Connected Components in MapReduce and Beyond

Raimondas Kiveris Silvio Lattanzi Vahab Mirrokni Vibhor Rastogi Sergei Vassilvitskii

Google

{rkiveris, silviol, mirrokni, rvibhor, sergeiv} @google.com

Abstract

Computing connected components of a graph lies at the core of many data mining algorithms, and is a fundamental subroutine in graph clustering. This problem is well studied, yet many of the algorithms with good theoretical guarantees perform poorly in practice, especially when faced with graphs with hundreds of billions of edges. In this paper, we design improved algorithms based on traditional MapReduce architecture for large scale data analysis. We also explore the effect of augmenting MapReduce with a distributed hash table (DHT) service. We show that these algorithms have provable theoretical guarantees, and easily outperform previously studied algorithms, sometimes by more than an order of magnitude. In particular, our iterative MapReduce algorithms run 3 to 15 times faster than the best previously studied algorithms, and the MapReduce implementation using a DHT is 10 to 30 times faster than the best previously studied algorithms. These are the fastest algorithms that easily scale to graphs with hundreds of billions of edges.

Categories and Subject Descriptors G.2.2 [*General*]: Graph algorithms

Keywords Connected Components, MapReduce Algorithms

1. Introduction

Large-scale graph mining is a basic tool for modeling social, communication, and information networks, and is an increasingly important problem in big data analysis. Performing this computation in a fault-tolerant and commonly-used distributed programming framework such as MapReduce or Hadoop is a necessary step toward developing a general-purpose easy-to-use graph mining framework. Computing connected components is the fundamental first step which lies at the core of many more sophisticated graph analysis techniques.

Most of previously studied algorithms with good theoretical guarantees may perform poorly in practice, because they either (i) require too many rounds of computation (for example, scaling with the diameter of the graph) [7, 15], (ii) increase the size of the graph during the computation [26], resulting in high communication costs, or (iii) yield unbalanced workloads with a handful of machines slowing down the rest of the computation.

In this work, we study this problem in depth, and provide algorithms which have strong theoretical guarantees, bounded communication cost, and load balanced computation. In addition, we augment the traditional MapReduce architecture with a distributed hash table (DHT) service and show how to replace some of the MapReduce rounds by simple lookups in a DHT, thereby further reducing the computation time.

Our algorithms easily scale to graphs with billions of nodes and hundreds of billions of edges on a shared cluster with commodity hardware. We perform extensive empirical evaluation of our algorithms, and compare them to the best known state of the art approaches. With all algorithms implemented on the same hardware, our algorithms outperform the fastest time algorithm reported in previous work [26] by an order of magnitude. Specifically, our iterative MapReduce algorithms run *3 to 15 times* faster than the previously studied algorithms, and the DHT-augmented implementations lead to an *additional* factor two speedup. Finally, we compare our approach with algorithms on other models of distributed computation, and show that on large graphs our algorithms outperform Pregel based connected components algorithms as well.

Algorithmic implications Connected components is a fundamental tool for hierarchical clustering, in particular several parallel hierarchical clustering algorithms use it as a subroutine [8, 9, 12, 22, 24, 25]. The core idea behind these algorithms is to first compute similarity scores between nodes and then compute connected components subject to different similarity thresholds. Therefore, the running time of all those approaches strictly depends on the scalability of the underlying parallel connected components algorithm.

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA.

ACM 978-1-4503-3252-1.

http://dx.doi.org/10.1145/2670979.2670997

Recent data mining work [13, 14] has focused on the study of the distribution of connected components in real world graphs. Our algorithms would let such analysis be performed on ever larger graphs that could not be analyzed before.

1.1 Related Work

The problem of finding the connected components of an undirected graph has been studied in many distributed settings, both theoretical and practical. At a high level, there are three models under which this problem has been studied: the classical PRAM model, the Block Synchronous Parallel (BSP) model introduced by Valiant [31] and the MapReduce paradigm.

PRAM . The Parallel RAM (PRAM) model of computation is the most popular model for analyzing the performance of distributed algorithms. The state of the art is the PRAM algorithm of Shiloach and Vishkin [29] which requires the PRAM to handle concurrent reads and writes (CRCW PRAM). The algorithm maintains a forest of directed trees and repeatedly applies pointer chasing operations (pointing to your parent's parent), or hooking (merging two trees). The algorithms we propose are conceptually similar, but do not require the shared memory that is integral to PRAM approaches and is hard to realize in practice. There are theoretical simulation results known [11, 17] that overcome the shared memory but at the cost of extra complexity, and an additional $O(\log n)$ rounds in case of CRCW PRAMs.

BSP: The BSP paradigm [31] is used by recent distributed graph processing systems like Pregel [21] and Giraph [6]. While, BSP is generally considered more efficient for graph processing, MapReduce has certain advantages owing to its fault tolerance. Previous work [26] showed that in congested clusters, MapReduce can have a better latency than BSP. Recent work [20] has proposed using asynchronous versions of BSP to further improve algorithm running times, however there is not yet a formal model of computation for this setting.

MapReduce: A simple way to find the number of connected components in parallel is to start growing a Breadth First Search (BFS) tree from each node in parallel. The number of iterations required for such an approach [7, 15, 23], is $O(d)$ where d is the diameter of the largest connected component. These techniques do not scale well to medium and large diameter graphs, where each iteration requires non-trivial time to perform.

Afrati et al [3] propose MapReduce 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 prohibitive communication per iteration as the transitive closure relation itself is quadratic in the size of largest component.

The work most related to ours is that of [26], which presents an $O(\log n)$ rounds MapReduce algorithm, where n is the number of nodes. The paper also proposes a simpler algorithm, called Hash-to-Min which is even faster in practice. (A similar approach was introduced by [28], but that work lacks any analytical guarantees, and is outperformed by the Hash-to-Min algorithm.) A major downside of Hash-to-Min is that it increases the intermediate data size in each mapreduce round (as much as doubling the graph size in the worst-case), and does not have good load balancing properties. Our algorithms address both these issues and outperform the Hash-to-Min approach.

Single Machine In addition to the parallel methods, recent work has explored efficient graph algorithms on a single machine [19]. Through careful scheduling and indexing the authors show impressive performance gains on multi billion edge graphs, however their approaches do not scale to graphs with hundreds of billions of edges. Even efficiently storing such graphs on a single machine is infeasible.

1.2 Our Contributions

We present a new algorithm, *Two-Phase*, which computes connected components by iteratively transforming the input graph over multiple mapreduce rounds. We prove that it converges in $\Theta(\log^2 n)$ rounds of MapReduce, where n is the number of nodes in the graph and show that our analysis is tight. We prove that unlike past approaches, this algorithm transforms the graph in a way that the number of edges never increases. Moreover, we can formally prove that the number of edges quickly decreases for certain families of random graphs.

The Two-Phase algorithm is local, with every node in the graph performing some rewiring decisions based solely on the structure of its neighborhood. This locality makes for an easy implementation in a distributed setting, but makes the running time dependent on the degree of each node. Furthermore, as the iterations proceed the degrees of some nodes become higher, until at convergence when they are connected to all nodes in their respective connected components. The load balancing step automatically breaks up high degree nodes into several nodes of lower degree while preserving the connectivity. As we will show this leads to a slight increase in the overall number of rounds, but makes each round of MapReduce much faster, resulting in an overall speedup.

