ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms

Laxman Dhulipala
CMU
ldhulipa@cs.cmu.edu

Changwan Hong
MIT CSAIL
changwan@mit.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

ABSTRACT

Connected components is a fundamental kernel in graph applications due to its usefulness in measuring how well-connected a graph is, as well as its use as subroutines in many other graph algorithms. The fastest existing parallel multicore algorithms for connectivity are based on some form of edge sampling and/or linking and compressing trees. However, many combinations of these design choices have been left unexplored. In this paper, we design the ConnectIt framework, which provides different sampling strategies as well as various tree linking and compression schemes. ConnectIt enables us to obtain several hundred new variants of connectivity algorithms, most of which extend to computing spanning forest. In addition to static graphs, we also extend ConnectIt to support mixes of insertions and connectivity queries in the concurrent setting.

We present an experimental evaluation of ConnectIt on a 72-core machine, which we believe is the most comprehensive evaluation of parallel connectivity algorithms to date. Compared to a collection of state-of-the-art static multicore algorithms, we obtain an average speedup of 37.4x (2.36x average speedup over the fastest existing implementation for each graph). Using ConnectIt, we are able to compute connectivity on the largest publicly-available graph (with over 3.5 billion vertices and 128 billion edges) in under 10 seconds using a 72-core machine, providing a 3.1x speedup over the fastest existing connectivity result for this graph, in any computational setting. For our incremental algorithms, we show that our algorithms can ingest graph updates at up to several billion edges per second. Finally, to guide the user in selecting the best variants in ConnectIt for different situations, we provide a detailed analysis of the different strategies in terms of their work and locality.

PVLDB Reference Format:
Laxman Dhulipala, Changwan Hong, and Julian Shun. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. PVLDB, 14(1): xxxx-yyyy, 2020.
doi:xx.xx/xxx.xx

1 INTRODUCTION

Computing the connected components (connectivity) of an undirected graph is a fundamental problem for which numerous algorithms have been designed. In the connected components problem, we are given an undirected graph and the goal is to assign labels to the vertices such that two vertices reachable from one another have the same label, and otherwise have different labels [25]. A recent paper by Sahu et al. [88] surveying industrial uses of graph algorithms shows that connectivity is the most frequently performed graph computation out of a list of 13 fundamental graph routines including shortest paths, centrality computations, triangle counting, and others. Computing connected components is also used to solve many other graph problems, for example, to solve biconnectivity and higher-order connectivity [105], as well as a subroutine in popular clustering algorithms [36, 37, 39, 83, 110, 111].

In the sequential setting, connected components can be easily solved using breadth-first search, depth-first search, or union-find. However, due to the multiple processing units available on most platforms today, it is important to have fast parallel algorithms for the problem. Dozens of parallel algorithms for connected components have been proposed in the literature over the past four decades (see, e.g., [4–9, 12–14, 16, 18–24, 31, 38, 40, 43, 45–48, 50, 52–56, 60–65, 67, 69, 72, 76, 77, 80, 84–87, 90, 91, 94, 98, 100, 101, 104, 106, 107, 112, 113] among many others). Recent state-of-the-art parallel implementations are based on graph traversal [31, 91, 94, 98], label-propagation [77, 91, 98], union-find [16, 55, 84], or the hook-compress paradigm [5, 7, 11, 14, 24, 43, 67, 72, 80, 90, 100, 109, 113]. Recent work by Sutton et al. [104] uses sampling to find the connected components on a subset of the edges, which can be used to reduce the number of edges that need to be inspected when running connectivity on the remaining edges. However, most prior work has provided only one, or a few implementations of a specific approach for a particular architecture, and there are many variants of these algorithmic approaches that have been left unexplored.

