# Shortcutting Label Propagation for Distributed Connected Components

Stergios Stergiou*
Google
Mountain View, CA, USA
utopcell@google.com

Dipen Rughwani
Yahoo Research
Sunnyvale, CA, USA
dipen@yahoo-inc.com

Kostas Tsioutsiouliklis
Yahoo Research
Sunnyvale, CA, USA
kostas@yahoo-inc.com

## ABSTRACT

Connected Components is a fundamental graph mining problem that has been studied for the PRAM, MapReduce and BSP models. We present a simple CC algorithm for BSP that does not mutate the graph, converges in $O(\log n)$ supersteps and scales to graphs of trillions of edges.

## CCS CONCEPTS

• **Information systems → Graph-based database models**; • **Information systems → Computing platforms**; • **Theory of computation → Distributed algorithms**; Graph algorithms analysis;

## KEYWORDS

connected components, label propagation, LPSC, graph algorithms, distributed systems

## 1 INTRODUCTION

Computing connected components of a graph lies at the core of many data mining algorithms and is a fundamental operation in graph clustering. Sequentially, connectivity can be solved optimally in linear time using breadth-first or depth-first traversals. The problem has been studied extensively in the past in the parallel context [3, 5, 10, 16, 17, 19, 21, 25, 28, 30, 32, 35, 37, 39, 41, 43, 45, 46, 49]. Algorithms have been presented for shared-memory CPUs [6, 7, 29, 40, 44, 47, 54], distributed memory systems [8, 13, 14, 38] and GPUs [9, 27, 55]. Many polylogarithimic-depth parallel connectivity algorithms have also been proposed [18, 22–24, 48, 50]. In [53], a particularly efficient, albeit $O(\log^3 n)$-depth single-machine algorithm is presented that can compute CCs on graphs of 500 million edges in a few seconds. Perhaps the most influential works in parallel CCs are [4, 52]. More recently, very large-scale CC algorithms have been proposed [36, 42]. In particular, [36] reports results on graphs of up to 550 billion edges. However the reported algorithms are either presented on impractical models (such as PRAM), for single machines, or require new graphs to be generated after each iteration. In this paper we present an algorithm for the BSP model that converges in $O(\log n)$ supersteps, which is optimum and does not mutate the input graph, which allows us to scale to graphs of trillions of edges. We compare against many published algorithms [33, 34, 36, 42, 51] and report many orders of magnitude speedups while also scaling to graph sizes never before reported.

*work done while author was with Yahoo Research

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*WSDM 2018, February 5–9, 2018, Marina Del Rey, CA, USA*
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5581-0/18/02…$15.00
https://doi.org/10.1145/3159652.3159696

## 2 GRAPH PROCESSING FRAMEWORK

The proposed algorithm is implemented on Hronos, a Yahoo proprietary in-memory / secondary-storage hybrid graph processing system, developed in C++11 on top of Hadoop MapReduce. Hronos has been used for many internal applications and has provided significant speedups for classic Machine Learning algorithms [56]. Graph nodes, and their related information including edges, are randomly placed on a set of partitions. Node-related state is maintained in memory, while the graph itself, specifically a serialization of the edges, can either be stored in memory or on disk. Messages originating from a source process are transparently buffered before being sent to target processes, with buffering occurring separately for each target partition. Each message contains an application-specific payload. The framework can optionally [de]compress buffers on-the-fly, transparently to the application. Communication across partitions is performed via a large clique of unidirectional point-to-point message queues.

The system builds on the following low-level communication primitives: clique communication for graph level computations; and one-to-all and all-to-one communication for partition-level computations. It implements loading and storing of vertex information through which it supports fault tolerance via check-pointing. Computation proceeds in supersteps. Each superstep is executed as a sequence of three primitives: a one-to-all communication primitive that sends initialization parameters to all peers, a clique communication primitive that transmits edge messages, and an all-to-one communication primitive that gathers partition-level results from the peers to the master partition. Within each superstep, each partition independently sends/receives messages to/from other partitions. Messages towards a particular node can either be unidirectional, or optionally request callback information from that node. The superstep completes when all sent messages have been processed and all partitions have signaled termination.

## 3 PROPOSED ALGORITHM

Let $G = (V, E)$ be an undirected graph on $n = |V|$ nodes and $m = |E|$ edges. We assume that nodes are totally ordered; for simplicity of presentation, $V = \{1, \ldots, n\}$. Let $N(v) = \{u|(v, u) \in E\}$ be the neighbors of $v$ and $N^+(v) = N(v) \cup \{v\}$. Let $\Delta$ (diameter) be the

length of the longest shortest path between any two nodes in the graph. Let $l_i : V \to V, i \in \mathbb{N}$ be functions on $V$, or labels associated with nodes, where $l_0(v) = v, \forall v \in V$ is the identity function. The Connected Components (CC) problem aims to assign labels $c(v)$ to each node $v \in V$ such that $c(u) = c(v)$ if and only if a path exists between $u$ and $v$.

Simple graph traversal (for instance, breadth–first traversal) solves CC in $O(m)$ time in the sequential case: While there exists an unvisited node $v$, a breadth-first traversal starts from $v$ and propagates its label $l_0(v)$ to all nodes reachable from $v$. This algorithm suggests a similar parallel algorithm called *label propagation (LP)*: At each iteration $i$, each node $v$ independently and concurrently updates its label $l_i(v)$ to the minimum label among itself or its neighbors in the previous iteration: $\forall v \in V : l_i(v) = \min\{l_{i-1}(u) | u \in N^+(v)\}$.

