Graph connectivity in log-diameter steps using label propagation

Paul Burkhardt*

July 20, 2018

Abstract

The fastest deterministic algorithms for connected components take logarithmic time and perform superlinear work on a Parallel Random Access Machine (PRAM). These algorithms maintain a spanning forest by merging and compressing trees, which requires pointer-chasing operations that increase memory access latency and are limited to shared-memory systems. Many of these PRAM algorithms are also very complicated to implement. Another popular method is "leader-contraction" where non-leader vertices are contracted to adjacent leaders. The challenge is to select a constant fraction of leaders that are adjacent to a constant fraction of non-leaders with high probability. Instead we investigate whether simple label propagation can be as efficient as the fastest known algorithms for graph connectivity. Label propagation exchanges representative labels within a component. This is attractive for other models because it is deterministic and does not rely on pointer-chasing, but it is inherently difficult to complete in a sublinear number of steps. We are able to solve the problems with label propagation for graph connectivity.

We introduce a simple framework for deterministic graph connectivity in log-diameter steps using label propagation that is easily translated to other computational models. We present new algorithms in PRAM, Stream, and MapReduce. Given a simple, undirected graph $G = (V, E)$ with $n = |V|$ vertices, $m = |E|$ edges, and D diameter, all our algorithms complete in $O(\log D)$ steps without pointer operations. We give the first label propagation algorithms that are competitive with the fastest PRAM algorithms, achieving $O(\log D)$ time and $O((m + n) \log D)$ work with $O(m + n)$ processors. Our main contribution is in Stream and MapReduce models. We give an efficient Stream-Sort algorithm that takes $O(\log D)$ passes and $O(\log n)$ memory, and a MapReduce algorithm taking $O(\log D)$ rounds and $O((m+n) \log D)$ communication overall. These are the first $O(\log D)$-step graph connectivity algorithms in Stream and MapReduce models that are also deterministic and simple to implement.

Keywords: graph connectivity, connected components, parallel algorithm, pram, stream, mapreduce

1 Introduction

Given a simple, undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, the connected components of G are partitions of V such that every pair of vertices in a connected component are endpoints of a path, which is a sequence of vertices $v_i, v_{i+1}, \ldots$ where the v_i vertex is adjacent to the v_{i+1} vertex. If two vertices are not connected by a path then they are necessarily in different components. The distance is the length of the shortest-path between two vertices, and the diameter D is the maximum distance in G. In this paper we consider parallel algorithms for finding connected components of G.

*Research Directorate, National Security Agency, Ft. Meade, MD 20755. Email: pburkha@nsa.gov

The work of a parallel algorithm is the product of the runtime with the number of processors. A parallel algorithm is considered work-optimal if the work matches that of a sequential algorithm. Connected components can be identified in linear time by a sequential algorithm using Breadth-First or Depth-First Search [45]. But there is no known work-optimal, deterministic parallel ($\mathcal{NC}$) algorithm for connected components. The first $\mathcal{NC}$ algorithm was given by Hirschberg, Chandra, and Sarwate [24] in 1979 and required $O(n^2 \log n)$ work. This was improved later by Chin, Lam, and Chen [9] to use fewer processors and overall $O(n^2)$ work. In 1982 Shiloach and Vishkin gave a $O(\log n)$ time, $O((m+n)\log n)$ work algorithm [43] and since then there have been near-optimal, up to a polylog factor in work, algorithms [3, 22, 13, 11, 25]. Randomized, work-optimal parallel algorithms exist but achieve linear-work and polylogarithmic or logarithmic time with high probability [18, 21, 44].

The fastest deterministic algorithms for connected components take logarithmic time and perform superlinear work on a Concurrent Read Concurrent Write (CRCW) Parallel Random Access Machine(PRAM). These algorithms maintain a spanning forest by merging and compressing trees [43, 3], which requires pointer-chasing operations that increase memory access latency. Pointer jumping was shown to be one of the primary sources of slowdown in a parallel minimum spanning tree algorithm [12]. Moreover the PRAM algorithm implementations are limited to globally-shared memory systems [19, 35, 4]. Another popular method is "leader-contraction", used in [2, 37, 29], where a fraction of vertices are randomly selected as leaders and then adjacent non-leader vertices are contracted to the leader. But to achieve logarithmic convergence a constant fraction of non-leader vertices must be adjacent to leader vertices in each step. The leaders themselves must also be contracted to other leaders so the challenge is to select a constant fraction of leaders that are adjacent to a constant fraction of non-leaders with high probability. Not only is this method randomized but it can require adding many more edges to the graph leading to high communication cost. In [2] the authors add edges so vertices have large degree and therefore have high probability of being adjacent to a smaller fraction of leader vertices. To avoid $\Omega(n^3)$ communication cost to increase the degrees, they carefully manage how edges are created. Our goal is to find a simple method that contracts the graph in a similar manner to achieve logarithmic convergence while also keeping the edge count, and therefore the work, constant each step.

We investigate whether label propagation can be as efficient as the fastest known algorithms for graph connectivity. In label propagation algorithms a representative label, such as the minimum vertex ID, is passed to neighboring vertices. This is attractive for other models because it is deterministic and does not rely on pointer-chasing. But it is inherently difficult to complete in a sublinear number of steps. The method was popular for image processing [38, 40, 42] where it uses simple graph traversal. These algorithms were often sequential and do not guarantee logarithmic convergence. Logarithmic convergence can be achieved if paths can be contracted with each label update so that the overall diameter decreases by a constant factor each step. To keep the edge count and therefore the work constant in each step, edges must be removed or replaced in like number. But this is still insufficient. Consider for example a path that is sequentially labeled from left to right, i.e. a chain, with one processor for each vertex. Each vertex gains an edge to the vertex that is a distance of two to its left, since that is the minimum label of the left neighbor. Edges to the left and right neighbors are removed. This connects all even-numbered and odd-numbered vertices separately, but bisects the graph in just the first step.