We then show how the Two-Phase algorithm can take advantage of a distributed hash table of size $O(n/\log n)$ to further reduce its running time to $O(\log n \log\log n)$ rounds. This reduction in the number of rounds has a significant impact in practice, reducing the overall running time. We compare the theoretical guarantees of our algorithms to previous work in Table 1. The table shows that our Two-Phase algorithm takes $O(\log^2 n)$ number of MapReduce rounds without DHT, and $O(\log n \log\log n)$ with DHT. This is slightly worse than the previous best of $O(\log n)$ in [26]. However, the overall run-

Name	# of steps	Communication
Pegasus [15]	$O(d)$	$O(m+n)$
Zones [7]	$O(d)$	$O(m+n)$
L Datalog [3]	$O(d)$	$O(n^2+m)$
NL Datalog [3]	$O(\log d)$	$O(n^2+m)$
PRAM [10, 16, 17, 27, 29]	$O(\log n)$	shared memory[1]
Hash-To-Min [26]	Unknown	$2(n+m)$
Hash-Greater-To-Min [26]	$O(\log(n))$	$2(n+m)$
Two-Phase	$O(\log^2 n)$	$2m$
Two-Phase + DHT	$O(\log n \log\log n)$	$2m$

Table 1. Complexity comparison with previous work . We let d denote the graph diameter.

ning time is significantly smaller because of the sparsity of transformed graphs and load balancing.

We demonstrate this by evaluating the performance of our algorithms on ten real world networks and a series of synthetically generated RMAT graphs. On larger real world networks the running time goes down from more than a day to a few hours, on smaller networks, running time decreases from several hours to tens of minutes. We also compare our algorithms against those on other distributed computing platforms, such as Pregel, and show that our approaches are faster when faced with very large graphs. Finally, we use the synthetically generated RMAT graphs to demonstrate the scalability of our algorithms and the utilization of the available resources as a function of input size.

2. Preliminaries

Let $G = (V,E)$ be an undirected graph, on $n = |V|$ nodes, and $m = |E|$ edges. For a node v, we denote by $\Gamma(v) = \{w|(v,w) \in E\}$ the neighbors of v. We will use $\Gamma^+(v) = \Gamma(v) \cup \{v\}$ to denote the neighborhood of v and itself. Furthermore, with every node v, we associate a real number label ℓ_v.

The algorithms in this paper make use of the small-star and large-star operations. Both operations are given a node u and a subset of its neighborhood, $N \subseteq \Gamma(u)$. Let $m(u) = \arg\min_{v\in\Gamma^+(u)} \ell_u$ be the neighbor node with the minimum label. The operations replace for every $v \in N$ the edge (v,u) with an edge $(v,m(u))$. Specifically, for small-star $N = \{v \in \Gamma(u) : \ell_v \leq \ell_u\}$, and for large-star $N = \{v \in \Gamma(u) : \ell_v > \ell_u\}$. We illustrate the small-star and large-star operations pictorially in Figures 1(a) and 1(b).

Note that both the small-star and large-star operations can easily be implemented in a distributed manner. We give the MapReduce version of the two operations in Algorithms 1 and 2.

Even though it seems from Figure 1(a) and Figure 1(b), that the two operations can disconnect the graph, we show below that performing the operations at all nodes in parallel preserves the connectivity of the graph.

Lemma 1. *Performing the large-star operation on all nodes in parallel preserves the connectivity of the graph.*

1: **Map** $\langle u;v\rangle$:
2: **if** $\ell_v \leq \ell_u$ **then**
3: Emit $\langle u;v\rangle$.
4: **else**
5: Emit$\langle v;u\rangle$.
6: **end if**
7: **Reduce** $\langle u;N \subseteq \Gamma(u)\rangle$:
8: Let $m = \arg\min_{v\in N\cup\{u\}} \ell_v$.
9: Emit $\langle v;m\rangle$ for all $v \in N$.

Algorithm 1: The small-star operation

1: **Map** $\langle u;v\rangle$:
2: Emit $\langle u;v\rangle$ and $\langle v;u\rangle$.
3: **Reduce** $\langle u;\Gamma(u)\rangle$:
4: Let $m = \arg\min_{v\in\Gamma^+(u)} \ell_v$.
5: Emit $\langle v;m\rangle$ for all v where $\ell_v > \ell_u$.

Algorithm 2: The large-star operation

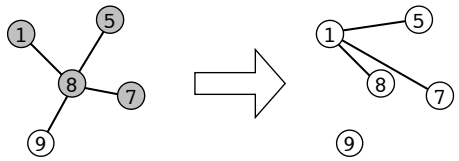
Proof. Given a graph $G = (V,U)$, let $C \subseteq V$ be an arbitrary connected component in G. Without loss of generality assume every label ℓ_v is unique, and orient every edge from the node of larger label to that of a smaller label, and let H be the resulting Directed Acyclic Graph.

To show the first part of the Lemma, consider any path (u,v,w) of length two. If the middle node has the smallest label, that is: $u \to v \leftarrow w$, then large-star on v results in all three nodes being connected to the same node. If the middle node has the largest label, that is: $u \leftarrow v \to w$, then large-star on u will connect both u and v to the same minimum node, and large-star on w will connect both v and w to the same node, preserving connectivity. Finally, if the nodes are in increasing order $u \to v \to w$, then large-star on v will preserve connectivity between u and v by connecting them to the same node, and large-star on w will preserve connectivity between v and w. □

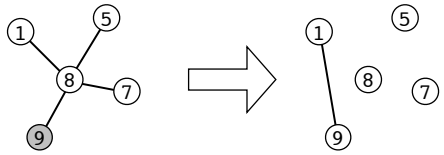
Lemma 2. *After one round of large-star the number of edges in the resulting graph is no larger than in the original graph.*

Proof. We show this property by giving a surjective function mapping directed edges (m,v) such that $\ell_m < \ell_v$ in the new graph to directed edges (u,v) such that $\ell_u < \ell_v$ in the original graph.

To define the mapping, note that an edge (m,v) was either already present in the previous step or must have resulted from at least some large-star operation. Let u be the node at which the operation happens. So we have that $\ell_u < \ell_v$ and $m = min(u)$. In this case we map (u,v) to (m,v). Now suppose (m,v) and (m',v) are two edges such that $\ell_m < \ell_v, \ell'_m < \ell_v$, and $m \neq m'$, and both edges both are mapped from (u,v). Then we know $m = m' = min(u)$, which leads to a contradiction. Thus the mapping is surjective, thus the claim follows. □



(a) The small-star operation at node 8.



(b) The large-star operation at node 8.

Figure 1. An illustration of small-star and large-star operations at node $v = 8$. In both cases, the minimum neighbor of 8 is 1. In Figure 1(a), small-star connects 1 to all nodes labels no larger than 8. In particular, 1 is connected to 8. In Figure 1(b), large-star connects 1 to all nodes with label strictly larger than 8. In particular, 1 is not connected to 8.

```
1: Input: Edges (u,v) as a set of key-value pairs ⟨u;v⟩.
2: Input: A unique label ℓ_v for every node v ∈ V.
3: repeat
4:     repeat
5:         large-star
6:     until Convergence
7:     small-star
8: until Convergence
```

Algorithm 3: The Two-Phase Algorithm

Lemma 3. *The small-star operation preserves connectivity and does not increase the number of edges.*

Proof. Consider any edge (u,v) and suppose without loss of generality that $\ell_u < \ell_v$. Then, after executing small-star on v, both u and v will be connected to the same node. In this case every new edge can also be mapped to an existing edge as in Lemma 2, and the claim about the number of edges follows. □

We note that both the small-star and large-star operations are similar in nature to the Hash-to-Min operation of [26]. However, unlike small-star and large-star, Hash-to-Min can introduce extra edges, doubling the total number of edges in the worst-case In practice, the smaller number of edges output by our algorithms results in a significant improvement in running times as shown by our experimental evaluation.