While the sequential algorithm is optimum, LP examines every edge in the graph at each iteration and there can be up to $\Delta$ iterations before convergence. For example, consider its behavior on the following graph.

$i = 0$   1 2 3 4   5 6
$i = 1$   1 1 2 3   5 5
$i = 2$   1 1 1 2   5 5
$i = 3$   1 1 1 1   5 5

In graphs where $\Delta$ is small, LP is very efficient. However, as $\Delta$ can be $\Theta(n)$, LP is not generally a practical algorithm, especially for models of computation such as MapReduce where the startup cost for each iteration is high.

Let us consider the following graph mutating operation. At each iteration, each node $v$ adds edges $(u_1, u_2)$ for all nodes $u_1 \in N(v), u_2 \in N(v)$. In essence, every path $(u_1, v, u_2)$ of length 2 is shortened to a direct edge between $u_1$ and $u_2$. Clearly the nodes in a connected component will form a clique after at most $\lceil \log n \rceil$ iterations. A single LP iteration will then suffice to label all the nodes in the connected component with the smallest label in it. This classic operation is called *path halving* or *pointer jumping* [31]. It allows us to bound the number of iterations to $O(\log n)$, which is optimum, but it does so at a significant cost: In the worst case, the resulting graph will contain $\Theta(n^2)$ edges, while iterations will become progressively slower. Moreover, even if the number of edges did not increase prohibitively, mutating the graph itself is a costly operation, because it either requires that the graph is maintained in memory (for BSP) or that a new graph is generated at every iteration (MapReduce.) For example, observe the behavior of path halving on the line graph of 8 nodes on Figure 1. While the number of required iterations for convergence reduces from 7 to 3, the resulting graph contains $\binom{8}{2}$ edges, up from 7.

Following, we will construct an algorithm that solves CC and has the following desirable properties: (1) it converges in $O(\log n)$ iterations; and (2) it only performs a single linear scan over the edges of the graph per iteration, without mutating it. The first property allows us to minimize the iterations initialization cost as well as the communication cost. The second property is more crucial: Linear scans from secondary storage are very efficient (in stark contrast to random disk accesses.) Therefore we can maintain the graph

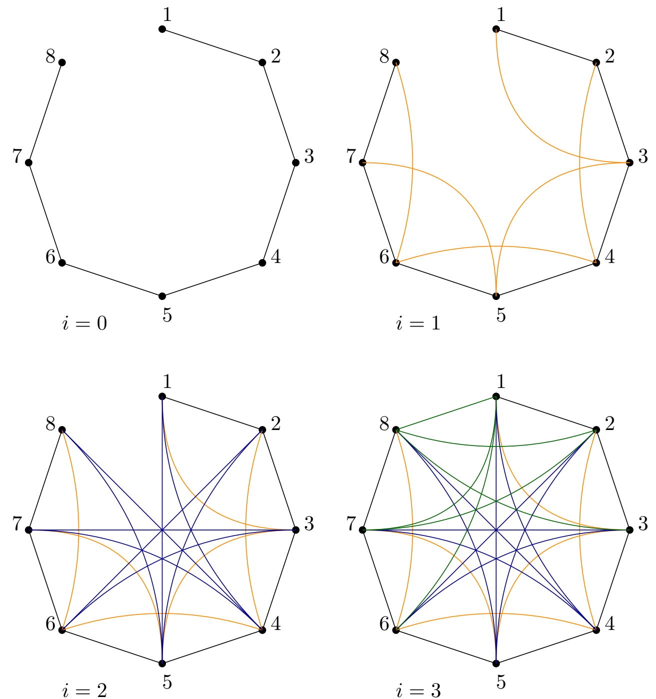



**Figure 1: Path halving on a line graph.**

on disk, which in turn allows us to scale CC to significantly larger graph instances than those reported in the literature.

The proposed algorithm is presented on Algorithm 1. It interleaves two supersteps until convergence. The first step is the equivalent of label propagation, executed in a manner that allows the graph edges to be processed sequentially. The second step modifies the node labels such that at most $O(\log n)$ label propagation iterations will suffice for convergence.

First, let us examine the PROPAGATE superstep which corresponds to LP. Instead of requiring that each node $v$ queries its neighbors for their label, we simply send its current label $l(v)$ to all nodes in $N(v)$. When a node $u$ receives a label message $m$ from a neighbor, it updates its label to the minimum of its current value and $m$. Before examining the second superstep, let us show that the algorithm correctly identifies the connected components of a graph.

THEOREM 3.1. *Algorithm 1 assigns unique labels for each connected component in the graph.*

PROOF. Les us assume that graph $G$ is comprised of $k$ connected components $cc_j \subset V, j = 1..k$. Initially each node is assigned a unique label. Let $m_j$ be the minimum label among all nodes in $cc_j$. The first superstep simulates a Breadth-First traversal starting from the nodes associated with labels $m_j$. For each connected component $j$, after at most $i$ iterations, all nodes reachable with paths of length $i$ from the node associated with label $m_j$, will update their label to $m_j$. Note that the second superstep never increases any of the labels. Additionally, none of the labels $m$ used to update $l(u)$ on Line 22 correspond to a node outside the respective connected component