We are able to solve the problems with label propagation for graph connectivity. Our solution to obtaining log-diameter convergence and near optimal work is surprisingly simple. In this paper we say that a (v, u) edge is directed from v to u and $(\cdot, \cdot)$ denotes an ordered pair that distinguishes (v, u) from (u, v). These counter-oriented $(v, u), (u, v)$ edges are twins of a conjugate pair. We wish to create just one twin for each edge otherwise there can be many duplicates that increase

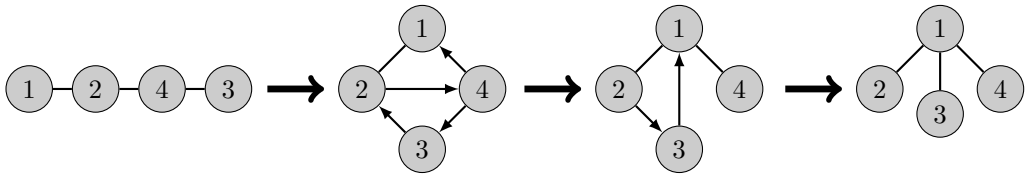

Figure 1: A path graph converges into a star after three steps. After each step the output graph becomes the input for the next step. Undirected edges denote a pair of oppositely oriented edges.

the edge count. We propagate labels as previously described, replacing each current edge with a new edge every step to keep the edge count from increasing. But also in each step we keep the minimum label edge that was propagated in the previous step, but in the reverse direction. Meaning if we had a $(v, l(v))$ edge in the previous step, where $l(v)$ was the minimum label for v, then we'll replace it with $(l(v), v)$ for the next step. We'll call these two edge operations *label propagation* and *symmetrization*, respectively. Thus we replace (v, u) edges with $(l(v), v)$ and $(u, l(v))$ edges.

Notice that there are four combinations for edge direction between label propagation and symmetrization. In our method the direction with respect to $l(v)$ is counter-oriented. These $l(v)$ are analogous to leaders in the leader-contraction method. If $l(v)$ were mutually-oriented the algorithm would disconnect the graph or not converge because eventually the same edges will be returned each step. The other counter-oriented choice would also fail because it is necessary for a vertex to receive new labels from previous minimum labels. In our method an edge between v and an $l(v)$ will rotate in the next step, and then be replaced as v receives a new minimum label. This rotation maintains a stable edge count because the same edge between v and $l(v)$ cannot be created repeatedly and yet rotating from $(v, l(v))$ to $(l(v), v)$ allows v to receive a new label from $l(v)$. We invite the reader to try the different combinations on a chain and then refer to Figure 1 for an example of how our method works.

This is a powerful new technique for graph connectivity algorithms because it is deterministic and doesn't require pointer-jumping. We'll describe in more detail how our method leads to parallel connected component algorithms that are competitive with the fastest PRAM algorithms. But our primary goal is to apply it in computational models where pointer operations, random access to global data, and memory are restricted. This includes the Stream model which in recent years has gained attention for processing large graphs with limited memory [15, 10, 32]. The MapReduce model has also been popular and shown to enable Petabyte-scale graph traversal [7] without pointer operations. But until this paper there have not been either Stream or MapReduce algorithms for graph connectivity that are deterministic and complete in $O(\log D)$ steps. Recently a $O(\log D \log\log_{m/n} n)$ round algorithm was given in the *MPC* model [2], which is a generalization of MapReduce, and that MapReduce algorithms can be simulated in *MPC* with the same runtime. But this *MPC* algorithm uses the leader-contraction method and is therefore randomized and only achieves $O(\log D)$ steps using more space than the input graph. In contrast, we give a deterministic algorithm in the more restrictive MapReduce model that takes $O(\log D)$ steps using linear space. Moreover, our algorithm is far simpler to implement.

2 Our contribution

We introduce a simple framework for deterministic graph connectivity in log-diameter steps using label propagation that is easily translated to other computational models. It produces the same result as the well-known PRAM algorithms, that is a forest of stars each rooted at a component

Model	Steps	Cost	Class	Author
CRCW	$O(\log n)$	$O((m+n)\log n)$	deterministic	Shiloach, Vishkin [43]
Stream-Sort	$O(\log n)$	$O(\log n)$	randomized	Aggarwal et al. [1]
MapReduce	$O(\log^2 n)$	$O(m \log^2 n)$	deterministic	Kiveras et al. [29]
CRCW	$O(\log D)$	$O((m+n)\log D)$	deterministic	This paper
Stream-Sort	$O(\log D)$	$O(\log n)$	deterministic	This paper
MapReduce	$O(\log D)$	$O((m+n)\log D)$	deterministic	This paper

Table 1: Comparison to state-of-the-art. Cost is work, memory, or communication for PRAM, Stream-Sort, and MapReduce, respectively.

label. Our framework creates a new graph each step without increasing the number of edges. This avoids explicit edge deletion and load-balancing operations.

We present new algorithms in PRAM, Stream, and MapReduce. All our algorithms complete in $O(\log D)$ steps without pointer operations. We give the first label propagation algorithms that are competitive with the fastest PRAM algorithms, achieving $O(\log D)$ time and $O((m+n)\log D)$ work with $O(m+n)$ processors. Our main contribution is in Stream and MapReduce models. We give an efficient Stream-Sort algorithm that takes $O(\log D)$ passes and $O(\log n)$ memory, and a MapReduce algorithm taking $O(\log D)$ rounds and $O((m+n)\log D)$ communication overall. These are the first $O(\log D)$-step graph connectivity algorithms in Stream and MapReduce models that are also deterministic and simple to implement. Refer to Table 1 for a summary of these results and comparison to the current state-of-the-art.

We begin with some background information in Section 3 on the computational models we explored for our algorithms. This is followed by a survey of related work in Section 4. Then in Sections 5 and 6 we introduce the framework behind our labeling technique with our PRAM algorithm described in Section 7. In Section 8 the framework is extended for Stream-Sort and MapReduce models, which introduces the subject of label duplication in Section 9 where we identify how pathological duplication of labels arises and how to address it. Finally we describe our Stream-Sort and MapReduce algorithms in the remaining Sections 10 and 11.

3 Computational models

The computational models for our algorithms are PRAM, Stream, and MapReduce. These models define the constraints under which an algorithm operates. We refer the reader to [8, 34, 31, 30] for surveys on these models of computation.

In a PRAM [17] each processor can access any global memory location in unit time. Processors can read from global memory, perform a computation, and write a result to global memory in a single clock cycle. All processors execute these instructions at the same time. Simultaneous read or write to a single memory location is managed by extensions to the model. In the Exclusive Read Exclusive Write (EREW) PRAM concurrent read and write is restricted to distinct memory locations. The Concurrent Read Exclusive Write (CREW) PRAM restricts writing to a single memory location by one processor at a time. Finally the Concurrent Read Concurrent Write (CRCW) PRAM permits any location in global memory to be simultaneously read or written by any number of processors. Write conflicts are still possible in the CRCW PRAM if different values are being concurrently written by multiple processors to the same memory location. Such write conflicts can be resolved by different protocols. A Combining Write Resolution uses an associative operator to combine all values in a single instruction. A Combining CRCW employs this to store a reduction of the values, such as the minimum, in constant time [41].

