as nodes that are local minimum nodes among the nodes at level $i - 1$, if the level $i - 1$ nodes are arranged in the order in which they occur in the path. Denote the set of nodes at level $i$ as $level(i)$.

*Proposition A.5:* The number of levels having more than 1 node is at most $\log n$.

*Proof:* The proof follows from the fact that for each level $i$, no consecutive nodes can be local minimum. Hence $|level(i)| \leq |level(i-1)/2)$. ■

*Lemma A.6:* Consider a path $P$ with three segments $P_1, P_2$ and $P_3$, where $P_1$ and $P_3$ are arbitrary, and $P_2$ has $r + 1$ level $\ell$ nodes $l_1, l_2, ...l_r, m_1$ going from left to right. Assume that labels are such that $l_1 < l_2 < \ldots < l_r$. For a node $l_i$, denote $\mathbf{l}(l_i, k, \ell)$ the closest level $\ell$ node $l_j$ from $l_i$ towards the right such that $min_k(l_j) > l_i$. Denote $T(k, \ell)$ and $M(k, \ell)$ as

$$T(k, \ell) = min_{P, l_i : \mathbf{l}(l_i,k,\ell) \neq l_r} \Delta(l_i, \mathbf{l}(l_i, k, \ell))$$

and

$$M(k, \ell) = min_{P : min_k(l_1) > min_{k-1}(m_1) \text{ or } min_k(m_1) > min_{k-1}(l_1)} \Delta(l_1, m_1)$$

.

Then the following is true:

1) $T$ satisfies the following recurrence realtion

$$T(k, \ell) \geq min(T(k-2, \ell) + min(T(k-1, \ell), M(k, \ell), M(k-1, \ell-1))$$

2) $M$ satisfies the following recurrence relation

$$M(k, \ell) \geq min(T(k-2, \ell), M(k-1, \ell-1))$$

*Proof:* Denote by $[l_t, l_u]$ the level $\ell$ nodes between $l_t$ and $l_u$, and $[l_t, l_u)$ those nodes except for $l_u$.

**Proof of claim 1:** Let $l_s = \mathbf{l}(l_i, k-1, \ell)$ and let $l_t$ be the level $\ell$ node just to the left of $l_s$. We know by definition of $l_s = \mathbf{l}(l_i, k-1, \ell)$ that $min_{k-1}(l_s) > l_i$, but for all $l \in [l_i, l_t], min_{k-1}(l) \leq l_i$. Now there are three cases:

1) $min_{k-1}(l_t) \in P_3$. Then $min_{k-1}(l_t) \leq l_i$ (from above), and any node $l \in [l_t, l_r]$ would have $min_{k-1}(l) \in P_3 \leq min_{k-1}(l_t)$. Hence $\min_{k-1}(l) \leq l_i$ for all $l \in \{l_i, \ldots, l_r\}$. Thus $\mathbf{l}(l_i, k-1) = l_r$.
2) $min_{k-1}(l_t) \notin P_3$ and $\Delta(l_i, l_t) \geq T(k-1, \ell)$. Denote $l_u = \mathbf{l}(l_t, k-1, \ell)$. Consider any $l \in [l_t, l_u)$. Let $a = min_{k-1}(l)$. Then from the definition of $l_u = \mathbf{l}(l_t, k-1)$ and since $l \in [l_t, l_u)$, we know that, $a = min_{k-1}(l) \leq l_t$. Now there are two sub-cases:

a) For all $l \in [l_t, l_u)$, $a = min_{k-1}(l) \notin P_3 \cup \{m_1\}$. In this case, we show that $\mathbf{l}(l_i, k, \ell)$ is a level $\ell$ node in $[l_u, l_r]$. For this we will show that for all $l \in [l_t, l_u)$ and $a = min_{k-1}(l)$, $min_{k-1}(a) \leq l_i$. If $a \notin P_3 \cup \{m_1\}$, then either (i) $a \in P_1$, in which case $a \leq l_1$ (otherwise $min_{k-1}(l)$ would be $l_1$ and not $a$), and hence obviously $min_{k-1}(a) \leq l_i$, or (ii) $a \in P_2$ but $a \neq m_1$, and then since $a \leq l_t$ (from above), $a \in [l_i, l_t]$. Hence $min_{k-1}(a) \leq l_i$.
In other words, after $k-1$ iterations, the minimum for any node $l \in [l_t, l_u)$ is $a$, which in turn has a minimum $b \leq l_i$. Thus in one Hash-to-Min step, $b$ would become $min_k(l)$. Hence after $k$ iterations and for any local minimum $l$ between $l_i$ and $l_u$, we have $min_k(l) \leq l_i$. This shows that $\mathbf{l}(l_i, k, \ell) \in [l_u, l_r]$. Then either $l_u$ is $l_r$ or the following holds.

$$\Delta(l_i, \mathbf{l}(l_i, k, \ell)) \geq \Delta(l_i, l_t) + \Delta(l_t, l_u) \geq T(k-2, \ell) + T(k-1, \ell-1)$$

b) There exists $l \in [l_t, l_u)$, such that $a = min_{k-1}(l) \in P_3$. Let $l_w = \mathbf{l}(l_i, k, \ell)$. Then $l_w$ is the first node in $[l_i, l_r]$ for which $min_k(l_w) > l_i$. If $\Delta(l_v, l_w) \leq M(k, \ell)$ then by definition after $k$ iterations $min_k(l_w) \leq min_{k-1}(l_t) \leq l_i$. Thus $\Delta(l_w, l_i) \geq M(k, \ell)$. Thus

$$\Delta(l_i, \mathbf{l}(l_i, k, \ell)) \geq \Delta(l_i, l_t) + \Delta(l_t, l_w) \geq T(k-2, \ell) + M(k, \ell)$$

3) $min_{k-1}(l_t) \in P_1 || P_2$ and $\Delta(l_i, l_t) \leq T(k-2, \ell)$. In this case, we argue that $\Delta(l_t, l_s) > M(k-1, \ell-1)$. Assume the contrary: $\Delta(l_t, l_s) \leq M(k-2, \ell-1)$. Since, $\Delta(l_i, l_t) \leq T(k-2, \ell)$, we know that $min_{k-2}(l_t) \leq l_i$. Since $l_t$ and $l_s$ are consecutive level $\ell$ nodes, all the level $\ell-1$ nodes between them are ordered. Hence by definition of $M(k-1, \ell-1)$, and the fact that $\Delta(l_t, l_s) \leq M(k-1, \ell-1)$, $min_{k-1}(l_s) \leq min_{k-2}(l_t) \leq l_i$ . This contradicts the assumption that $l_s = \mathbf{l}(l_i, k-1, \ell)$. Thus $\Delta(l_t, l_s) > M(k-1, \ell-1)$.

$$\Delta(l_i, \mathbf{l}(l_i, k)) \geq \Delta(l_i, l_t) + \Delta(l_t, l_s) \geq M(k-1, \ell-1)$$

Combining the above cases we complete the proof of claim 1.

**Proof of claim 2:** If $\Delta(l_1, l_r) \leq T(k-2, \ell)$, then $min_{k-2}(l_r) \leq l_1$. Also if $\Delta(l_r, m_1) \leq M(k-1, \ell-1)$, then $min_{k-1}(m_1) \leq min_{k-2}(l_r) \leq l_1$. If both $\Delta(l1, l_r) \leq T(k-2, \ell)$ and $\Delta(l_r, m_1) \leq M(k-1, \ell-1)$, then $min_{k-1}(m_1) \leq l_1$. Hence by claim 1, $min_k(m) \leq min_{k-1}(l_1)$. This completes the proof.

■

*Lemma A.7:* Let $T(k, \ell)$ be the quantity as defined in Lemma A.6. Then $T(k, \ell) \geq 2^{k/2-\ell}$.

*Proof:* We prove the lemma using induction.

*Base Cases: (i)* $\ell = 0$ *and* $k \geq 1$. *Then by Lemma A.2,* $T(k, 0) \geq 2^k \geq 2^{k/2-0}$. *(ii)* $k = 1$ *and* $\ell \geq 1$. For any level $\ell \geq 1$, $T(1, \ell) \geq 1 \geq 2^{1/2-\ell}$.