In this paper, we design the ConnectIt framework for multicore CPUs, which enables many possible implementation choices of the algorithmic paradigms for parallel connectivity described in the literature. Furthermore, as many real-world graphs are frequently updated under insertion-heavy workloads (for example, there are about 6,000 tweets per second on Twitter's network, but only a few percent of tweets are deleted [3]), ConnectIt provides algorithms that can maintain connectivity under incremental updates (edge insertions) to the graph. A subset of the implementations in ConnectIt also support computing the spanning forest of a graph in both the static and incremental settings. We focus on the multicore setting as the largest publicly-available real-world graphs can fit in the memory of a single machine [31, 33]. We also compare ConnectIt's results with reported results for the distributed-memory setting, showing that our multicore solutions are significantly faster and much more cost-efficient. In the following, we give a high-level overview of ConnectIt, and how it relates to prior work. Figure 1 provides an overview of the features and functionality supported by the ConnectIt framework, as described above.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:xx.xx/xxx.xx

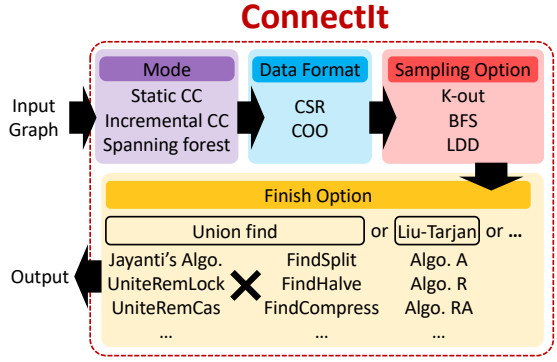


Figure 1: This figure shows an overview of the features and functionality supported by the ConnectIt framework. For a given connectivity problem (Mode), and graph input format (Data Format), users can specify different sampling strategies (Sampling Option) to combine with different finish algorithms (Finish Option). Only a few of the possible finish algorithms supported by ConnectIt are shown here. We describe the sampling strategies in detail in Section 3.2, and the finish algorithms in Section 3.3.

ConnectIt Overview

Union-Find. Union-find algorithms for connectivity maintain vertices in disjoint sets, represented as rooted trees. Sets are combined based on membership of the endpoints of each edge by changing the parent pointer of the root of one tree to point to a vertex in another tree. At termination, the disjoint sets define the connected components of the graph. ConnectIt includes different concurrent union-find algorithms, obtained by using different rules for uniting two sets and for path compression during find operations. The rules for union are based on concurrent versions of Rem's algorithm [81], randomized linking by ID [58], and randomized linking by rank [59]. The path compression techniques include path-halving, path-splitting, full path compression, and splicing.

The concurrent randomized linking by ID and randomized linking by rank algorithms were proposed and analyzed theoretically by Jayanti et al. [58, 59]. A variant of one of these algorithms using path-halving was recently implemented on GPUs by Jaiganesh and Burtscher [55]. ConnectIt incorporates all of the implementation choices in the algorithms by Jayanti et al. We also introduce the full path compression technique, which we found to work better in practice in some cases. Patwary et al. [81] present two multicore implementations of Rem's algorithm with splicing, one which is a natural extension of the serial algorithm to use locks, and the other is a lock-free implementation using speculative execution (we found their lock-free implementation to not always produce the correct result). ConnectIt includes the lock-based Rem's algorithm by Patwary et al. [81], but also introduces a lock-free version using compare-and-swap. In addition to splicing, ConnectIt also supports path-halving and path-splitting for path compression in Rem's algorithm. We prove that our multicore implementations of Rem's algorithm are *phase-concurrent* [92] in that the algorithm produces correct results as long concurrent operations at any time only consist of either only unions or only finds. Blelloch et al. [16] present a deterministic multicore union-find connectivity algorithm based on linking by ID that is also phase-concurrent.

Very recently, Alistarh et al. [2] performed a performance evaluation of concurrent union-find implementations for multicores on much smaller graphs. They introduce a lock-free version of concurrent Rem's algorithm with path-splitting similar to our implementation, but do not prove that the algorithm is correct, or consider other path-compaction methods in their algorithm. ConnectIt implements a similar lock-free Rem's algorithm that is more general, allowing the algorithm to be combined with path-halving, and splicing in addition to path-splitting.