**Algorithm 1** LABEL PROPAGATION WITH SHORT CUTTING

```
1:  function LPSC(G)
2:      ∀v ∈ V : l′(v) ← v
3:      repeat
4:          l ← l′
5:          PROPAGATE(G, l, l′)                      ▷ label propagation
6:          SHORTCUT(G, l, l′)                       ▷ short cutting
7:      until l = l′
8:      return l
9:  superstep PROPAGATE(G, l, l′)
10:     for all (u, v) ∈ E do
11:         send l(u) to v
12:     for all received messages m to u do
13:         l′(u) = min(l′(u), m)
14: superstep SHORTCUT(G, l, l′)
15:     for all u ∈ V do                             ▷ send messages
16:         if l′(u) < l(u) then
17:             send l′(u) to l(u)
18:         request l(l′(u)) from l′(u)              ▷ send requests
19:     for all requests from v to u do              ▷ receive requests
20:         send l(u) to v
21:     for all messages m to u do                   ▷ receive messages
22:         l′(u) = min(l′(u), m)
```

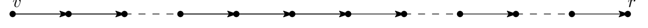
of $u$. Therefore their values will always be at least $m_j$ for all nodes in $cc_j$. □

Let us now examine the behavior of the SHORTCUT superstep. Initially, labels $l(v) = v$. During the execution of the algorithm, $l(v)$ is either decreasing or stays the same. Therefore, at any given point, a sequence of nodes $v, l(v), l(l(v)), \cdots, u$ will never repeat itself because of the invariant $v \geq l(v)$. We formulate the directed *parent graph* $G_p = (V, E')$ where $E'$ contains all edges of the form $(v, l(v))$. Each node has a single outgoing edge towards its "parent" node, and as shown before, the graph is cycle-free (ignoring self-loops.) Therefore $G_p$ is a forest graph. Before convergence, the nodes of each connected component on the graph form a set of trees. After convergence, each connected component will correspond to a star (a single tree of height 1.)

The SHORTCUT superstep performs a dual role. First, it performs pointer jumping on $G_p$ towards nodes with smaller labels (on Lines 18, 20 and 22.) In a single iteration, pointer jumping can reduce the height of a tree of height $h$ to $\lceil h/2 \rceil$. Therefore, an individual tree will be converted into a star in at most $\lceil \log n \rceil$ iterations. Second, whenever the label of a node $v$ gets updated to a smaller value $l'(v)$, node $v$ informs its previous parent $l(v)$ of the new value (on Lines 17 and 22.) Trees are connected through edges on the original graph, and therefore are merged together via PROPAGATE. Specifically, for a given edge $(u, v) \in E$, the subtree rooted at $l(u)$ will be attached to $l(v)$ if $l(v) < l(u)$ at some point during the execution of the algorithm. Following, we show that $O(\log n)$ iterations suffice for converting all trees to stars in the presence of tree mergings.

THEOREM 3.2. *Algorithm 1 converges in $O(\log n)$ iterations.*

PROOF. Let us consider the path that is followed by the algorithm that connects an arbitrary node $v$ to the root node $r$ of its connected component. As the sequence of labels $l(v)$ of $v$ is decreasing, the length of the path is bounded by $n$. Edges in $E'$ are solid, while edges in $E$ that are exercised by the algorithm are dashed.




All contiguous solid edges subpaths will be reduced to stars in at most $\lceil \log n \rceil$ iterations. During these iterations, dashed edges may be exercised, leading to merging of the corresponding trees. In the worst case, such tree merges will be delayed at most until all trees have been converted to stars. At that point, a single tree will be formed of depth at most $n$ (since there can only be at most $n$ trees.) An additional $\lceil \log n \rceil$ iterations are sufficient for converting this tree to a star. □

Following, we bound the number of messages processed by the algorithm until convergence.

THEOREM 3.3. *Algorithm 1 processes $O((m + n) \log n)$ messages until convergence.*

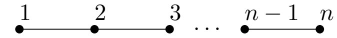
PROOF. Superstep PROPAGATE processes $O(m)$ messages while superstep SHORTCUT processes $O(n)$ messages and the algorithm converges in $O(\log n)$ iterations. □

Figure 2 depicts an example graph with $G_p$ presented at the beginning of the algorithm, and after each superstep of all iterations until convergence. We observe that initially, the single connected component is represented by 9 single-node trees. After the first PROPAGATE, the same connected component is represented by two trees that include nodes $\{3, 4, 5, 6, 7, 20\}$ and $\{8, 10, 15\}$ respectively. At convergence, a single star represents the connected component.

## 3.1 Optimizations

*3.1.1 Redundant Messages.* We observe that sending label $l(u)$ from node $u$ to node $v$ (Line 11 on Algorithm 1) can reduce $l'(v)$ (Line 13) only if the label of node $u$ has itself been reduced compared to the previous iteration. Otherwise, any possible effect of $l(u)$ to the label of node $v$ has already occurred in the previous iteration. Therefore, we only send messages on Line 11 when $l(u)$ has decreased. This optimization has a significant impact on execution times, especially during the later iterations of the algorithm in which most nodes in a connected component have already detected their minimum label.

*3.1.2 Label Bucketing.* Let us examine the behavior of LP on a line graph of $n$ nodes, where nodes have been assigned sequential ids.