3. Algorithms

We propose two algorithms for connected components, namely Two-Phase and Alternating. Both algorithms combine invocations of large-star and small-star until a fixed point is reached at which point the overall graph is reduced to a union of disconnected stars, one for each connected component.

We present the Two-Phase and Alternating algorithms in Algorithm 3 and Algorithm 4, respectively. The only difference between the two algorithms is the ordering of the invocations of large-star and small-star. In Alternating, the large-star and small-star operations are repeated one after the other until convergence, while the Two-Phase algorithms

```
1: Input: Edges (u,v) as a set of key-value pairs ⟨u;v⟩.
2: Input: A unique label ℓ_v for every node v ∈ V.
3: repeat
4:     large-star
5:     small-star
6: until Convergence
```

Algorithm 4: The Alternating Algorithm

proceeds in phases. In each phase, large-star is repeated until convergence followed by one small-star operation. In the next two sections, we analyze the running time and communication complexity of the algorithms, and present some further optimizations.

3.1 The Two-Phase algorithm

3.1.1 Number of Rounds

Intuitively this algorithm efficiently simulates the classical Union-Find algorithm in parallel. In the inner-loop each node, *finds* its local-minimum and adds an edge to it (this is equivalent to path compression). In the outer loop, distinct local minimums are connected and then joined in a *union* step.

Theorem 1. *The Two-Phase Algorithm converges after $O(\log^2 n)$ MapReduce rounds.*

To prove the theorem we will show that the inner loop is executed at most $O(\log n)$ times per iteration of the outer loop (Lemma 4), and the outer loop is executed at most $O(\log n)$ times (Lemma 5).

Lemma 4. *Starting with any graph $G = (V,E)$, $O(\log n)$ invocations of large-star leads to a fixed point.*

Proof. Starting with the graph G, direct all of the edges to go from a node with a lower label to that with a higher label, and let H be the resulting Directed Acyclic Graph. We will show that in every iteration of large-star the depth of the graph is reduced by a factor of $3/2$ (until the depth is 2). Let $L_0, L_1, \ldots, L_k$, with $L_i \subseteq V$ be the level sets of H: L_0 are those nodes with no incoming edges, and nodes in L_{i+1} are direct descendants of nodes in L_i.

Note that after every iteration of large-star each node in L_i (for $i \geq 2$) is connected to a node in L_{i-2}. Thus there is a direct path on the newly added edges of length $L_{\lceil i/2 \rceil}$ between a node in L_0 and a node in L_i. Here the level of each node is reduced at least by a factor 3/2. Since the initial graph has depth at most n, the Lemma follows. □

Lemma 5. *The outer loop of the Two-Phase algorithm is executed at most $O(\log n)$ times.*

Proof. Starting with the graph G, direct all of the edges to go from a node with a lower label to that with a higher label, and let H be the resulting Directed Acyclic Graph. We prove the Lemma by induction on the size of L_0 (defined as in the previous lemma) in the graph present after large-star is run to convergence. Note that initially the size of L_0 is bounded by n. Furthermore, iterations of large-star leave the size of L_0 unchanged. Now consider an iteration of small-star, we will show that the size of L_0 in each connected component is reduced by a constant factor every two rounds and the Lemma follows.

For each connected component, we consider the graph $R = (L,A)$ where $L \subseteq L_0$ is the set of level 0 nodes corresponding to this component, and there is an edge $(u,v) \in A$ if there is a directed path $v \to w \leftarrow u$ in H. Observe that after a single iteration of small-star two nodes can remain in L only if they were not connected in R. Therefore after two iterations two nodes can remain in L only if they are connected by a path of length at least 3 in R. Let L' be the set of nodes remaining after two iterations. Then for any two nodes $u,v \in L'$ we know that their neighborhoods in R do not overlap. Therefore, there is a surjective map from nodes in L' to nodes in $L \setminus L'$, which further implies that $|L'| \leq |L|/2$. □

3.1.2 Lower Bounds

In this section we show that the analysis presented above is tight. We will present a graph G such that the Two-Phase algorithm on G takes $\Omega(\log^2 n)$ rounds.

A building block of our construction is a path graph, P_i on 4^i nodes (and hence with $4^i - 1$ edges) such that for the first $i-1$ iterations of the outer loop both small-star and large-star take a single round to converge. However, the i-th iteration requires $\Omega(i)$ rounds to converge.

We define G as the union of $\log(n/4)$ graphs, $P_1, P_2, \ldots, P_{\log(n/4)}$. Then, for each step i we have to wait i steps for the inner loop to converge. For a moment we assume to have a way of constructing the path P_i and we show the main results of this subsection:

Lemma 6. *Let $G = P_1 \cup P_2 \cup \ldots \cup P_k$ be a union of $k = \log(n/4)$ path graphs as described above. Then running the Two-Phase Algorithm on G requires $\Omega(\log^2 n)$ steps to convergence.*

Proof. The total number of nodes in G is $\sum_{i=1}^{\log(n/4)} 4^i = \sum_{i=1}^{\log(n/2)-1} 2^{2i} \leq 2^{\log n} = n$. In the i-th phase of the algorithm the graph P_i will take $\Omega(i)$ rounds to converge, while the other path graphs will have converged in at most a single round. Therefore, the total number of iterations of the inner loop will be $\sum_{i=1}^{k} \Omega(i) = \Omega(k^2) = \Omega(\log^2 n)$. □

To create the i-th graph, P_i, start with a line graph L_i on 2^i nodes with labels arranged in increasing order. For example, L_2 is a graph on 4 nodes, $u_1 \to u_2 \to u_3 \to u_4$ with each u_j having a label j. Note that large-star takes $\Omega(j)$ rounds to converge on a graph L_j since the DAG corresponding to L_j has 2^j levels and large-star decreases the number of levels by a constant factor at every iteration.

To create P_i we start with the graph L_i and then will iteratively transform it so that it is the result of running $i-1$ phases of the Two-Phase algorithm.

Lemma 7. *Given a path graph $P = (p_1, p_2, \ldots, p_n)$ on n nodes, we can create a path graph P' on $4n-1$ nodes so that running large-star until convergence on P' followed by one iteration of small-star results in P.*

Proof. Starting with a graph P, let m be the maximum label in P. Let I_x be the interval $(m, m+1]$ and I_y be the interval $(m+1, m+2)$. Then let P' be the graph obtained from P by replacing each edge (p_i, p_{i+1}) by a length four path $p_i, x_{2i-1}, y_i, x_{2i}, p_{i+1}$, where the nodes in x_j are assigned arbitrary labels from I_x and nodes y_j are assigned arbitrary labels from I_y.

Consider executing large-star on the resulting graph P'. For any three nodes p_i, x_j, y_k we have $\ell_{y_i} > \ell_{x_k} > \ell_{p_j}$. Therefore, running large-star on a node p_i or y_k produces no new edges since p_i is a local minimum, and y_k is a local maximum. Running large-star on a node x_j results in an edge between its neighbors, $p_{\lfloor (j+2)/2 \rfloor}$ and $y_{\lfloor (j+1)/2 \rfloor}$, as well as an edge from x_j to its p-neighbor, $p_{\lfloor (j+2)/2 \rfloor}$. Call this graph P''. At this point, large-star has converged. Indeed, running large-star on a node p_i results in it retaining its neighboring edges, since it's a local minimum, and running it on y_k does not produce any change since it is a local maximum.