The Stream model [33, 23] focuses on the trade-off between working memory space s and number of passes p over the input stream, allowing the computational time to be unbounded. It was motivated by massive datasets where the input data is presented as a sequential stream of elements to be processed in order and the available memory for computation is much less than the total input. In W-Stream [39] an algorithm can write to the stream for subsequent passes and in Stream-Sort [1] the input or intermediate output stream can also be sorted. In both W-Stream and Stream-Sort the output streams become the input stream in the next pass. In Stream-Sort the streaming and sorting passes alternate so a Stream-Sort algorithm reads an input stream, computing on the items in the stream, while writing to an intermediate output stream that gets reordered for free by a subsequent sorting pass. An algorithm in Stream-Sort is efficient if it takes polylog n passes and memory.

The MapReduce model [27, 20, 36] appeared some years after the programming paradigm was popularized by Google [14]. The MapReduce computational model employs two functions *map* and *reduce*, executed in sequence. The input is a set of $\langle key, value\rangle$ pairs that are "mapped" by instances of the *map* function into a multiset of $\langle key, value\rangle$ pairs. Each *map* instance reads and writes a single pair at a time and does not maintain state between successive pairs. The *map* output pairs are "reduced" and also output as a multiset of $\langle key, value\rangle$ pairs by instances of the *reduce* function where a single *reduce* instance gets all values associated with a key. A *round* of computation is a single sequence of *map* and *reduce* executions where there can be many instances of *map* and *reduce* functions. An iterative algorithm is then a sequence of such *rounds*. In each round the *reduce* instances cannot complete until all *map* instances have completed, so computation is both stateless and synchronous. Each *map* or *reduce* function can complete in polynomial time for input n. Each *map* or *reduce* instance is limited to $O(n^{1-\epsilon})$ memory for a constant $\epsilon > 0$, and an algorithm is allowed $O(n^{2-2\epsilon})$ total memory. An algorithm can execute any number of instances of *map* and *reduce* functions within the memory limit. The number of machines/processors is bounded to $O(n^{1-\epsilon})$, but each machine can run more than one instance of a *map* or *reduce* function.

4 Related work

The famous 1982 algorithm by Shiloach and Vishkin [43] takes $O(\log n)$ time using $O(m+n)$ processors on a CRCW PRAM, performing $O((m+n)\log n)$ work overall. The Shiloach-Vishkin algorithm maintains a forest of trees that are merged and shortened in each iteration. It is difficult to translate to other computational models because of the pointer operations. Our CRCW algorithm achieves the same bounds for $D = O(n)$ without pointer operations. It is more advantageous when $D \leq O(\log n)$ and does not require reading over all edges twice in each iteration step as done in Shiloach-Vishkin, which is a significant savings in practice for very large graphs.

The best known deterministic Stream algorithm for connected components is given by Demetrescu, et al. [16], taking $O((n\log n)/s)$ passes and s working memory size in W-Stream. Their algorithm permits flexible tradeoff between space and passes but can only achieve $O(\log n)$ passes using $s = O(n)$ memory. Moreover it requires four steps per pass and a pre-existing, deterministic connected component algorithm, which it applies to a subset of the edges in memory at each pass. For $s \leq O(\log n)$ the algorithm in [16] would run in $\Omega(n)$ steps. This can be improved using sorting in the Stream-Sort model. A randomized s-t-connectivity algorithm by Aggarwal, et al. [1] takes $O(\log n)$ passes and memory in Stream-Sort. Their algorithm can be modified to compute connected components with the same bounds [34], but requires sorting in three of four steps in each pass. In contrast we'll present the first efficient, deterministic Stream-Sort con-

nected components algorithm taking $O(\log D)$ passes and $O(\log n)$ memory. Moreover it requires only a single sorting step per pass, always completes in $O(\log D)$ passes, and is straightforward to implement.

A randomized MapReduce algorithm by Rastogi et al. [37] was the first logarithmic-round MapReduce algorithm for connected components that appeared in the literature[1]. But their algorithm communicates an entire connected component from one task to another at each step and it must store all connected components in the memory of a single machine, both of which will be problematic for very large graphs. They addressed this latter issue with their *Hash-to-Min* algorithm [37] but are unable to prove its performance, showing only that it empirically takes $2 \log D$ rounds and $O(m + n)$ communication per round. It was later shown in [2] that in fact it takes $\Omega(\log n)$ rounds. It also uses a single task to send an entire component to another, which for a giant component will effectively serialize the communication. In 2014 Kiveris et al. [29] introduce their *Two-Phase* algorithm, which takes $O(\log^2 n)$ rounds to converge using $O(m)$ communications per round leading to $O(m \log^2 n)$ overall communication cost. Unlike the *Hash-to-Min* algorithm it avoids sending the giant component to a single reduce task and has provable complexity bounds. The authors in [29] improve the *Two-Phase* time to $O(\log n \log \log n)$ rounds by replacing the inner loop with a lookup into a Distributed Hash Table (DHT), but this is really beyond the realm of the MapReduce framework. Until this paper, the *Two-Phase* algorithm represented the state-of-the-art connected components algorithm in MapReduce.

We introduce a new MapReduce algorithm that has better runtime than *Two-Phase* while being deterministic and simple to implement. Like *Two-Phase* our algorithm does not load components into memory or send an entire component through a single communication channel. We go further in memory conservation by maintaining $O(1)$-space working memory. Our MapReduce algorithm completes in $O(\log D)$ rounds using $O((m + n) \log D)$ overall communication thereby improving the state-of-the-art by a factor of $O(\log n)$ in both convergence and communication in the worst case. To the best of our knowledge this is the fastest and most efficient deterministic MapReduce algorithm for connected components.

Although we do not study the *MPC* model in this paper, we want to highlight the recent work by Andoni, et al. [2] who give a randomized $O(\log D \log \log_{m/n} n)$ step algorithm in *MPC* for connected components. Their algorithm uses the leader-contraction method but they accomplish faster than $O(\log n)$ convergence by selecting a smaller fraction of leader vertices while maintaining high probability that non-leader vertices are adjacent. To achieve this the authors add edges so the graph has uniformly large degree but to avoid $\Omega(n^3)$ communication cost they carefully manage how edges are added. Their algorithm is very complicated and only achieves $O(\log D)$ steps by using more space than the input graph. Since the *MPC* model is more powerful than MapReduce, and can therefore simulate a MapReduce algorithm without slowdown, it implies that our MapReduce algorithm in the *MPC* model is more efficient. Our MapReduce algorithm achieves $O(\log D)$ steps with linear space while being very simple.

5 Framework

Our framework employs two edge creation operations, *label propagation* and *symmetrization*. The label propagation operation adds an edge between the minimum label of a vertex and each of its neighbors. The symmetrization operation keeps only the minimum label edge created by label propagation in the previous step. The label propagation operation will contract paths while the symmetrization ultimately retains edges in the final rooted star even if the corresponding

[1] We had developed a deterministic, logarithmic-round MapReduce algorithm internally in 2013, later presented in [6].

vertex isn't involved in the label propagation step for many iterations. Previous edges are either replaced or ignored by these operations each step.