*Induction Hypothesis (IH)* For all $k_0 \leq k-1$ and $\ell_0 \leq \ell - 1$, $T(k_0, \ell_0) \geq 2^{k_0/2-\ell_0}$

*Induction Step:* By Lemma A.6, we know that:

$$\begin{aligned} T(k, \ell) &\geq min\left(T(k-1, \ell) + T(k-2, \ell), T(k-2, \ell-1)\right) \\ &\geq min\left(2^{(k-1)/2-\ell} + 2^{(k-2)-\ell}, 2^{(k-2)/2-\ell+1}\right) \quad \text{(using IH)} \\ &\geq min\left(2^{k/2-\ell}(1/2 + 1/\sqrt{2}), 2^{k/2-\ell}\right) \geq 2^{k/2-\ell} \end{aligned}$$

■

Finally, we can complete the proof of Theorem 3.6. Since $T(k, l)$ is less than the length of the path, we know that $T(k, l) < n$. Now from Prop. A.5, the number of levels having more than 1 node is at most $\log n$. Hence $\ell \leq \log n$. Finally, from Lemma A.7, we know that $T(k, \ell) \geq 2^{k/2-\ell}$. Thus $2^{k/2-\log n} \leq 2^{k/2-\ell} \leq n$. Thus $k \leq 4 \log n$. This completes the proof of Theorem 3.6.

### *B. Proof of Theorem 3.9*

We first restate Theorem 3.9 below.

*Theorem A.8 (3.9):* Algorithm Hash-Greater-to-Min correctly computes the connected components of $G = (V, E)$ in expected $3 \log n$ map-reduce rounds (expectation is over the random choices of the node ordering) with $2(|V| + |E|)$ communication per round in the worst case.

*Proof:* After $3k$ rounds, denote $M_k = \{min(C_v) : v \in V\}$ the set of nodes that appear as minimum on some node. For a minimum node $m \in M_k$, denote $GT_k(m)$ the set of all nodes $v$ for which $m = min(C_v)$. Then by Lemma 3.8, we know that $GT_k(m) = C_{\geq}(m)$ after $3k$ rounds. Obviously $\cup_{m \in M_k} GT_k(m) = V$ and for any $m, m' \in M_k$, $GT_k(m) \cap GT_k(m') = \emptyset$.

Consider the graph $G_{M_k}$ with nodes as $M_k$ and an edge between $m \in M_k$ to $m' \in M_k$ if there exists $v \in GT_k(m)$ and $v' \in GT_k(m')$ such that $v, v'$ are neighbors in the input graph $G$. If a node $m$ has no outgoing edges in $G_{M_k}$, then $GT_k(m)$ forms a connected component in $G$ disconnected from other components, this is because, then for all $v' \notin GT_k(m)$, there exists no edge to $v \in GT_k(m)$.

We can safely ignore such sets $GT_k(m)$. Let $MC_k$ be the set of nodes in $G_{M_k}$ that have at least one outgoing edge. Also if $m \in MC_k$ has an edge to $m' < m$ in $G_{M_k}$, then $m$ will no longer be the minimum of nodes $v \in GT_k(m)$ after 3 additional rounds. This is because there exist nodes $v \in GT_k(m)$ and $v' \in GT_k(m')$, such that $v$ and $v'$ are neighbors in $G$. Hence in the first round of Hash-Min, $v'$ will send $m'$ to $v$. In the second round of Hash-Min, $v$ will send $m'$ to $m$. Hence finally $m$ will get $m'$, and in the round of Hash-Greater-to-Min, $m$ will send $GT_k(m)$ to $m'$.

If $|MC_k| = l$, W.L.O.G, we can assume that they are labeled 1, 2, ..., $l$ (since only relative ordering between them matters anyway). For any set, $GT_k(m)$, the probability that it its min $m' \in (l/2, l]$ after 3 more rounds is $1/4$. This is because that happens only when $m \in [(l/2, l)$ and all its neighbors $m' \in G_{M_k}$ are also in $(l/2, l]$. Since there exist at least one neighbor $m'$, the probability of $m' \in (l/2, l]$ is at most $1/2$. Hence the probability of any node $v$ having a min $m' \in (l/2, l]$ after 3 more rounds is 1/4.