Consider what happens when we run small-star on P''. Since any node p_i is a local minimum, small-star on p_i has no effect. On the other hand running small-star on a node y_i leads to an edge between p_i and p_{i+1} since both are connected to y_i and both are smaller than y_i. □

3.2 The Alternating Algorithm

Motivated by the Two-Phase algorithm and the lower bound example, we introduce the Alternating algorithm. The algorithm is designed to circumvent the lower bound, and indeed it is easy to see that the alternating algorithm takes $O(\log n)$ rounds on the bad example. Unfortunately, we cannot prove that this is always the case. We show below that the algorithm always converges, but leave the analysis of the number of rounds as a challenging open question.

Lemma 8. *The Alternating algorithm converges in $O(n)$ rounds.*

Proof. Consider a potential function which assigns to every node the smallest label of its neighbors,

$$\phi = \sum_v \min_{u \in \Gamma^+(v)} \ell_u.$$

Every iteration of small-star and large-star can only decrease the potential function. Furthermore, the only time neither small-star nor large-star change the topology of the graph is when the overall graph is a union of disjoint stars, one for each connected component.

To prove the bound on the number of rounds, fix a connected component, and observe that after execution of both small-star and large-star, the node with the minimum label gains at least one additional neighbor. Therefore, after $2(n-1)$ rounds, the algorithm must converge. □

3.3 Connected components with a DHT

We discuss how to improve the running time of our Two-Phase algorithm using a distributed hashtable (DHT). This reduces the total number of rounds of MapReduce, and as a result significantly improves the running time of our implementation. Roughly speaking, the idea is to use a read-only DHT (e.g. see [5]) to reduce the inner loop to one round of large-star. The DHT represents the graph at the end of the small-star operation, with every node pointing to its smallest neighbor. We call the resulting algorithm Two-Phase DHT and show that it can compute connected components in $O(\log n \log\log n)$ rounds of MapReduce by using a DHT of size $O\left(\frac{n}{\log n}\right)$.

We will use the DHT to replace the inner loop of the Two-Phase algorithm. The DHT stores a mapping from a node to its smallest neighbor, with local minima pointing to themselves. Then, every node repeatedly looks up its smallest neighbor in the DHT, until it reaches convergence. At this point it emits a key value pair, representing an edge directly from itself to the minimum neighbor.

To understand the memory requirements for the DHT, note that we do not need to store the labels of the local maxima, i.e. nodes all of whose neighbors have smaller labels. Once a node is a local maximum, it remains a local maximum since any additional edges will be created to nodes with smaller labels. After the inner loop is run to completion every node is either a local maximum or a local minimum. Furthermore, from Lemma 5 we know that the number of local minimum nodes is reduced by a constant factor every two phases.

As a result, after $2C$ phases the number of non local maxima nodes is at most $\frac{n}{2^C}$, therefore by using a DHT of size $\frac{n}{2^C}$, we can implement the algorithm in $\log \frac{n}{2^C}$ rounds of MapReduce (one round for each phase). Before getting to this phase, each inner loop costs at most $\log n$ rounds of MapReduce, and therefore the number of MapReduce rounds before this phase is at most $O(2C \log n)$. Therefore, using a DHT of size $\frac{n}{2^C}$, we can compute connected components in $O(C \log n) + O\left(\log \frac{n}{2^C}\right)$ rounds of MapReduce. So, for example, by setting $C = \log\log n$ we get that our algorithms computes connected components in $O(\log n \log\log n)$ rounds of MapReduce using a DHT of size $O\left(\frac{n}{\log n}\right)$.

3.4 Communication cost

MapReduce algorithms are often compared by only looking at their number of rounds. However, an important component of the running time is the total communication between rounds, and having smaller communication leads to noticeable improvements in the running time.

First, note that the communication complexity of each round of large-star or small-star is proportional to the number of edges in the graph, in particular we have that at most $2m$ messages are exchanged between machines when we execute one of the two operations. Furthermore by Lemma 2 and 3 we know that during the algorithm the number of edges in the graph does not increase so we get that the total communication cost is $O(m \log^2 n)$.

Experimentally, however, we found the cost to be much smaller, since the number of edges steadily decreases, and in the end all of the algorithms end up with graphs on no more than $n-1$ edges. We explore this empirically in Section 5.7, but here show the same result analytically. We restrict our attention to $G(n,p)$ random graphs, i.e. those where each of the possible $\binom{n}{2}$ edges is present independently with probability p.

Lemma 9. *Let $G = (V,E)$ be a randomly generated $G(n,p)$ graph. Then after one round of large-star the resulting graph $G' = (V,E')$ is such that $|E'| \in O(\min(n^2 p, n^{3/2} \log n))$ with high probability.*

Proof. By Lemma 2 we already know that $|E'| \leq |E|$, and well known concentration results show that $|E| \in O(n^2 p)$ with high probability for $G(n,p)$. Hence $|E'| \in O(n^2 p)$. Note that if $p < 1/\sqrt{n}$ then $O(\min(n^2 p, n^{3/2} \log n)) = O(n^2 p)$, and we are done. Hence we only need to consider $p \geq 1/\sqrt{n}$.

Consider any pair of nodes m, v with $\ell_m < \ell_v$. If $(m,v) \in E'$ then there exists a node u with $\ell_m < \ell_u < \ell_v$, such that both:

- $(u,v) \in E$ and $(m,u) \in E$, and
- $m = min(u)$, i.e. u is not connected to any node m' such that $\ell'_m < \ell_m$.

Consider the number of nodes that are local minima in G. Note that the probability that the node with label i is a local minimum is $(1-p)^{i-1}$. Hence, the expected number of local minima is $\sum_{i=1}^{n} (1-p)^{i-1} \in O(p^{-1})$, and an application of Chernoff bounds shows this quantity is sharply concentrated around the mean. Moreover, each node has degree at most $O(np)$, therefore the number of minima adjacent to any node is at most $O(n)$ with high probability.

Map $\langle u; v\rangle$:
if u is marked as a root node **then**
  Emit $\langle v; u\rangle$.
else if $|\Gamma(u)| > \tau$ and u is not a copy **then**
  Hash v to $i \in \{1, 2, \ldots, \tau\}$.
  Make a copy u_i of u having label $\ell_{u_i} = \ell_u + i\varepsilon$ where $\varepsilon > 0$ is infinitesimally small
  Emit $\langle u; u_i\rangle$ and $\langle u_i; v\rangle$.
else
  Emit $\langle u; v\rangle$ and $\langle v; u\rangle$.
end if
Reduce $\langle u; \Gamma(u)\rangle$:
  Let $m = \arg\min_{v \in \Gamma(u)} \ell_v$.
  If $m = u$, mark u as a root node.
  Emit $\langle v; m\rangle$ for all v where $\ell_v > \ell_u$.

Algorithm 5: The large-star-optimized operation

Now consider the second case. We show that for any $p \geq 1/\sqrt{n}$, there are at most $O(\sqrt{n}\log n)$ nodes m that serve as minima for all of the nodes in the graph. In other words, there is a set of nodes $M \subseteq V$ with $|M| \in O(\sqrt{n}\log n)$ such that for any node u, $\arg\min_{v \in \Gamma^+(u)} \ell_v \in M$. Note that this implies the Lemma because each such node can have degree at most n. To prove the claim, observe that with high probability the label of any node acting as a minimum is at most $3\sqrt{n}\log n$. Since the edges are chosen uniformly, the probability that any node is only connected to nodes with a higher label is:

$$\left(1 - \frac{3\sqrt{n}\log n}{n}\right)^{pn} \leq \exp(-3\sqrt{n\log n} \cdot p) \in o(1/n^3).$$

A union bound over all n nodes, implies that this holds for the full graph with high probability. Therefore every node can only be connected to a node of id at most $O(\sqrt{n}\log n)$ and the result follows. □

3.5 Load balancing

It is well known that a lot of real world large datasets exhibit skewed non-uniform distributions. Any practical algorithm must take the degree skew into consideration, since a naive parallelization scheme will suffer from "the curse of the last reducer," where a single machine becomes a bottleneck for the whole computation [30].