In G an undirected edge is comprised of a conjugate pair, thus there are $2m$ edges in total and G is symmetric. As stated in the introduction, we are careful to create just one twin of a conjugate pair. Our edge operations replace (v, u) edges with $(l(v), v)$ and $(u, l(v))$ edges each step without the counter-oriented twins. Thus at each step the graph may be directed, but a single (v, u) edge indicates that a path exists between v and u regardless of the direction of the edge. A consequence of our method is that direction of the edge $(v, l(v))$ rotates before being replaced. This will maintain the connectivity of the graph and keep the edge count from growing.

Let's say a (v, u) edge was created by label propagation and there was no prior (u, v) twin. Also $u = l(v)$ and u gets a new minimum. Now symmetrization will rotate this edge to the counter-oriented twin, (u, v), thus preserving the connectivity between u and v. Since (v, u) is not created then (u, v) cannot be redundantly created again. Moreover v, u will no longer be connected by an edge because the (u, v) edge is replaced with $(v, l(u))$. Not only does this keep the edge count from progressively increasing, it ensures that edges that were removed will not reappear. If u is a local minimum, or ultimately the representative label for the component, then the conjugate twins $(u, v), (v, u)$ are retained each step.

There is another important result to this edge rotation. An $l(v)$ at step k can give its latest minimum label to v at step $k + 2$. This implies logarithmic convergence because every two steps v will get a new label that is a constant factor closer to the representative label. For example, $l(v)$ at step k is two steps from v. At step $k + 1$ this $l(v)$ receives a new minimum that is two steps away from it. The new minimum for $l(v)$ is then passed to v at step $k + 2$, hence v receives a label that is a distance of at least four from it. Simultaneously, any new $l(v)$ that v receives will also get a minimum that was a distance of at least four from it and can pass its new minimum to v. When v gets the representative label the edge to it will be maintained for each step forward because the conjugate pair will be completed by label propagation and symmetrization. We'll give a formal analysis of the convergence in the next section.

6 Principle algorithm

We begin with Algorithm 1 to establish the core algorithmic principles from which we then derive our other algorithms. We don't specify any model now so we can focus on the basic procedures. We will use $N_k(v)$ to denote the neighborhood of a vertex v at step k and $N_k^+(v) = \{v\} \cup N_k(v)$ as the inclusive neighborhood. Then let $l(v) = \min(N_k^+(v))$ be the current minimum label for v. We use this $l(v)$ notation without a step subscript for simplicity. In our algorithm listings we use arrays L_k in its place, e.g. $L_k[v]$ holds $l(v)$ for v at step k. Only two such arrays are needed in each step. Before the algorithm starts L_1 is initialized with the $l(v)$. For all algorithms we use E_k to denote the edges that will be processed at step k, but E_k is a multiset because it may contain duplicates.

In Algorithm 1, label propagation and symmetrization are performed by the first and second **for all** loops, respectively. At each step the label propagation operation replaces (v, u) edges with $(u, l(v))$ if u is not the minimum label of v. The symmetrization operation creates an $(l(v), v)$ edge for v whose minimum label is not v. This is repeated until labels converge. After each step the previous edges are discarded so the algorithm does not continuously augment a previous set. The final result is a forest of stars with each star representing a connected component and is rooted at the minimum label for that component. The algorithm is illustrated in Figure 1 on a simple path graph converging to a star in three steps, where undirected edges denote a pair of oppositely oriented edges.

Algorithm 1

L_k ▷ arrays for $l(v)$ at each step k

Initialize L_1 with all starting $l(v)$

1: **for** $k = 1, 2, \ldots$ until labels converge **do**
2: set $L_{k+1} := L_k$
3: **for all** $(v, u) \in E_k$ **do**
4: **if** $L_k[v] \neq u$ **then**
5: add $(u, L_k[v])$ to E_{k+1} and set $L_{k+1}[u] := \min(L_{k+1}[u], L_k[v])$
6: **for all** $v \in V$ **do**
7: **if** $L_k[v] \neq v$ **then**
8: add $(L_k[v], v)$ to E_{k+1}

Lemma 1. *All connected components in G are correctly identified by Algorithm 1.*

Proof. At any step a (v, u) edge will become (u, v) by symmetrization if $u = l(v)$. Otherwise it is replaced with $(u, l(v))$ by label propagation and $(l(v), v)$ is created by symmetrization.

For any $(v, u), (u, v)$ edges starting in G, a path is retained between v and u after the first step as follows.

If $u = l(v)$ and $u = l(u)$ then $(u, v), (v, u)$ edges are created.

If $u = l(v)$ and $u \neq l(u)$ then $(u, v), (v, l(u)), (l(u), u)$ are created.

If $u \neq l(v)$ and $u = l(u)$, then $(u, l(v)), (l(v), v), (v, u)$ are created.

If $u \neq l(v)$ and $u \neq l(u)$ then $(u, l(v)), (l(v), v), (v, l(u)), (l(u), u)$ are created.

Then by induction, for any (v, u) edge created in some step a path will exist between v and u in all subsequent steps. Thus all vertices in a component remain connected. Now let u be the minimum label of a component. Then any new (v, u) edge gained by label propagation will be retained and symmetrized. Therefore each component will transform to a star rooted at the representative minimum label. □

Lemma 2. *For every value of k, $|E_k| \leq 2m + n$ in Algorithm 1.*

Proof. After each step the current edges are replaced with new edges as follows.

If $l(v) \neq u$ then (v, u) is replaced with $(u, l(v))$ by label propagation.

If $l(v) = u$ then (v, u) is replaced with (u, v) by symmetrization.

Observe there is no increase in edge count.

But if $l(v) \neq v$ and $l(v) \neq u$ then $(l(v), v)$ is added by symmetrization.

Hence if v did not have an explicit edge to its minimum label then symmetrization adds one more edge than the number of its current neighbors. There are at most n such vertices so the maximum edge count is $2m + n$ in this step. But in this same step label propagation from $l(v)$ adds the $(v, l(v))$ edge so in the next step an extra edge cannot be created from such a v. Therefore at any step k, $|E_k| \leq 2m + n$ edges. □

Lemma 3. *Algorithm 1 converges in $O(\log D)$ steps.*

Proof. Label propagation induces at least one length-2 path contraction in any arbitrary permutation of a 4-path because a 4-path is not a star. After label propagation the maximum distance from a vertex to the minimum label is two, forming a 3-path. It takes at most two steps for an arbitrary 3-path to converge to a star rooted at the minimum label as follows. Label propagation creates an edge between the two leaf nodes, one of which must be the minimum label. Note that a cycle has been formed. In the next step the cycle is broken because two of the nodes now share the minimum label as a neighbor and hence the 3-path is now rooted at the minimum label. The

4-path is now a star and one final step verifies no change in labels. Therefore it takes at most four steps to converge a 4-path to a star rooted at the minimum label.