The algorithm will converge in $n - 1$ iterations. However, many redundant label updates will be performed. In our example, node $i$ will be updated $i - 1$ times before it reaches its final value. If the algorithm knew beforehand the set $S$ of minimum labels corresponding to each of the connected components in the graph, it would be possible to restrict propagation (i.e. message sending on Line 11 on Algorithm 1) only to nodes whose labels are in $S$. Although this is not possible, it would be meaningful to assume that

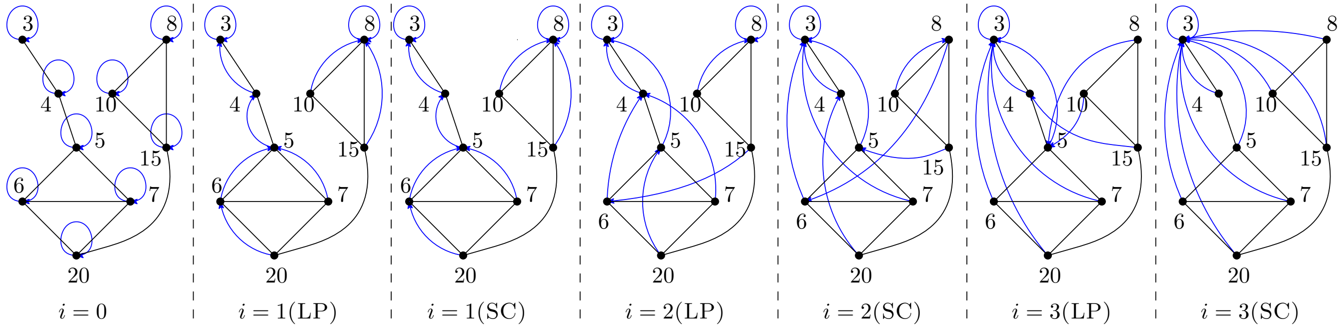
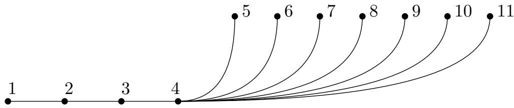


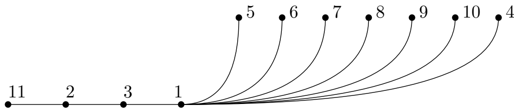

**Figure 2: Supersteps of the proposed algorithm on a simple graph of 9 nodes. Initially, the single CC is covered by 9 single-node trees. After the first superstep, the trees are reduced to 2. After convergence, graph $G_p$ is a single rooted star. Note that $SC$ at $i = 1$ never alters $G_p$ since pointer jumping occurs via $l$ (Lines 18 and 20.) The algorithm remains valid even if $l'$ is used instead.**

$S$ has a larger overlap with the set of $k$ smallest labels compared with a random subset of $k$ labels. In this optimization, we run the proposed algorithm twice: once restricting propagation to a small subset of the smallest labels $B$, and once afterwards without any restrictions. In our example, where $S = \{1\}$, $B$ could be set to $\{1, 2\}$, which reduces the total number of label updates to $2n - 3$ instead of $\binom{n}{2}$.

*3.1.3 Label Ordering by Node Degree.* Let us examine the behavior of LP for large degree nodes (for example, node 4 on the following graph.)




In all iterations until node 4 detects the minimum label, it will be propagating its current label to its neighbors. While this propagation happens for all nodes, it is particularly expensive for large-degree nodes. This optimization sorts nodes non-increasingly according to their degree and assigns initial labels in this order. Following is a possible assignment of labels for the example graph.




While this optimization seems promising, we have not investigated its impact on our algorithm and is presented here as future work.

*3.1.4 Label Request Aggregation at Source.* Algorithm 1, as presented, contains a bottleneck. Let us assume than the largest component on the input graph contains $\Theta(n)$ nodes. As the algorithm progresses, and the nodes in $G_p$ form a star rooted at the node $v$ with the smallest label, there will be $\Theta(n)$ request messages towards $v$ on Line 18. Although the total number of requests is always $O(n)$, in this case the majority of these requests will be sent towards a single partition (the partition that contains node $v$) and thus, the algorithm is serialized at that point.

We realize that this weakness can in fact be converted to an advantage for the proposed algorithm by performing the following optimization: We modify the algorithm to only send out requests for each unique $l'(u)$ on Line 18. We also maintain a list of nodes for each unique request target, that are updated as soon as the response (sent on Line 20) arrives. Consequently, only $O(p)$ request messages will be sent towards node $v$ instead of $\Theta(n)$, where $p \ll n$ is the number of partitions.

*3.1.5 Large in-Degree Node Detection.* We realize that request aggregations are only necessary for root nodes of large in-degree in $G_p$. Ideally we would like to apply aggregation only for those nodes. However, without global synchronization information, each partition does not know which are the large in-degree nodes on other partitions. Nevertheless, since nodes are randomly distributed across partitions, edges of the form $(v, r)$ in $G_p$ will also be randomly distributed among the $p$ partitions. Therefore, it suffices to detect the most frequent targets within each partition and apply request aggregation only for those. However, computing target frequencies requires $O(|uniqueTargets|)$ memory. Instead, we detect all targets that cross a particular local frequency threshold, by adopting a heavy hitters algorithm [12, 20]. Specifically, before executing superstep SHORTCUT at each iteration, we detect the $k$ (at most) targets whose frequency exceeds $n_p/k$ locally in each partition, where $n_p$ is the number of nodes in partition $p$, and apply request aggregation only for those. Heavy hitters is an efficient linear algorithm that requires $O(k)$ memory and $O(n_p)$ time.