Using ConnectIt, we can obtain a total of 144 different concurrent union-find implementations. Our union-find algorithms can be used both for connected components and for spanning forest, and support incremental updates to the graph. Similar in spirit to the experimental analysis of 55 different sequential union-find implementations by Patwary et al. [82], we analyze the performance of our 144 union-find implementations in the parallel setting.

Other Root-Based Algorithms. We broadly classify the union-find algorithms above in our framework as *root-based algorithms*, in that each of their union operations only performs a link if it operates on the root of a tree. Other than union-find, another classic parallel connected components algorithm that falls under this categorization is the Shiloach-Vishkin (SV) algorithm, which ConnectIt includes. Many variants of SV have been described in the literature [5, 7, 11, 14, 24, 43, 72, 80, 100, 109, 113], but none of these implementations combine SV with sampling approaches, as done in this paper.

Min-Based Algorithms. *Min-based algorithms* are algorithms that scan all edges and for each edge, apply an operation that modifies the parents of the endpoints over a number of rounds. On each round, the algorithm tries to update the parents of vertices based on the values of their current parents, updating a parent only if the new parent is *smaller* than the previous one, hence the name for this category. The algorithm terminates once the parent values stop changing. One example of such an algorithm is the folklore label-propagation algorithm.

In ConnectIt, we implement a recent min-based algorithm by Stergiou et al. [101], as well as 16 algorithms from a framework described recently by Liu and Tarjan [67], but were not implemented. To the best of our knowledge, these are the first implementations and experimental evaluation of the algorithms by Liu and Tarjan. Stergiou et al. [101] provide a distributed-memory implementation of their algorithm, which we adapt to the multicore setting. A subset of the Liu-Tarjan algorithms perform updates only to roots of trees, and thus can also be categorized as root-based algorithms.

Sampling and Two-phase Execution. Inspired by the Afforest algorithm [104], ConnectIt produces algorithms with two phases: the ***sampling*** phase and the ***finish*** phase. In the sampling phase, we run connected components on a subset of the edges in the graph, which assigns a temporary label to each vertex. We then find the most frequent label, $L_{\max}$, which corresponds to the ID of the largest component (not necessarily maximal) found so far. In the finish phase, we only need to run connected components on the incident edges of vertices with a label not equal to $L_{\max}$, which can significantly reduce the number of edge traversals. We observe that this optimization is similar in spirit to the direction-optimization

in breadth-first search [10], which skips over incoming edges for vertices, once they have already been visited during dense iterations. If sampling is not used, then the finish phase is equivalent to running one of the previously described algorithms on all vertices and edges.

The sampling strategy of the Afforest algorithm [104] runs connectivity on the first k edges out of each vertex using a variant of union-find, and then uses the same union-find algorithm for the finish phase. CONNECTIT implements a sampling strategy similar to Afforest, but instead of picking the first k edges out of each vertex, we pick the first edge out of every vertex, and $k-1$ random edges out of each vertex to avoid adversarial inputs. We then use any of our 144 union-find variants on these edges. We refer to this strategy as ***k-out***. CONNECTIT includes another sampling strategy, which is to run direction-optimizing breadth-first search from a random sample of vertices, such that the largest connected component is likely to be explored. $L_{\max}$ is set to be the label of vertices in the largest component found. Lastly, we include a sampling strategy based on computing a low-diameter decomposition (LDD) [31, 75, 94] of the graph, which partitions the graph into connected sets of vertices, such that the diameter of each set is small and the number of edges between sets is small. $L_{\max}$ is then set to be the label of vertices in the largest set in the decomposition.