In our case, the culprit is the large-star operation. Since repeated invocations of large-star result in star graphs, the root nodes of these stars have increasingly high degree. Indeed, the final output of both the Two-Phase and Alternating algorithms is a star for each connected component. Recall from Algorithm 2, that in the mapreduce implementation, the reducer with key v receives the entire neighborhood $\Gamma(v)$ of v. Therefore in the final stages, the reducer corresponding to the node with the lowest label will receive the entire connected component! In this section, we propose large-star-optimized, an optimized version of large-star than ensures

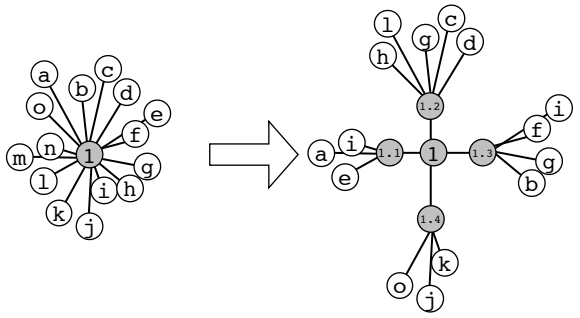


Figure 2. An example in large-star-optimized operation creates several copies of a high degree node for load balancing.

each reducer looks at a bounded number of (key, value) pairs independent of the size of the largest connected component. The algorithm is given in Algorithm 5.

The large-star-optimized algorithm differs from large-star in two ways. Recall that large-star emits $\langle u; v\rangle$ and $\langle v; u\rangle$ for all edges (u, v) of the graph. This ensures that a node u gets the entire neighborhood $\Gamma(u)$ in the reducer. There are two cases when large-star-optimized algorithm does not emit $\langle u; v\rangle$ in the mapper.

The first case is when u is a root, i.e. the minimum id node m in its neighborhood is u itself. In this case, conceptually large-star operation at u just needs to connect $m = u$ to all nodes $v \in \Gamma(u)$ such that $\ell_v > \ell_u$. So it is enough to emit $\langle v; u\rangle$ in the mapper to ensure this connection. We do not need to emit the reverse, i.e. $\langle u; v\rangle$, as the minimum at u would be m and reducer at u will only connect $m = u$ to v again, adding no extra information. This optimization needs to maintain for each node u whether it is a root. This is easily done at the reduce step with a simple check of $m = u$.

The second case when large-star-optimized does not emit $\langle u; v\rangle$ is when u is not a root, but $|\Gamma(u)|$ is larger than some threshold parameter τ. The parameter τ explicitly bounds the maximum number of values a reducer can get. For nodes with large neighborhoods, we make τ copies of the node u, with the i^{th} copy u_i having label $\ell_{u_i} = \ell_u + i\varepsilon$, where $\varepsilon > 0$ is infinitesimally small. We hash v to a $i \in \{1, \ldots, D\}$ and emit $\langle u; u_i\rangle$ and $\langle u_i; v\rangle$ connecting u to u_i and u_i to v. Note that this ensures that the maximum degree of any node is $\max(\tau, n/\tau)$. We show this change pictorially in Figure 3.5

We call the variant of the Alternating algorithm, which invokes large-star-optimized instead of large-star as Optimized Alternating. We can prove similar convergence properties for Optimized Alternating as Alternating. The only difference is that it ensures load-balanced computation while doing map-reduces. This advantage is clear when we look at the empirical performance of the two algorithms in section 5.

3.6 Performance and network topologies

In the previous sections we have shown several nice properties of the alternating algorithm, here we study the perfor-

mance of the algorithm when the input network has some specific property. In particular we will show that for dense $G(n,p)$ graphs after two rounds of the outer loop the algorithm will converge. This implies that the simple Two-Phase algorithm would take $O(\log n)$ rounds of MapReduce to converge for dense $G(n,p)$ graphs and that the Two-Phase algorithm with DHT service would converge in only 2 rounds(as we will observe for RMAT graphs).

Lemma 10. *Assume $G = (V,E)$ is a randomly generated $G(n,p)$ graph where each node has a distinct random label between $1 \ldots n$. Then, with high probability if $p > 5\frac{\log n}{n^{3/4}}$, the Two-Phase algorithm converges after two rounds of the outer loop.*

Proof. We say that a node is a local minimum if it has the minimum id of its neighbors. To prove the statement we first show that with high probability each local minimum is connected to the absolute minimum id node through either a path of length 2 or through a path of length 4. Moreover, such a path has the node with he maximum id in the center of the path. Then we show that this property is sufficient for the proof.

Let u be the absolute minimum node and v be a local minimum. Since a local minimum node is not connected with any node with label smaller than itself, the probability that a node with label bigger than $2n^{3/4}$ is a local minimum is $(1-p)^{2n^{3/4}} = \left(1 - \frac{\log n}{n^{3/4}}\right)^{2n^{3/4}} = e^{-2\log n} = n^{-2}$. So with high probability every local minimum has label at most $2n^{3/4}$.

Now we restrict our attention to the neighbors of a local minimum with label smaller than $\frac{n}{2}$, an application of the Chernoff bound shows that with high probability every local minimum node has at least $2n^{1/4}\log n$ nodes of degree bigger than $2n^{3/4}$ and smaller than $\frac{n}{2}$. If any one of these nodes is also a neighbor of u, we are done, otherwise we use those nodes as intermediate nodes in the path of length 4 between u and v.

Consider now any node with label bigger than $\frac{n}{2}$. The probability that such a node is not connected to any neighbor of v with degree between $2n^{3/4}$ and $\frac{n}{2}$ is $(1-p)^{2n^{1/4}\log n} = e^{-\frac{2\log^2 n}{\sqrt{n}}} \leq 1 - \frac{2\log^2 n}{\sqrt{n}} + \frac{2\log^4 n}{n}$. Thus the probability that it is connected to at least one is $\frac{2\log^2 n}{\sqrt{n}} - \frac{2\log^4 n}{n}$. So, using the fact that there are $\frac{n}{2}$ nodes of degree $\frac{n}{2}$, with high probability there are more than $\sqrt{n}\log n$ nodes with degree bigger than $\frac{n}{2}$ that are connected to v through a node of degree between $2n^{3/4}$ and $\frac{n}{2}$.

But now what is the probability that none of those $\sqrt{n}\log n$ is connected through an intermediary of degree between $2n^{3/4}$ and $\frac{n}{2}$ to u? Recalling that u has $2n^{1/4}\log n$ neighbors with such a degree we get $\left((1-p)^{2n^{1/4}\log n}\right)^{\sqrt{n}\log n} \in o(1)$. Thus with high probability we have such a path between u and any local minimum v.

It remains to show that this property is sufficient to ensure convergence in two outer loop steps. First note that at the end of a large-star iteration followed by a small-star operation all the edges are incident to at least one node that was a local minimum at the beginning of the phase.