# 4 EXPERIMENTAL RESULTS

We report experimental results for the graphs on Table 1. R-Mat graphs are synthetic power-law graphs [15] while Twitter [2], UK [11], BTC [26] and CommonCrawl [1] are publicly available real-world graphs. The Yahoo graph is a subset of Yahoo's web graph. All graphs are converted to bidirectional graphs: for every edge $(u, v)$, edges $(u, v)$ and $(v, u)$ are added and duplicate edges are removed. All results are collected on a shared cluster of 128GB RAM, Intel Xeon CPU E5-2620 v2 nodes (2x 12 cores) and 10Gbps interconnect. We note that due to the large number of experiments and their long execution times, average load on the cluster may vary, affecting execution times by a small factor.

We compare with algorithms AltOpt [36], CCF [34], Cracker [42], Hash2Min [51] and Pegasus [33], as implemented in Spark by

**Table 1: Graphs**

| graph | nodes | edges |
|---|---:|---:|
| rMat22 | 2,394,731 | 130,289,234 |
| rMat24 | 8,859,723 | 526,329,896 |
| rMat26 | 32,762,582 | 2,119,388,092 |
| rMat28 | 121,061,651 | 8,515,740,646 |
| rMat30 | 447,070,684 | 34,164,226,400 |
| rMat32 | 1,642,244,247 | 136,975,171,827 |
| rMat34 | 6,004,895,462 | 548,797,112,715 |
| Twitter | 52,579,682 | 3,228,213,000 |
| UK | 131,837,068 | 9,694,060,460 |
| BTC | 303,152,123 | 1,682,860,224 |
| CC2012 | 3,443,082,320 | 227,193,341,978 |
| Yahoo | 272,070,468,294 | 5,922,604,696,718 |

the authors of [42]. We also compare with our own Hadoop MapReduce implementation of AltOpt [36]. Results for these algorithms are presented on Table 3. Execution times are faster than what is reported in [42]. We believe this is due to the fact that they collected their results on a small cluster with 1Gbps interconnect. We observe that Cracker is relatively faster and is the only algorithm able to complete on rMat30 within the allotted 3 hour limit. However, we can already detect their limits. None of these algorithms was able to complete on the larger rMat graphs or on the CC2012, the largest publicly available web graph.

We report results for Algorithm 1 on Table 2. In our case, we limit resources to only 100 workers for graphs with fewer than 100 billion edges, 1000 workers for graphs with up to 1 trillion edges and 5000 workers for the Yahoo web graph. To the best of our knowledge, no other algorithm has been demonstrated to work on graphs of trillions of edges. We report results without using the bucket heuristic and with using a 0.01% bucket size. We complete rMat30 in 178" and rMat34 in 537". rMat34 is the largest graph on which [36] report results. While no absolute execution times are presented in [36], they report that their algorithm completes in a couple hours using several hundreds of cluster nodes (ie thousands of workers.) Our algorithm completes in 341" for CC2012 and in 3808" for the Yahoo web graph.

**Table 2: Results for Algorithm 1**

| graph | P | I | $T_{100\%}$ (sec) | $T_{0.01\%}$ (sec) |
|---|---:|---|---:|---:|
| rMat22 | 100 | 5 | 1.72 | 1.32 |
| rMat24 | 100 | 5 | 6.48 | 4.46 |
| rMat26 | 100 | 5 | 28.26 | 19.66 |
| rMat28 | 100 | 5 | 80.21 | 55.85 |
| rMat30 | 100 | 5 | 328.61 | 177.73 |
| rMat32 | 1000 | 6 | 215.01 | 139.87 |
| rMat34 | 1000 | 6 | 784.55 | 536.95 |
| Twitter | 100 | 6 | 23.48 | 20.6 |
| UK | 100 | 21 | 108.47 | 94.5 |
| BTC | 100 | 14 | 89.79 | 42.7 |
| CC2012 | 1000 | 13 | 475.89 | 341.44 |
| Yahoo | 5000 | 10 | 9112 | 3808 |