Observe that a path can be constructed by concatenating a 4-path with another 4-path. Each 4-path in this new 8-path will simultaneously reach a minimum label by four steps and at most two extra steps are then required to merge the two intermediate star graphs. Doubling again will require six steps as before and then another two steps to merge the pair of forming star graphs. By induction, k successive doubling from a 4-path results in a path of size $r = 4 \cdot 2^k$ and requires $2k$ additional steps for merging the intermediate star graphs. Therefore any path of size $r > 4$ must converge in as many steps as it takes for an arbitrary path of size $2^{\lfloor \log r \rfloor + 1}$ constructed from $k = \lfloor \log r \rfloor - 1$ successive doubling from a 4-path. Thus convergence for any r-path takes at most $4 + 2k = 4 + 2(\lfloor \log r \rfloor - 1) \leq 2 + 2 \log r = O(\log r)$ steps.

Now since any arbitrary path in G is contracted to a star in a logarithmic number of steps, then the diameter of G also decreases logarithmically. If G were already a star rooted at some v that is not the minimum label, it would only take two steps to exchange that v with the minimum label because in the first step label propagation gives the minimum label to all other vertices and the next step finalizes the star. Therefore it takes $O(\log D)$ steps to converge. □

We've established the basic concepts of our framework in Algorithm 1. We'll show how to extend and apply it in Stream-Sort and MapReduce models, but first we'll describe it on a PRAM since it is directly applicable and it may be of interest for small D.

7 PRAM algorithm

Our Algorithm 1 can run on a PRAM as described but to more clearly demonstrate the edge operations we make it edge-centric in our next algorithms. We will use the following semantics for our parallel algorithms. All statements are executed sequentially from top to bottom but all operations contained within a **for all** construct are performed concurrently. All other statements outside this construct are sequential. Hence there is an implicit barrier between **for all** constructs. Recall that in a synchronous PRAM all processors perform instructions simultaneously and each instruction takes unit time.

We begin now with Algorithm 2 which runs on a Combining CRCW PRAM to ensure the correct minimum label is written in $O(1)$ time [41]. Algorithm 2 is nearly identical to Algorithm 1 except the label propagation and symmetrization stages are combined into a single pass over the edges. In Algorithm 1 the symmetrization stage is a separate pass over the vertices and so a single processor can be assigned to each vertex, thus only a single $(L_k[v], v)$ can be written. To ensure this same result in Algorithm 2, where one processor is assigned to each edge, the algorithm uses a simple array W of size $O(n)$ so that each processor working on its (v, u) edge would write its processor ID to W indexed at v. Any write conflict is resolved using Combining Write Resolution so the processor that wins will execute the symmetrization operation.

Lemma 4. *Algorithm 2 on a PRAM does not require load-balancing at any step.*

Proof. Recall from Lemma 2 that the number of edges is $|E_k| \leq 2m + n$ for all k. Initially assign each of $2m + n$ processors to a memory cell in the PRAM, then load the $2m$ edges of G into the same memory cells leaving n cells empty. Every (v, u) edge is replaced but an additional edge can be created by symmetrization when v does not have an explicit edge to its minimum label. For such a v the extra edge is written to the location indexed at $2m + v$. Thus at every step each processor updates its cell with a new edge, and if a processor's memory cell is empty or if the contents are stale, i.e. not updated in the last step, then that processor performs no operation. The initialization and reading from memory cells takes constant time. □

Algorithm 2

```
   L_k                                              ▷ arrays for l(v) at each step k
   W                                                ▷ array of size n
   p                                                ▷ processor id
   Initialize L_1 with all starting l(v)
1: for k = 1, 2, ... until labels converge do
2:     set L_{k+1} := L_k
3:     for all (v, u) ∈ E_k do
4:         if L_k[v] ≠ u then
5:             add (u, L_k[v]) to E_{k+1} and set L_{k+1}[u] := min(L_{k+1}[u], L_k[v])
6:         set W[v] := min(W[v], p)              ▷ parallel reduction to minimum processor ID
7:         if p = W[v] then
8:             add (L_k[v], v) to E_{k+1}
```

Theorem 1. *All connected components in G can be found by Algorithm 2 on a CRCW PRAM in $O(\log D)$ time and $O((m+n)\log D)$ work using $O(m+n)$ processors.*

Proof. Evaluating minimum labels requires indexing into the L_k array which takes $O(1)$ work. The update to L_{k+1} is concurrent under Combining Write Resolution and takes $O(1)$ work. All comparisons and edge creation operations take $O(1)$ work. Therefore each step takes $O(1)$ work per edge and by Lemma 4 the work does not require load-balancing. Since there are $O(m+n)$ edges each step according to Lemma 2, and according to Lemma 3 it takes $O(\log D)$ steps to converge, there is $O((m+n)\log D)$ total work. Then given $O(m+n)$ processors it takes $O(\log D)$ time in total. □

On small-world graphs the diameter can be $D \leq O(\log n)$ [5] so Algorithm 2 would take $O(\log \log n)$ time and $O((m+n)\log \log n)$ work, which is more efficient than the Shiloach-Vishkin algorithm. Algorithm 2 also has the advantages of not reading all edges twice in each iteration step and it does not use pointer operations that in practice can incur higher latency.

Note that a EREW algorithm follows directly from Algorithm 2 because it is well-known that a CRCW algorithm can be simulated in a EREW with logarithmic factor slowdown [28]. The only read/write conflict in Algorithm 2 is in the minimum label update. Thus for a p-processor EREW, reading $L_k[v]$ takes $O(\log p)$ time by broadcasting the value in binary tree order to each processor attempting to read it. It isn't difficult to see that a minimum value can be found in $O(\log p)$ time using a binary tree layout to reduce comparisons by half each step [2].

Theorem 2. *All connected components in G can be found by Algorithm 2 on a EREW PRAM in $O(\log n \log D)$ time and $O((m+n)\log n \log D)$ work using $O(m+n)$ processors.*

The fastest $\mathcal{NC}$ EREW algorithm takes $O(\log^{3/2} n)$ time using $O(m+n)$ processors [26]. On a EREW our Algorithm 2 is only $O(\log^{1/2} n)$ factor less efficient in the worst case, but it is much simpler to implement. This result also holds on a CREW PRAM. We will now show how to extend our framework to Stream and MapReduce computational models.