In the following we show that at the beginning of the second iteration only the global minimum is a local minimum. In fact take a local minimum node at the beginning of the algorithm, if it has a node with label smaller than its label at distance 2 after a small-star operation it will no longer be a local minimum. So we can focus only on nodes that have minimum label in their distance 2 neighborhood. All of these nodes have a path of length 4 to the minimum, furthermore this path has the maximum id node in the center of the path, so after a round of large-star they will be a distance 2 from the minimum and thus after a round of small-star they will be connected to it. So after an iteration of the two-phase algorithm only the minimum will be a local minimum and the claim follows. □

4. Incremental Algorithms

While some real world applications are well tuned to the batch processing paradigm, others ask for an answer in a more incremental setting. One particular model that has been further explored in the data streams community assumes that all of the nodes are known apriori, and edges arrive over time. The algorithms we propose are readily adaptable to this setting, as we denote below. However, this is not the case In a richer model, where edge removal operations are also allowed. In such a setting, the algorithms we propose cannot be restarted from the end state, and must be rerun from the beginning.

For the addition setting, it is easy to detect if an edge addition is between two nodes in the same connected component (connected to the same central node), in which case no further action is needed. If, on the other hand, the newly arrived edge connects two distinct components, then a large-star operation followed by a single small-star operation is guaranteed to reach convergence to the new set of connected components.

5. Empirical Evaluation

In this section, we present an empirical evaluation of our algorithms for connected components. We ran all of our jobs three times on a shared production cluster with commodity hardware, and report the median running time in the results below. We report the total running time including the setup time of machines. The setup time accounts for approximately 1% to 10% of the total running time, and this percentage decreases as the size of the graph increases. Since the actual times vary greatly with the configuration of the cluster and its load (which depends on other jobs running in the cluster), instead of reporting the actual running times, we will report the relative performance either to the best

Network	# nodes	# edges
LiveJournal authorship , [1]	4,847,571	68,993,773
Friendster friendship [1]	65,608,366	1,806,067,135
Patent citation [1]	3,774,768	16,518,948
Twitter Followers [18]	35,689,148	41,652,230
UK web subgraph [2]	105,896,555	6,603,753,128
Google+ subgraph	177,878,516	2,917,041,952
Orkut social subgraph	157,546,418	17,474,363,130
Related entity subgraph	31,386,984	3,773,675,522
Keyword similarity subgraph	371,441,631	3,514,501,483
Document similarity subgraph	4,697,876,565	452,638,340,365
RMAT 22	2,396,986	128,311,950
RMAT 24	8,871,645	520,757,402
RMAT 26	32,803,311	2,103,850,648
RMAT 28	121,228,778	8,472,338,793
RMAT 30	447,774,395	34,044,283,063
RMAT 32	1,652,692,682	136,596,059,559
RMAT 34	6,097,235,142	547,511,932,254

Table 2. Sizes of the graphs used for algorithm evaluation.

known baseline (the Hash-to-Min algorithm in [26]), or, for scalability experiments, the running time on smaller graphs. When reporting speedup with respect to the Hash-to-Min algorithm, we make sure that all the algorithms are run simultaneously to take into account the congestion of the cluster. We first describe the data sets used in our empirical study, and then report the empirical results.

5.1 Data

We ran our algorithms on four categories of graphs, the details for all can be found in Table 2.

- A collection of publicly available graphs including a LiveJournal authorship graph [1], a Twitter user follower graph [18], a Friendster friendship graph [1], a patent co-authorship graph [1], and a snapshot of the UK web subgraph [2].
- A collection of networks associated with various Google projects including subgraphs of the Orkut social network and Google+[2], document and entity similarity subgraphs, and a keyword-query similarity subgraph.
- A collection of eight RMAT benchmark graphs that are randomly generated graphs with increasing number of nodes and edges[4].
- A long path graph of length 100M nodes: We use this graph as a baseline example to examine sensitivity of the running time of our algorithms to large diameter.

In all of these graphs, there exists a giant connected component that includes a majority of nodes, and in most cases several smaller connected components. For instance, the largest connected component has 4,607,785,638 nodes for Document Similarity, 104,288,749 for UK web, 65,608,366 for Friendster and 6,093,534,405 for RMAT34.

[2] The social network sub-graphs we use are anonymized, and the connectivity in the sub-graphs does not reflect on the general connectivity in the complete graph.

We will use the first two collections to demonstrate the performance of our algorithms on real world graphs. On the other hand, the synthetically generated path and RMAT graphs will allow us to evaluate the scalability of our approaches.

RMAT Graphs. RMAT is a recursive model of randomly generating graph with several desirable properties such as power-law degree distribution property, small world property, and inclusion of many dense bipartite subgraphs [4]. To generate an RMAT graph, one recursively subdivides the adjacency matrix in four equal quadrants and elects to recurse on one of the four quadrants with unequal probability (a, b, c or d). We generate seven graphs from this family with $2^{22}, 2^{26}, \ldots, 2^{34}$ nodes by setting the parameters $(a, b, c, d) = (0.57, 0.19, 0.19, 0.05)$.

5.2 Algorithms

We compare performance of four different algorithms for computing connected components: three pure MapReduce implementations and a MapReduce implementation with an additional distributed hash-table(DHT) service. These are:

- Hash-to-Min [26]: This is the algorithm with the best practical performance from [26][3].
- Alternating Algorithm: This is the algorithm that alternates between small-star and large-star as described in Section 4.
- Optimized Alternating Algorithm: This an implementation of Alternating Algorithm with an additional optimization for load balancing as described in Section 3.5.
- Two-Phase Algorithm: This is an implementation of the two-phase algorithm where the implementation of the outer loop is based on a pure MapReduce implementation and the inner loop of the two-phase algorithm is implemented using a read-only DHT as described in Section 3.3.
- Pregel: This is an implementations is based on Boruvka's distributed MST algorithm in the Pregel framework [21]. Note the Pregel framework allows us to run algorithms with many more rounds, since each round tends to run much faster in Pregel.

5.3 Running times

In this section, we report performance of our algorithms compared to the performance of Hash-to-Min, the best algorithm from [26]. All of the algorithms were implemented in ran on the same shared cluster and use the same number of resources (virtual machines, memory, etc.).

Figure 3 shows the relative speedup of each of the algorithms over the Hash-To-Min baseline. On large graphs the

[3] We also examined other baseline implementations such as Pegasus or Hash-Greater-to-Min discussed in [26], however, as observed in [26] we confirm that empirically Hash-to-Min works best among these algorithms, and use it as the strongest baseline in this paper.

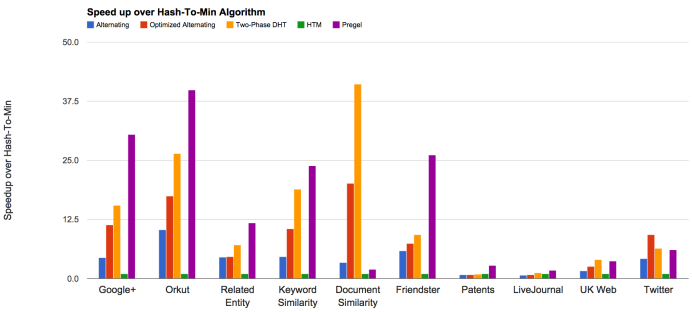


Figure 3. Speedups for real-world graphs. For the biggest RMAT graphs, we also got 10 to 30 times speedup from Hash-to-Min to the two-phase implementation.

proposed algorithms easily improve upon past approaches, with pure MapReduce algorithms being two to *fifteen* times faster, with the two-phase algorithm using DHT obtaining speedups of almost 30x on very large graphs. Note that while on relatively small graphs such as LiveJournal and the Patent citation graph, which only have a few million nodes the speedup is negligible. However, since these graphs can easily fit into memory on a modern machine, distributed computing is not be the best tool for the job. We observe that the speedup is related to the size of the network, and the speedup for the two-phase algorithm is larger for graphs with large diameter, e.g., for Friendster graph with diameter 32 [1]. In terms of absolute running time, for some big graphs, the actual running time went down from more than a day to a couple of hours, or from a few hours to tens of minutes, when running on several hundreds of machines.