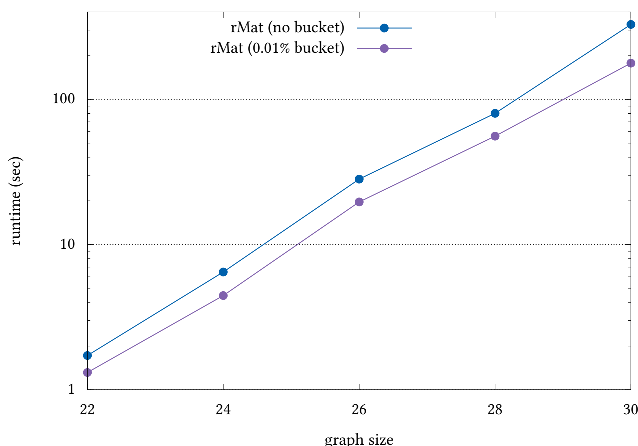



**Figure 3: Log-execution times for rMat22-30**

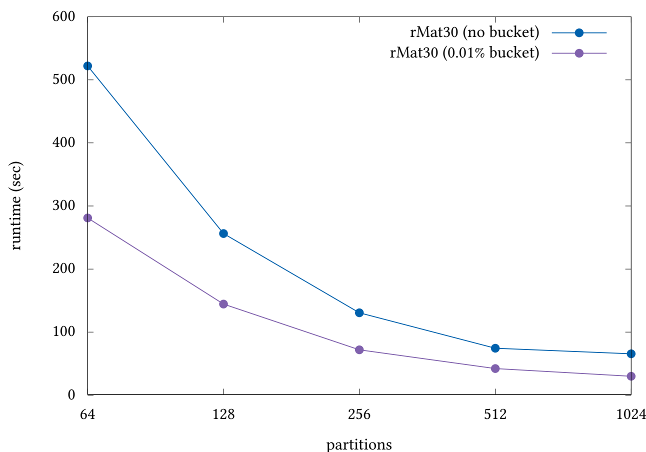



**Figure 4: Runtimes for rMat30 when workers vary from 64 to 1024**

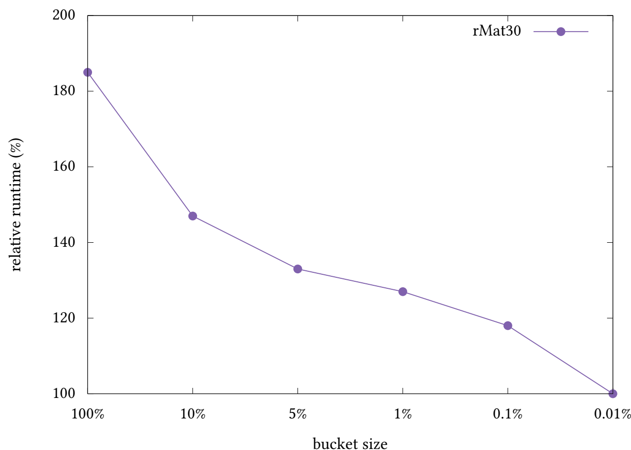



**Figure 5: effect of bucket size on execution times**

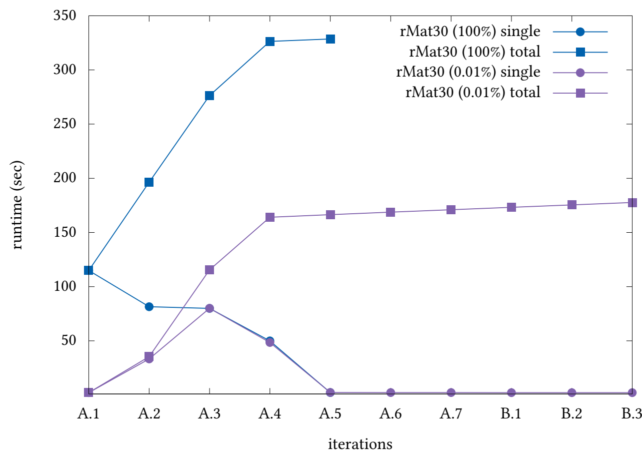



**Figure 6: Individual and aggregate iteration times for rMat30, with and without the bucketing heuristic. The algorithm converges in 5 iterations without bucketing (100% bucket size) while it converges in 10 iterations split into 7 (when bucketing is used) and 3 (for the remaining nodes.)**

We observe how the algorithm scales with the graph size on Figure 3. For a fixed number of 100 workers, we report execution times as the graph size increases and observe near-linear scaling. We also present results for rMat30 by varying the number of workers on Figure 4. On Figure 5, we study the effect of bucket sizing on execution times. The bucket heuristic allows for approximately 2X performance increase. On Figure 6 we present the effect of bucketing on the number of iterations. We present individual iteration times, together with aggregate times both with and without bucketing. The bucketing heuristic generally increases performance by reducing the execution times of the first few iterations, as expected. Note that the total number of iterations increases, but the later ones are typically much faster.

## 5 CONCLUSION

In this work, we presented a simple CC algorithm for BSP that does not mutate the graph, converges in $O(\log n)$ supersteps and scales to graphs of trillions of edges. The algorithm is practically orders of magnitude faster compared to the state of the art and is able to produce results for significantly larger graphs than previously reported in the literature.

## REFERENCES


[1] 2016. Common Crawl. (2016). http://commoncrawl.org/.

[2] 2016. Twitter Graph. (2016). http://konect.uni-koblenz.de/networks/twitter_mpi.

[3] Ajit Agrawal, Lena Nekludova, and Willie Lim. 1987. *A parallel O (log N) algorithm for finding connected components in planar images.* Thinking Machines Corporation.

[4] Baruch Awerbuch and Yossi Shiloach. 1987. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Comput.* 10, C-36 (1987), 1258–1263.

[5] Baruch Awerbuch and Tripurari Singh. 1983. New Connectivity and MSF Algorithms for Ultracomputer and PRAM.. In *ICPP*, Vol. 83. 175–179.

[6] David A Bader and Guojing Cong. 2005. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *J. Parallel and Distrib. Comput.* 65, 9 (2005), 994–1006.

[7] David A Bader, Guojing Cong, and John Feo. 2005. On the architectural requirements for efficient execution of graph algorithms. In *ICPP '05*. IEEE, 547–556.

[8] David A Bader and Joseph JáJá. 1996. Parallel algorithms for image histogramming and connected components with an experimental study. *Journal of parallel and distributed computing* 35, 2 (1996), 173–190.

[9] Dip Sankar Banerjee and Kishore Kothapalli. 2011. Hybrid algorithms for list ranking and graph connected components. In *HiPC (2011)*. IEEE, 1–10.

[10] Guy E Blelloch. 1996. Programming parallel algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

[11] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Graph. *SIGIR Forum* 42, 2 (2008), 33–38.