Our main observation is that these sampling techniques are general strategies that reduce the number of edge traversals in the finish phase, and thus accelerate the overall algorithm. In CONNECTIT, any of the union-find, root-based algorithms, and min-based algorithms can be used for the finish phase in combination with any of the three sampling schemes. For the implementations where the finish phase is a root-based algorithm, CONNECTIT also supports spanning forest computation. Our generalized sampling paradigm and integration into CONNECTIT enables us to express over 232 combinations of parallel algorithms of parallel graph connectivity and 192 implementations for parallel spanning forest (note that only our root-based algorithms are supported for spanning forest).

Incremental Connectivity. Due to the frequency of updates to graphs, various parallel streaming algorithms for connected components have been developed [1, 32, 34, 71, 89, 97]. Motivated by this, the CONNECTIT framework supports a combination of edge insertions and connectivity queries in the graph. Both the union-find and root-based algorithm implementations in CONNECTIT support batched edge insertions/queries. Additionally, all of the union-find implementations other than Rem's algorithms combined with the splice compression scheme support asynchronous updates and queries, and most of them are lock-free or wait-free.

Experimental Evaluation. The CONNECTIT framework enables one to easily combine different algorithmic choices into a single program to achieve high performance for parallel connectivity and spanning forest. We conduct a comprehensive experimental evaluation of all of the connectivity implementations in CONNECTIT on a 72-core multicore machine, in both the static and incremental setting. Using hardware performance counters, we analyze the tradeoffs of different strategies in terms of their work and locality. The trends for the spanning forest implementations are similar to connectivity, and on average the additional overhead needed to obtain the spanning forest compared to connectivity is 23.7%.

System	Graph	Mem. (TB)	Threads	Nodes	Time (s)
Mosaic [68]	Hyperlink2014	0.768	1000	1	708
FlashGraph [26]	Hyperlink2012	.512	64	1	461
GBBS [31]	Hyperlink2012	1	144	1	25.8
GBBS (NVRAM) [33]	Hyperlink2012	0.376	96	1	36.2
Galois (NVRAM) [41]	Hyperlink2012	0.376	96	1	76.0
Slota et al. [99]	Hyperlink2012	16.3	8192	256	63
Stergiou et al. [101]	Hyperlink2012	128	24000	1000	341
Gluon [27]	Hyperlink2012	24	69632	256	75.3
Zhang et al. [113]	Hyperlink2012	$\geq$ 256	262,000	4096	30
CONNECTIT	Hyperlink2014	1	144	1	**2.83**
	Hyperlink2012	1	144	1	**8.20**

Table 1: System configurations, including the memory in terabytes, number of hyper-threads and nodes, and running times (seconds) of existing connectivity results on the Hyperlink graphs. The Hyperlink2012 graph has over 200 billion edges, and is the largest publicly-available graph today. The last rows show the fastest times in CONNECTIT. The fastest time for each graph is bolded in green.

Compared to existing work, CONNECTIT implementations significantly outperform state-of-the-art implementations using the new sampling techniques proposed in this paper, and often outperform prior work even without applying sampling. Our fastest algorithms using sampling are always faster than the fastest currently available implementation. As an example of our performance, CONNECTIT is able to produce implementations that can process the Hyperlink2012 graph [73], which is the largest publicly-available real-world graph and has over 3.5 billion vertices and 128 billion edges, on a single 72-core machine with a terabyte of RAM. We show existing results on this graph, and the smaller Hyperlink2014 graph (1.7 billion vertices and 64 billion edges) in Table 1. Our running times are between 3.65–41.5x faster than existing distributed-memory results that report results for Hyperlink2012, while using orders of magnitude fewer computing resources. CONNECTIT also significantly outperforms existing state-of-the-art multicore connectivity algorithms for this graph, improving on the previous fastest algorithm from GBBS [31] by over 3.14x on the same machine.

Contributions. The contributions of this paper are as follows.

(**1**) We introduce the CONNECTIT framework, which provides several hundred different multicore implementations of parallel connectivity, spanning forest, and incremental connectivity, most of which are new.

(**2**) CONNECTIT provides different choices of sampling methods based on provably-efficient graph algorithms, which can be used in two-phase execution to reduce the number of edge traversals.