5.3.1 Comparison to the BSP paradigm (Pregel)

While the main purpose of our study is to understand the performance of computing connected components in mapreduce, we present a brief comparison of our algorithms with two implementations of the connected component algorithms in the BSP paradigm [31] via Pregel [21].

Before presenting the quantitative results, we note that MapReduce and Pregel each have their pros and cons as a distributed computing framework. While Pregel, and the open source variants like Giraph [6][4], are generally considered a more efficient framework for graph processing they do not handle machine failures and pre-emption as effectively as the MapReduce framework. For example, in order to achieve some level of fault-tolerance, Pregel needs to store on disk the state of the run after every few rounds of computation. This makes it less robust against failures. On the other hand, the MapReduce framework is designed to handle failures effectively.

Moreover, the basic implementation loads the adjacency lists of all nodes into memory (or allows random access to them), and therefore in very large real-world graphs with a skewed degree distribution, this presents an issue when dealing with high degree nodes. As a result, graph mining tools that are based on MapReduce tend to run easier with limited number of resources under machine pre-emption and failures. All these reasons make the MapReduce-based implementations more appropriate for a general-purpose graph mining library in certain environments with a limited number of machines and with machines with limited memory.

We compared the running time of the optimized two-phase algorithm with the running times of these implementations in Pregel while using the same number of machines and the same memory resources. We first observe in Figure 3 that for a handful of small/medium size graphs (e.g., the Twitter, Patent citation, and Friendster graphs) with less than 2B edges, Pregel's implementation ran faster (by almost a factor of two). However, as the size of the graphs increases, the relative performance of the two-phase MapReduce implementation steadily improves. For example, for the document graph with more than 400B edges, the Pregel implementation runs almost thirty times slower. We also note that for graphs with more than 100B edges, the Pregel implementations do not easily run with the same number of machines and the same resources, and we need to increase the number of machines and resources to be able to run them until completion.

5.4 Number of Rounds

In this section we report the number of rounds each of the algorithms took on each of the datasets, the results are shown in Figure 4. We note that while the number of rounds is a useful shorthand when comparing different algorithms, the figure alone does not tell the full story.

First, the number of rounds of the Hash-To-Min algorithm is always smaller than that of the Alternating algorithm, yet the Hash-To-Min algorithm is slower in practice. We explore this in detail in Section 5.7. Moreover, the optimized alternating algorithm typically takes more rounds than the non-optimized version, however since each of the individual rounds is faster (because of load balancing) it is faster in practice.

Finally, we note that the total number of rounds of the two-phase algorithm that uses the DHT service is much smaller, which is not surprising since it only requires a MapReduce round when using the outer loop of the algorithm.

5.5 Scalability of different Implementations

To investigate the scalability of our algorithms we turn to the synthetically generated RMAT graphs. We show the number of rounds used by each algorithm in Table 3. Even as the graphs get larger, the number of rounds does not significantly change, and stay much lower than $O(\log^2 n)$ suggested by the theory.

[4] Note that for the experiments we used the internal version of Pregel to have a fair comparison with the MapReduce framework.

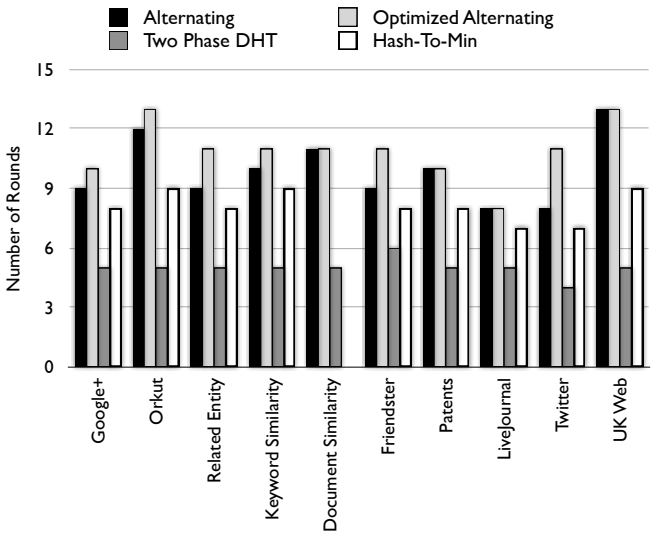


Figure 4. Number of Rounds taken by various algorithms.

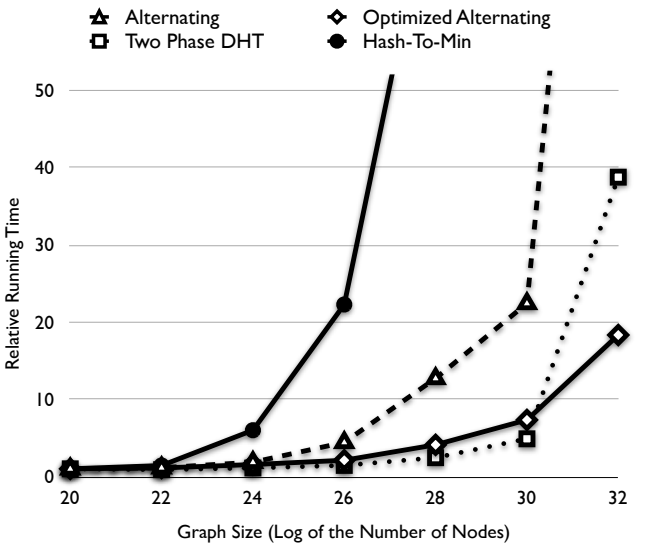


Figure 5. Running time as a function of graph size.

	Hash-To-Min	Alternating	Optimized Alternating	Two-Phase DHT
20	5	5	6	2
22	5	5	6	2
24	5	6	6	2
26	5	6	6	2
28	6	6	7	2
30	-	6	7	2
32	-	6	7	2

Table 3. The number of rounds taken by different algorithms on RMAT graphs. A - indicates that the algorithm did not finish in more than a day on the dataset

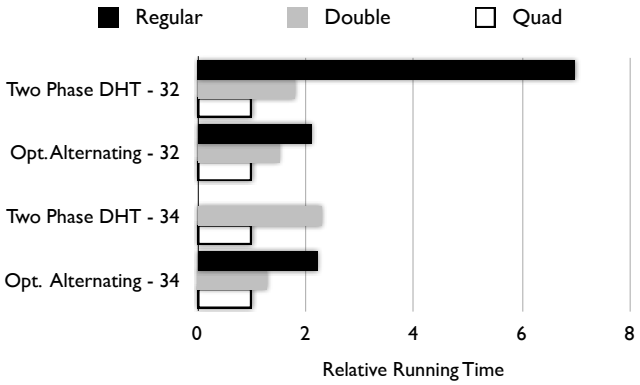


Figure 6. The relative speed of algorithms as we change the resource sizes.

However, while the number of rounds stays the same, the time per round changes dramatically. We plot the running time versus the log of the number of vertices in Figure 5.