Now since no set, $GT_k(m)$, ever get splits in subsequent rounds, the expected number of cores is 3l/4 after 3 more rounds. Hence in three rounds of Hash-Greater-to-Min, the expected number of cores reduces from $l$ to $3l/4$, and therefore it will terminate in expected $3 \log n$ time.

The communication complexity is $2(|V|+|E|)$ per round in the worst-case since the total size of clusters is $\sum_v C_{\geq v} = 2(|V|)$. ■

### C. Proof of Theorem 5.6

We first restate Theorem 5.6 below.

*Theorem A.9 (5.6):* The distributed Algorithm 3 simulates the centralized Algorithm 2, i.e., it outputs the same clustering as Algorithm 2.

*Proof:* Let $C_{central}$ be the clustering output by Algorithm 2. Let $C_{distributed}$ be the clustering output by Algorithm 3. We show the result in two parts.

First, for any cluster $C_{distributed} \in C_{distributed}$, there exists a cluster $C_{central} \in C_{central}$ such that $C_{distributed} \subseteq C_{central}$. Since Algorithm 3 uses the splitting algorithm 6, it outputs only cores having cluster splits $C_l, C_r$ for which $Stop_{local}(C_l)$ and $Stop_{local}(C_r)$ equal false. Thus we can invoke Lemma 5.5 on $C_{distributed}$ to prove that $C_{distributed}$ is valid, and the existence of $C_{central}$ such that $C_{distributed} \subseteq C_{central}$.

Having shown that $C_{distributed} \subseteq C_{central}$, we now show that, in fact, $C_{distributed} = C_{central}$. Assume the contrary, i.e. $C_{distributed} \subset C_{central}$. Since $C_{distributed}$ is valid, even the centralized algorithm constructed $C_{distributed}$ some time during its execution, and then merged it with some other cluster, say $C'_{central}$.

Since $C_{distributed}$ is in the output of Algorithm 3, then three cases are pospsible: (i) $C_{distributed}$ forms a connected component by itself, disconnected from the rest of the graph, or (ii) $Stop_{local}(C_{distributed})$ is true, and the algorithm stops because of the stopping condition, or (iii) $Stop_{local}(C_{distributed})$ is false, but it merges with some cluster $C'_{distributed}$ for which $Stop_{local}(C'_{distributed})$ is true. In the first two cases, even the centralized algorithm can not merge $C_{distributed}$ with any other cluster, contradicting that $C_{distributed} \subset C_{central}$.

For case (iii), we show below that in fact $C'_{distributed} \subseteq C'_{central}$. Since the central algorithm merges $C'_{central}$ with $C_{central}$, $Stop_{local}(C'_{central})$ has to be false. Since $Stop_{local}$ is monotonic, and $C'_{distributed} \subseteq C'_{central}$, $Stop_{local}(C'_{distributed})$ has to be false as well, contradicting the assumption made in case (iii). Thus we proved all three cases are impossible, contradicting our assumption of $C_{distributed} \subset C_{central}$. Hence $C_{distributed} = C_{central}$.

Now we show that $C'_{distributed} \subseteq C'_{central}$. Both $C'_{distributed}$ and $C'_{central}$ have to be closest to $C_{distributed}$, i.e. in $nbrs(C_{distributed})$, in order to get merged with it in either the central or distributed algorithms. Denote $v$ to be the node, such that the singleton cluster $\{v\}$ is in $nbrs(C_{distributed})$. Hence, by the property of single linkage clustering, both $C'_{distributed}$ and $C'_{central}$ must contain the node $v$. Since $C'_{distributed}$ is valid, there must be a cluster in the central algorithm's output containing it. Finally, since clusters in the output have to be disjoint, the cluster containing $C'_{distributed}$ has to be $C'_{central}$, and thus $C'_{distributed} \subseteq C'_{central}$. ■