[2] Given an array M having n values and $p = \lceil n/2 \rceil$ processors, then at each step $M[i] \leftarrow min(M[2i-1], M[2i])$ for $1 \leq i \leq p$, where p is halved after each step.

8 Extending to other models

The Stream and MapReduce models restrict globally-shared state so the minimum label for each vertex must be carried with the graph at each step. In MapReduce the map and reduce functions are sequential so a giant component that is processed by one task will serialize the entire algorithm. We address these limitations by slightly altering the label propagation and symmetrization operations.

Label propagation will now only proceed from vertices where $v \neq l(v)$ to mitigate sequential processing of a giant component, thus skipping over intermediate representative labels. The symmetrization operation will add both edges, $(l(v), v), (v, l(v))$, so that v is always paired with its minimum label due to the absence of random access to global state. Recall in Algorithm 1 that a vertex may not have an explicit edge with its minimum label and must get $l(v)$ from the global L_k array, but in Stream and MapReduce we use the symmetrization operation to carry the minimum label with each vertex at every step. We also remark that symmetrization must be this way when ignoring v where $v \neq l(v)$ because the vertices that would have created the edge $(v, l(v))$ are now ignored.

These minor changes do not invalidate the correctness or convergence established by Algorithm 1 because all the same edges are created but with some added duplicates. The primary difference is that both $(v, l(v)), (l(v), v)$ edges are created in the same step rather than strided across two sequential steps. So if the new label propagation operation ignores any v where $v = l(v)$, the $(l(v), u)$ edges that would have been created by Algorithm 1 will get created during symmetrization for each u neighbor of such a v. Thus the new symmetrization operation creates the $(v, l(v))$ edges that are not created by the new label propagation operation. The disadvantage to not splitting the $(v, l(v)), (l(v), v)$ across two steps is it incurs label duplication that can lead to a progressive increase in edges if left unchecked.

9 Label duplication

The new label propagation and symmetrization for Stream and MapReduce can lead to $O(\log n)$ factor inefficiency as a result of increased edge count from label duplication. Generally this increase can be due to duplicate labels returned by different neighbors, or labels that had already been received in previous steps. The duplicates can increase the graph size progressively on some graphs such as sequentially labeled path or tree graphs. This leads to the following crucial observations (see Appendix for proofs).

Observation 1. *At each step k of Algorithm 1 on a path graph sequentially labeled from $1..n$, the label difference follows a Fibonacci recurrence, specifically $\Delta_k(v, l_k(v)) = v - l_k(v) = F_k$, where $F_k = F_{k-1} + F_{k-2}$. Therefore the minimum label of every vertex follows $l_{k+1}(v) = l_k(v) + l_{k-1}(v) - v$ until $l_{k+1} = 1$.*

Observation 2. *Adding both counter-oriented edges during symmetrization in Algorithm 1, on a path graph that is sequentially labeled from $1..n$, will pair a v with each of its new $l(v)$ for three steps.*

Once an $l(v)$ is replaced with an updated minimum for v, it is no longer needed and only adds to the edge count. Each vertex in a chain is a minimum label to vertices up the chain, then each vertex will in turn be back-propagated down the chain. Since this follows a Fibonacci sequence the duplication of labels is grows rapidly. For example we can see from Observation 1 that vertex 2 in a chain will pair with vertices $3, 4, 5, 7, 10, 15, 23, \ldots$, and each of these vertices in the sequence will return vertex 2 back to the neighbor from which it received vertex 2. Subsequently, every

Algorithm 3

initialize E_1 with sorted E and set $last_v := \infty$
▷ Use one bit for marking edges, e.g. $OLD = 0, NEW = 1$
1: **for** $k = 1, 2, \ldots$ until labels converge **do**
2: **for** $(v, u) \in E_k$ **do**
3: **if** $v \neq last_v$ **then**
4: set $l(v) := \min(v, u)$ and $last_v := v$
5: **if** $l(v) \neq v$ **then**
6: add $\big((v, l(v)), NEW\big)$ and $\big((l(v), v), NEW\big)$ to E'_{k+1}
7: **else if** $l(v) \neq v$ **then**
8: add $\big((u, l(v)), NEW\big)$ and $\big((v, u), OLD\big)$ to E'_{k+1}
9: sort E'_{k+1} ▷ NEW and OLD edges get sorted together for each (v, u)
10: **if** $\big((v, u), NEW\big) \in E'_{k+1}$ but $\big((v, u), OLD\big) \notin E'_{k+1}$ **then**
11: add (v, u) to E_{k+1}

vertex in the chain is retained by the neighbors with greater vertex ID following the Fibonacci sequence. Therefore as seen in Observation 2, each new $l(v)$ is retained by v for three steps thereby progressively increasing the edge count. Relabeling the graph can avoid the pathological duplication but a robust algorithm is more desirable, especially in graphs that are not totally sequential but may contain a very long chain. On a PRAM this can be addressed by not creating an (v, u) edge if $u = l(v)$ in any of the last three steps, but this offers no general improvement over Algorithm 2. For Stream and MapReduce models, we will leverage sorting to remove duplicates.

Suppose now that from v a $(u, l(v))$ edge is added to E_{k+1} only if that edge is not currently in E_k. This is essentially testing that if $l_k(v) \notin N_k(u)$ then it can be added to $N_{k+1}(u)$. Such a membership test can be handled by using a hash table or an adjacency matrix but leads to either average worst-case runtime or $O(n^2)$ space. Instead we leverage sorting to not only identify the next minimum label for each vertex but also remove labels that would otherwise fail the membership test.

Let E'_{k+1} be the intermediate edges that are created during the k^{th} step and from which a subset are retained for the $k + 1$ step. Sorting edges in E_k and E'_{k+1} will identify those edges that are duplicated across both steps and therefore should be removed. But an edge that is duplicated in only E'_{k+1} must be retained for proper label propagation. To avoid inadvertently removing such an edge, all duplicates in the E'_{k+1} are first removed before merging and sorting with edges from the k step. Then any remaining duplicate edges are those from both E_k and E'_{k+1} and should be removed. Hence all remaining edges with multiplicity greater than one must be removed. The edges from E_k are only needed to determine the membership test and so these are also removed before the next step. We apply this in our next algorithms.

10 Stream-Sort algorithm

We turn our attention to the Stream-Sort model, which permits a sorting pass on intermediate output streams. We derive Algorithm 3 from Algorithm 1 but extend it as described in Section 8. Duplicates are removed by sorting in the manner described at the end of Section 9. It requires two stages per iteration step. The first stage performs label propagation and symmetrization and also recreates the existing edges. The second stage eliminates duplicates. A one-stage algorithm [6] was described in 2016, which is simpler to implement, but does not address label duplication.