A straight line on this graph represents an algorithm whose running time is logarithmic in the number of vertices, $O(\log n)$. The baseline algorithm, has a fast growing running time, as we will show below in Section 5.7, this is largely due to graph densifying over time, resulting in longer and longer times for the shuffle step.

This is remedied by the alternating algorithm, which guarantees that the overall number of edges does not increase in each iteration. The algorithm scales better, but runs into a different performance bottleneck, that of high degree nodes. As we described in Section 3.5 the bottleneck in each round of the algorithm lies in updating the edges adjacent to high degree nodes (this step is proportional to the degree of the node). Since the end state of the algorithm is a single star graph for each connected component, the unoptimized alternating algorithm will eventually hit this performance bottleneck.

We can see however, how the additional optimizations overcome this bottleneck. The optimized alternating algorithm breaks up high degree nodes, which greatly reduces its running time. The same effect is achieved by running the inner loop using a read-only DHT service, also reducing the overall congestion.

5.6 Resource Usage

In Figure 6 we demonstrate the performance of the different algorithms as we double and triple the number of resources (cores) available. We plot the relative running times of each approach on the RMAT-32 and RMAT-34 graphs.

A perfectly work efficient algorithm would take twice as much time when we halve the number of resources, and three times as much when we reduce the number of resources by a factor of three. We observe that for the most part our algorithms are work efficient. The exception comes when we compute large graphs on small resources; here the slow down is nonlinear due to machines getting overwhelmed.

For example, the Two-Phase DHT algorithm does not finish on the regular number of resources on the largest RMAT-34 graph because of the DHT getting overwhelmed. Thus we see only two of the three bars in Figure 6 for the Two-Phase DHT algorithm.

5.7 Evolution of Graph Sizes

As we mentioned in Section 5.5, a major advantage of the alternating algorithm over Hash-to-Min is the fact that it is guaranteed to reduce the number of edges (and thus the total communication) in each iteration. To demonstrate this, we plot the total number of edges in the graph as a function of the iteration number for each of the algorithms in Figure 7.

Notice that the Hash-To-Min algorithm *more than doubles* the number of edges in the initial phases of the graph, which results in poor scalability. On the other hand both the Alternating and the Optimized Alternating algorithms quickly reduce the overall number of edges. Note that the y-axis is log-scaled and the decrease in the first few iterations for Alternating and Optimized Alternating is quite drastic: in each of the first 4 to 6 rounds the number of edges almost halves, and then converges to a value which is linear in the number of nodes.

5.8 Sensitivity to Diameter

The number of MapReduce rounds and therefore the running time of our algorithms for computing connected components should increase as the diameter of the graph increases. We observe this phenomenon in our real-world graphs, e.g., the number of rounds was higher for the Friendster network which has large diameter of 32 compared to other graphs.

We study this further by running our algorithms on a Path graph with random node IDs and present the results in Table 4. We observe, as in Section 5.4 that the number of rounds for the pure MapReduce algorithms is much larger than that for the Two-Phase DHT algorithm, which results in the Two-Phase DHT algorithm being almost twice as fast as the pure MapReduce implementations. As before, the Optimized Alternating algorithm takes additional rounds to process large degree vertices, and thus needs more rounds than Alternating. (Although the initial graph is a path, as it is slowly transformed into a star, the maximum degree monotonically increases.) The loss in the number of rounds is made up for in the total running time, where the algorithm is only 2% slower.

Algorithm	Speedup	# Rounds
Alternating	1.50	39
Optimized Alternating	1.53	47
Two-Phase DHT	2.86	12

Table 4. Speedup on path w/ seq. node ID compared to Hash-to-Min, and # MR rounds

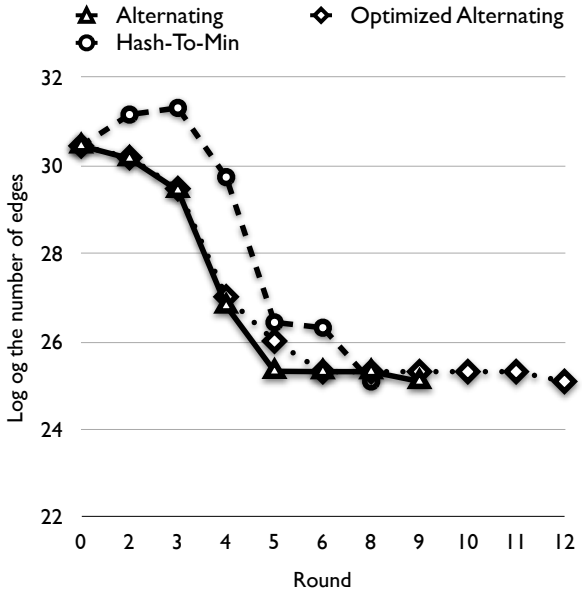


Figure 7. The number of edges remaining during the execution of the algorithms on the Twitter graph.

6. Conclusion

In this work we presented new parallel connected components algorithms that outperform the current state of the art approaches by more than an order of magnitude on many diverse datasets. Although the number of rounds of a MapReduce algorithm has served as a standard evaluation metric, we showed that total communication and the degree to which the computation is load balanced also play significant roles. Finally, we showed how to combine the existing MapReduce framework with an distributed hashtable (DHT) to get an additional boost in the algorithms' performance. Many interesting questions remain. One open question is to further improve the complexity of the connected component algorithms. A different direction is to further investigate the symbiotic relationship between MapReduce and a DHT to see what other algorithms can be improved by combining both of these systems. Finally, as we mentioned in the introduction, the question of finding connected components is a basic building block for many graph mining algorithms; adapting these to the MapReduce framework is a challenging research direction.

References

[1] Stanford large network dataset collection. http://snap.stanford.edu/data/index.html.

[2] Temporal evolution of the uk web. http://law.di.unimi.it/webdata/uk-2007-05/.

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

[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *In Fourth SIAM International Conference on Data Mining (SDM)*, 2004.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.

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

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

[8] E. Dahlhaus. Parallel algorithms for hierarchical clustering and applications to split decomposition and parity graph recognition. *J. Algorithms*, 36(2).

[9] C. Doll, T. Hartmann, and D. Wagner. Fully-dynamic hierarchical graph clustering using cut trees. In *WADS*, 2011.

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

[11] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching, and simulation in the mapreduce framework. In T. Asano, S.-I. Nakano, Y. Okamoto, and O. Watanabe, editors, *ISAAC*, volume 7074 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2011.

[12] H. il Koo and D. H. Kim. Scene text detection via connected component clustering and nontext filtering. *IEEE Transactions on Image Processing*, 22(6), 2013.

[13] U. Kang and C. Faloutsos. Big graph mining: algorithms and discoveries. *SIGKDD Explorations*, 14(2), 2012.

[14] U. Kang, M. McGlohon, L. Akoglu, and C. Faloutsos. Patterns on the connected components of terabyte-scale graphs. In *ICDM*, 2010.

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

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

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

[18] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International World Wide Web (WWW) Conference*, 2010.

[19] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, pages 31–46, 2012.

[20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[21] 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.

[22] C. F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21, 1995.

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

[24] S. Rajasekaran. Efficient parallel hierarchical clustering algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 16(6).

[25] 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.

[26] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce in logarithmic rounds. In *ICDE*, 2013.

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

[28] T. Seidl, B. Boden, and S. Fries. Cc-mr - finding connected components in huge graphs with mapreduce. In *ECML/PKDD*, 2012.

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

[30] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[31] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.