[12] Robert S Boyer and J Strother Moore. 1991. MJRTY–a fast majority vote algorithm. In *Automated Reasoning*. Springer, 105–117.

[13] Libor Buš and Pavel Tvrdík. 2001. A parallel algorithm for connected components on distributed memory machines. In *European Parallel Virtual Machine / Message Passing Interface Users Group Meeting*. Springer, 280–287.

[14] Edson Norberto Cáceres, Henrique Mongelli, Christiane Nishibe, and Siang Wun Song. 2010. Experimental results of a coarse-grained parallel algorithm for spanning tree and connected components. In *High Performance Computing and Simulation (HPCS), 2010*. IEEE, 631–637.

[15] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[16] Francis Y Chin, John Lam, and I-Ngo Chen. 1982. Efficient parallel algorithms for some graph problems. *Commun. ACM* 25, 9 (1982), 659–665.

[17] Ka Wong Chong and Tak Wah Lam. 1995. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. *J. of Algorithms* 18, 3 (1995), 378–402.

[18] Richard Cole, Philip N Klein, and Robert E Tarjan. 1996. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*. ACM, 243–250.

[19] Richard Cole and Uzi Vishkin. 1991. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation* 92, 1 (1991), 1–47.

[20] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding Frequent Items in Data Streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.

[21] Xing Feng, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Computing Connected Components with linear communication cost in pregel-like systems. In *International Conference on Data Engineering (ICDE)*. IEEE, 85–96.

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

[23] Shay Halperin and Uri Zwick. 1996. An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. System Sci.* 53, 3 (1996), 395–416.

[24] Shay Halperin and Uri Zwick. 2001. Optimal randomized EREW PRAM algorithms for finding spanning forests. *Journal of Algorithms* 39, 1 (2001), 1–46.

[25] Yujie Han and Robert A Wagner. 1990. An efficient and fast parallel-connected component algorithm. *Journal of the ACM (JACM)* 37, 3 (1990), 626–642.

[26] Andreas Harth. 2009. Billion Triples Challenge data set. Downloaded from http://km.aifb.kit.edu/projects/btc-2009/. (2009).

[27] Kenneth A Hawick, Arno Leist, and Daniel P Playne. 2010. Parallel graph component labelling with GPUs and CUDA. *Parallel Comput.* 36, 12 (2010), 655–678.

[28] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. 1979. Computing connected components on parallel computers. *Commun. ACM* 22, 8 (1979), 461–464.

[29] Tsan-Sheng Hsu, Vijaya Ramachandran, and Nathaniel Dean. 1997. Parallel implementation of algorithms for finding connected components in graphs. *Parallel Algorithms: Third DIMACS Implementation Challenge, October 17-19, 1994* 30 (1997), 20.

[30] Kazuo Iwama and Yahiko Kambayashi. 1994. A simpler parallel algorithm for graph connectivity. *Journal of Algorithms* 16, 2 (1994), 190–217.

[31] Joseph JáJá. 1992. *An introduction to parallel algorithms*. Addison-Wesley.

[32] Donald B Johnson and Panagiotis Metaxas. 1997. Connected components in $O(\log^{3/2} n)$ parallel time for the CREW PRAM. *J. Comput. System Sci.* 54, 2 (1997), 227–242.

[33] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. 2009. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM'09*. IEEE, 229–238.

[34] Hakan Kardes, Siddharth Agrawal, Xin Wang, and Ang Sun. 2014. Ccf: Fast and scalable connected component computation in mapreduce. In *ICNC (2014)*. IEEE, 994–998.

[35] David R Karger, Noam Nisan, and Michal Parnas. 1992. Fast connected components algorithms for the EREW PRAM. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. ACM, 373–381.

**Table 3: Experimental Results for Prior Art Algorithms**

| Graph | Workers | Running Time of various Algorithms | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | AltOpt | | CCF | | Cracker | | HashToMin | | Pegasus | | AltOpt$_{MR}$ | |
| | | Time | Iter.* | Time | Iter. | Time | Iter. | Time | Iter. | Time | Iter. | Time | Iter.* |
| rMat22 | 10 | 678 | 4 | 278 | 5 | 348 | 4 | 1481 | 5 | 1887 | 5 | 454 | 4 |
| | 100 | 295 | 4 | 62 | 5 | 88 | 4 | 410 | 5 | 204 | 5 | 309 | 4 |
| | 256 | 249 | 4 | 88 | 5 | 92 | 4 | 572 | 5 | 300 | 5 | 671 | 4 |
| | 512 | 532 | 4 | 377 | 5 | 345 | 4 | 837 | 5 | 614 | 5 | 557 | 4 |
| rMat24 | 10 | 1821 | 4 | 984 | 5 | 418 | 5 | x | | x | | 1670 | 4 |
| | 100 | 1361 | 4 | 225 | 5 | 200 | 5 | 765 | 5 | 888 | 5 | 1576 | 4 |
| | 256 | 794 | 4 | 206 | 5 | 307 | 5 | 1349 | 5 | 523 | 5 | 984 | 4 |
| | 512 | 980 | 4 | 169 | 5 | 431 | 5 | 1684 | 5 | 260 | 5 | 2014 | 4 |
| | 1024 | 1244 | 4 | 532 | 5 | 329 | 5 | 2063 | 5 | 327 | 5 | 1991 | 4 |
| rMat26 | 100 | 4311 | 4 | 786 | 6 | 568 | 5 | x | | 2978 | 6 | 7823 | 4 |
| | 256 | 3117 | 4 | 537 | 6 | 351 | 5 | x | | 1278 | 6 | 5830 | 4 |
| | 512 | 3713 | 4 | 526 | 6 | 350 | 5 | 7231 | 6 | 1519 | 6 | 3901 | 4 |
| | 1024 | 4082 | 4 | 793 | 6 | 333 | 5 | 6865 | 6 | 393 | 6 | 5547 | 4 |
| rMat28 | 512 | x | | 4198 | 6 | 1487 | 5 | x | | x | | x | |
| | 1024 | x | | 2297 | 6 | 1255 | 5 | x | | 1734 | 6 | x | |
| | 2048 | x | | 2311 | 6 | 1069 | 5 | x | | 1176 | 6 | x | |
| rMat30 | 2048 | x | | x | | 9915 | 6 | x | | x | | x | |
| twitter | 1024 | 10295 | 5 | 1195 | 7 | 487 | 9 | 9548 | 7 | 2799 | 13 | 5169 | 5 |
| | 2048 | 7544 | 5 | 2510 | 7 | 1178 | 9 | 12710 | 7 | 5593 | 13 | 4442 | 5 |
| uk | 1024 | x | | 9977 | 20 | 1087 | 19 | x | | x | | x | |
| | 2048 | x | | 9048 | 20 | 8252 | 19 | x | | x | | x | |
| btc | 100 | x | | x | | 1253 | 12 | x | | x | | x | |
| | 512 | x | | x | | 3011 | 12 | x | | x | | x | |
| | 1024 | x | | x | | x | | x | | x | | x | |
| cc2012 | 2048 | x | | x | | x | | x | | x | | x | |