Recall that in each step our framework does not return the current edges but creates new

edges by the symmetrization and label propagation operations. But in the Stream-Sort model we must return the current edges temporarily in the intermediate sorting stage in order to ignore duplicate edges. Hence we mark these edges to distinguish old from new. Here a *NEW* edge resulted from either symmetrization or label propagation. An *OLD* edge is an existing edge that is returned for the sorting pass. In the first stage Algorithm 3 reads sorted edges, hence $l(v) = \min(v, u)$ from the first edge of v. If $l(v) \neq v$ then the symmetrization operation adds $\big((v, l(v)), NEW\big), \big((l(v), v), NEW\big)$ edges to an intermediate stream E'_{k+1}. Then for each subsequent edge of v a $\big((u, l(v)), NEW\big)$ edge is added to E'_{k+1} by label propagation. An identity operation also adds $\big((v, u), OLD\big)$. The intermediate stream E'_{k+1} is sorted by a sorting pass and input to the second stage. In the second stage the edges are sorted so all *NEW* and *OLD* versions for (v, u) are grouped together. Then any edge that results solely from either symmetrization or label propagation is added to a new final stream E_{k+1}, which will be the input stream in the next pass. Both intra- and inter-step duplicates have been removed. The algorithm repeats this procedure until no new minimum can be adopted.

Theorem 3. *All connected components in G can be found by Algorithm 3 in Stream-Sort taking $O(\log D)$ passes and $O(\log n)$ memory.*

Proof. The input stream E_k is scanned in one pass and the intermediate output stream E'_{k+1} is subsequently sorted in a single sorting pass. There are then a constant number of sorting passes each step, which are essentially free. Only duplicate edges are removed so the total edge count per step is $O(m + n)$ following from Lemma 2. Moreover sorting at each step requires only $O(\log n)$ bits of memory to compare $l(v)$ with another vertex. Thus the sorting pass needs $O(\log n)$ memory. By Lemma 3 it takes $O(\log D)$ steps for convergence so there are a total of $O(\log D)$ passes. Overall it takes $O(\log D)$ passes and $O(\log n)$ memory. □

This is the first efficient, deterministic connected components algorithm in Stream-Sort.

11 MapReduce algorithm

Our new MapReduce algorithm described in Algorithm 4 is very similar to Algorithm 3, following the same principle of removing duplicates introduced in Section 9. It takes two rounds per iteration step, the first to perform label propagation and symmetrization and the second to remove duplicates. A single-round algorithm[6] with less efficient communication is also available to the interested reader.

The values for each key are sorted hence intra-step duplicates are adjacent and easily removed, permitting the algorithm to maintain $O(1)$ working memory. Since the values are sorted then $l(v)$ is simply the lesser between the key and first value. We omit the specifics on sorting and getting $l(v)$ for brevity. The first round performs label propagation and symmetrization for every v where $v \neq l(v)$, but each newly created edge is marked *NEW* to indicate that it came from either of these two operations. All current edges are also returned but marked *OLD* to indicate an identity operation. The second round accumulates the marked edges and every edge without an *OLD* counterpart is returned with no markings. Thus the first reduce function returns $\langle u, (l(v), NEW)\rangle$ and $\langle v, (u, OLD)\rangle$ for each neighbor u. In the next round the key-value pairs are accumulated so it is possible to get the input $\langle v, \{(u, NEW), (w, OLD), (w, NEW), \ldots\}\rangle$ and hence w would be ignored. The two rounds are repeated until labels converge. Since duplicates are removed the total communication cost is $O(m + n)$ per round.

Theorem 4. *All connected components in G can be found by Algorithm 4 in $O(\log D)$ rounds using $O((m + n) \log D)$ communication overall.*

Algorithm 4

▷ Use one bit for marking edges, e.g. $OLD = 0, NEW = 1$
procedure REDUCE-1($key = v, values = N_k(v)$) ▷ sorted values
  set $l(v) := \min(N_k^+(v))$
  if $l(v) \neq v$ **then**
    emit $\langle v, (l(v), NEW)\rangle$ and $\langle l(v), (v, NEW)\rangle$
    for $u \in values : l(v) \neq u$ **do**
      emit $\langle u, (l(v), NEW)\rangle$ and $\langle v, (u, OLD)\rangle$
procedure REDUCE-2($key = v, values = \{(u, i) : i \in \{OLD, NEW\}\}$) ▷ sorted values
  if $\big((v, u), NEW\big) \in values$ but $\big((v, u), OLD\big) \notin values$ **then**
    emit $\langle v, u\rangle$
for $k = 1, 2, \ldots$ until labels converge **do**
  MAP $\mapsto$ Identity
  REDUCE-1
  MAP $\mapsto$ Identity
  REDUCE-2

Proof. The number of iteration steps is given by Lemma 3. There are two rounds per each iteration step leading to $2 \log D = O(\log D)$ total number of rounds. The communication cost is proportional to the number of edges written after all rounds. Both inter- and intra-step duplicates are removed in each iteration step so the total number of edges after the second round of each step is $O(m + n)$ according to Lemma 2. Thus for $O(\log D)$ rounds the overall communication is $O((m + n) \log D)$ as claimed. □

This is $O(\log n)$ factor better in runtime and communication than [29], making it the fastest deterministic MapReduce algorithm for connected components. Since MapReduce can be simulated by the more powerful *MPC* model, this is also faster than the best-known connectivity algorithm in the *MPC* model [2].

Acknowledgments

The author thanks his internal private reviewers and especially David G. Harris and Christopher H. Long.

A Proof of Observation 1

Proof. Let $\Delta(x, y) = x - y$ where $x > y$. At each step label propagation pairs a neighbor u with $l(v)$, also each $l(v)$ will get v due to symmetrization. Hence in the previous step there was a (w, u) edge where $u = l(w)$, so the edge (u, w) is created for the next step. The difference $\Delta(w, l(u))$ can be determined as follows. Using $w = \Delta(w, u) + u$ in $\Delta(w, l(u))$ gives

$$\Delta(w, l(u)) = w - l(u) = \Delta(w, u) + u - l(u)$$

Now let (u, w) be (v, u) in this step so,

$$\Delta(u, l(v)) = \Delta(v, l(v)) + \Delta(w, u)$$

The $\Delta(w, u)$ is just $\Delta(v, l(v))$ from the previous step. Then by induction over k steps the difference is

$$\Delta_k(u, l(v)) = \Delta_k(v, l(v)) + \Delta_{k-1}(v, l(v))$$

Setting $k = k - 1$ and $\Delta_k(v, l(v)) = \Delta_{k-1}(u, l(v))$ gives,

$$\Delta_k(v, l(v)) = \Delta_{k-1}(v, l(v)) + \Delta_{k-2}(v, l(v))$$

This is clearly a Fibonacci recurrence, therefore $\Delta_k(v, l(v)) = F_k$. Then expanding $\Delta_{k+1}(v, l(v))$ and solving for l_{k+1} yields the result as claimed. □

B Proof of Observation 2

Proof. Recall from Observation 1 that the label difference $v - l(v)$ follows a Fibonacci sequence $F_k = F_{k-1} + F_{k-2}$, hence $l_{k+1}(v) = l_k(v) + l_{k-1}(v) - v$. Since symmetrization retains each $l(v)$ for the next step then v gets the fixed label $l(v) = v - F_k$ for three steps because of the recurrence of F_k. Thus any new $l(v)$ that v receives will return to v a total of three steps unless $l(v)$ is the minimum label for the component of v. □

Bibliography

References

[1] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, FOCS'04, pages 540–549, 2004.

[2] A. Andoni, C. Stein, Z. Song, Z. Wang, and P. Zhong. Parallel graph connectivity in log diameter rounds, 2018. To appear in Proceedings of the 59th Annual IEEE Symposium on Foundations of Computer Science.

[3] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and pram. *IEEE Transactions on Computers*, C-36:1258–1263, 1987.

[4] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors. In *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS'04, 2004.

[5] B. Bollobás and O. Riordan. The diameter of scale-free random graph. *Combinatorica*, 24(1):5–34, 2004.

[6] P. Burkhardt. Easy graph algorithms under hard constraints. Technical report, U.S. National Security Agency, May 2016. Keynote at the 2016 Seventh Annual Graph Exploitation Symposium (GraphEx).

[7] P. Burkhardt and C. A. Waring. A cloud-based approach to big graphs. In *Proceedings of the 19th annual IEEE conference on High Performance Extreme Computing*, HPEC '15, pages 1–8, 2015.

[8] D. K. Campbell. A survey of models of parallel computation. Technical report, Unversity of York Department of Computer Science, March 1997.

[9] F. Y. L. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, 1982.

[10] R. Chitnis, G. Cormode, M. T. Hajiaghayi, and M. Monemizadeh. Parameterized streaming: maximal matching and vertex cover. In *Proceedings of the 26th ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, pages 1234–1251, 2015.

[11] K. W. Chong and T.-W. Lam. Finding connected components in $O(\log n \log \log n)$ time on the EREW PRAM. *Journal of Algorithms*, 18:378–402, 1995.

[12] S. Chung and A. Condon. Parallel implementation of Boruvka's minimum spanning tree algorithm. In *Proceedings of International Conference on Parallel Processing*, ICPP '96, pages 302–308, 1996.

[13] R. Cole and U. Vishkin. Approximate parallel scheduling. part ii: Applications to logarithmic-time optimal graph algorithms. *Information and Computation*, 92:1–47, 1991.

[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on symposium on operating systems design and implementation*, OSDI '04, pages 137–150, 2004.

[15] C. Demetrescu, B. Escoffier, G. Moruz, and A. Ribichini. Adapting parallel algorithms to the w-stream model, with applications to graph problems. *Theoretical Computer Science*, 411(44–46):3994–4004, 2010.

[16] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. In *Proceedings of the 17th annual ACM-SIAM symposium on discrete algorithms*, SODA '06, pages 714–723, New York, NY, USA, 2006. Society for Industrial and Applied Mathematics.

[17] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the ACM Symposium on Theory of Computing*, STOC'78, pages 114–118, 1978.

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

[19] S. Goddard, S. Kumar, and J. F. Prins. Connected components algorithms for mesh-connected parallel computers. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge*, 1995.

[20] M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching, and simulation in the MapReduce framework. In *Proceedings of International Symposium on Algorithms and Computation*, ISAAC'11, pages 374–383, 2011.

[21] S. Halperin and U. Zwick. An optimal randomized logarithmic time connectivity algorithm for the erew pram. *Journal of Computer and System Sciences*, 53:395–416, 1996.

[22] Y. Han and R. A. Wagner. An efficient and fast parallel-connected component algorithm. *Journal of the ACM*, 37(3):626–642, 1990.

[23] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report 1998-011, DEC Systems Research Center, 1998.

[24] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[25] D. B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for the crew pram. *Journal of Computer and Systems Sciences*, 54(2):227–242, 1997.

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

[27] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the 21st annual ACM-SIAM symposium on discrete algorithms*, SODA '10, pages 938–948, 2010.

[28] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869–932. MIT Press, Cambridge, MA, USA, 1990.

[29] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilviskii. Connected components in MapReduce and Beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC'14, pages 1–13, 2014.

[30] R. Li, H. Hu, H. Li, Y. Wu, and J. Yang. MapReduce parallel programming model: A state-of-the-art survey. *International Journal of Parallel Programming*, 44(4):832–866, 2016.

[31] A. McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[32] A. McGregor, S. Vorotnikova, and H. T. Vu. Better algorithms for counting triangles in data streams. In *Proceedings of the 35th ACM Symposium on Principles of Database Systems*, PODS'16, pages 401–411, 2016.

[33] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12(3):315–323, 1980.

[34] T. C. O'Connell. A survey of graph algorithms under extended streaming models of computation. In *Fundamental Problems in Computing*, pages 455–476. Springer, 2009.

[35] M. M. A. Patwary, P. Refsnes, and F. Manne. Multi-core spanning forest algorithms using the disjoint-set data structure. In *Proceedings of the International Parallel and Distributed Processing Symposium*, IPDPS'12, pages 827–835, 2012.

[36] A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. Space-round tradeoffs for MapReduce computations. In *Proceedings of the 2012 ACM international conference on supercomputing*, SC '12, pages 235–244, 2012.

[37] V. Rastogi, A. Machanavjjhala, L. Chitnis, and A. D. Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Proceedings of the 2013 IEEE international conference on data engineering*, ICDE'13, pages 50–61, 2013.

[38] A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM*, 13(4):471–494, 1966.

[39] M. Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 2003.

[40] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):579–586, 1988.

[41] B. Schmidt, J. Gonzalez-Dominguez, C. Hundt, and M. Schlarb. *Parallel Programming: Concepts and practice*. Morgan Kaufmann, 1st edition, 2017.

[42] L. G. Shapiro. Connected component labeling and adjacency graph construction. *Machine Intelligence and Pattern Recognition*, 19:1–30, 1996.

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

[44] J. Shun, L. Dhulipala, and G. E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 2014 ACM symposium on parallelism in algorithms and architectures*, SPAA'14, pages 143–153, 2014.

[45] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2), 1972.