x – Experiment resulted in timeout (3 hours) or insufficient memory error

* – Large-Star and Small-Star are counted as one combined iteration [36]


[36] Raimondas Kiveris, Silvio Lattanzi, Vahab Mirrokni, Vibhor Rastogi, and Sergei Vassilvitskii. 2014. Connected components in mapreduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–13.

[37] Václav Koubek and Jana Kršňáková. 1985. Parallel algorithms for connected components in a graph. In *Fund. of Computation Theory*. Springer, 208–217.

[38] Arvind Krishnamurthy, Steven Lumetta, David E Culler, and Katherine Yelick. 1997. Connected components on distributed memory machines. *Third DIMACS Implementation Challenge* 30 (1997), 1–21.

[39] Clyde P Kruskal, Larry Rudolph, and Marc Snir. 1990. Efficient parallel algorithms for graph problems. *Algorithmica* 5, 1 (1990), 43–64.

[40] Subodh Kumar, Stephen M Goddard, and Jan F Prins. 1995. Connected-Components Algorithms For Mesh-Connected Parallel Computers. In *3rd Dimacs Implementation Challenge Workshop*.

[41] Willie Lim, Ajit Agrawal, and Lena Nekludova. 1986. *A fast parallel algorithm for labeling connected components in image arrays*. Thinking Machines Corporation.

[42] Alessandro Lulli, Emanuele Carlini, Patrizio Dazzi, Claudio Lucchese, and Laura Ricci. 2017. Fast connected components computation in large graphs by vertex pruning. *IEEE Transactions on Parallel and Distributed systems* 28, 3 (2017), 760–773.

[43] Dhruva Nath and SN Maheshwari. 1982. Parallel algorithms for the connected components and minimal spanning tree problems. *Inform. Process. Lett.* 14, 1 (1982), 7–11.

[44] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 456–471.

[45] Noam Nisan, Endre Szemeredi, and Avi Wigderson. 1992. Undirected connectivity in $O(\log^{1.5} n)$ space. In *Foundations of Computer Science, 1992*. IEEE, 24–29.

[46] Ha-Myung Park, Namyong Park, Sung-Hyon Myaeng, and U Kang. 2016. Partition aware connected component computation in distributed systems. In *International Conference on Data Mining (2016)*. IEEE, 420–429.

[47] Md Mostofa Ali Patwary, Peder Refsnes, and Fredrik Manne. 2012. Multi-core spanning forest algorithms using the disjoint-set data structure. In *International Parallel & Distributed Processing Symposium (2012)*. IEEE, 827–835.

[48] Seth Pettie and Vijaya Ramachandran. 2002. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.* 31, 6 (2002), 1879–1895.

[49] CA Philips. 1989. Parallel graph contraction. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*. ACM, 148–157.

[50] Chung Keung Poon and Vijaya Ramachandran. 1997. A randomized linear work EREW PRAM algorithm to find a minimum spanning forest. In *International Symposium on Algorithms and Computation*. Springer, 212–222.

[51] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. 2013. Finding connected components in map-reduce in logarithmic rounds. In *International Conference on Data Engineering (2013)*. IEEE, 50–61.

[52] Yossi Shiloach and Uzi Vishkin. 1982. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms* 3, 1 (1982), 57–67.

[53] Julian Shun, Laxman Dhulipala, and Guy Blelloch. 2014. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM, 143–153.

[54] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *International Parallel and Distributed Processing Symposium (2014)*. IEEE, 550–559.

[55] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW) (2010)*. IEEE, 1–8.

[56] Stergios Stergiou, Zygimantas Straznickas, Rolina Wu, and Kostas Tsioutsiouliklis. 2017. Distributed Negative Sampling for Word Embeddings. In *AAAI Conference on Artificial Intelligence (2017)*. 2569–2575.