(**3**) The fastest implementation in CONNECTIT achieves an average speedup of 2.3x (and ranges from 1.5–4.02x speedup) over the fastest existing static multicore connectivity algorithms.

(**4**) For incremental connectivity, the multicore implementations in CONNECTIT achieve a speedup of 1,461–28,364x over existing multicore solutions and a throughput of between 4.6M updates per second for very small batch sizes to 7.1 billion edges per second for large batches, on graphs of varying sizes.

(**5**) We present a detailed experimental analysis of the different implementations in CONNECTIT to guide the user into finding the most efficient implementation for each situation.

(**6**) The CONNECTIT source code will be made publicly-available.[1]

[1] A preliminary version of the code is available in the connectivity repository in the Graph Based Benchmark Suite: https://github.com/ParAlg/gbbs.

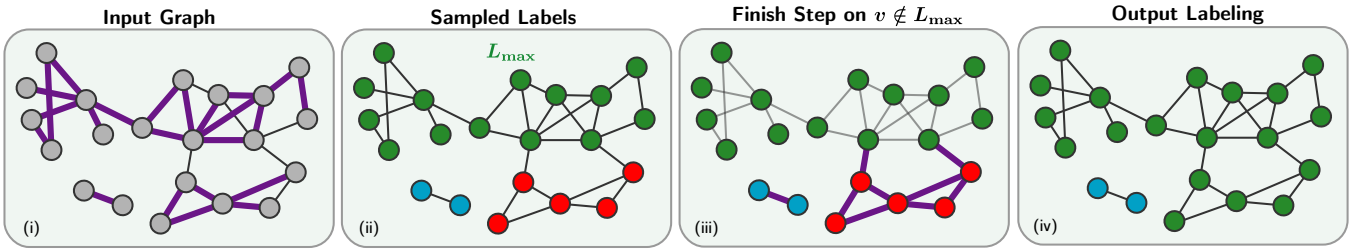


Figure 2: This figure illustrates the CONNECTIT framework for connectivity using k-out Sampling on a small input graph. **(i)** illustrates the input graph. The bolded purple edges are those selected by the k-out sample with $k = 2$. **(ii)** shows the partial connectivity labeling after computing connectivity on the sampled edges (e.g., through a union-find algorithm). $L_{\max}$ indicates the vertices in the largest component (shown in green). **(iii)** shows the edges (bolded in purple) which still must be processed in the finish step, namely all edges incident to vertices $v \notin L_{\max}$. Lastly, **(iv)** shows the output connectivity labeling.

2 PRELIMINARIES

This section presents definitions of problems and primitives that will be used throughout this paper.

Graph Notation and Formats. We denote an unweighted graph by $G(V, E)$, where V is the set of vertices and E is the set of edges in the graph. We use $n = |V|$ to refer to the number of vertices and $m = |E|$ to refer to the number of edges. In our graph representations, vertices are indexed from 0 to $n - 1$, and we consider two graph formats—***compressed sparse row (CSR)*** and ***edge/coordinate list (COO)***. In CSR, we are given two arrays, I and A, where the incident edges of a vertex v are stored in $\{A[I[v]], \ldots, A[I[v + 1] - 1]\}$ (we assume $A[n] = m$). In COO, we are given an array of pairs (u, v) corresponding to edge endpoints.

Compare-and-Swap. A ***compare-and-swap (CAS)*** takes three arguments: a memory location x, an old value *oldV*, and a new value *newV*. If the value stored at x is equal to *oldV*, the CAS atomically updates the value at x to be *newV* and returns *true*; otherwise the CAS returns *false*. CAS is supported in almost all modern processors.

Work-Depth Model. We analyze our parallel algorithms using the ***work-depth measure*** [25, 57]. The ***work*** is the number of operations used by the algorithm and the ***depth*** is the length of the longest sequential dependence. The work-depth measure is a fundamental tool in analyzing parallel algorithms, e.g., see [15, 31, 33, 44, 95, 102, 103, 108] for a sample of recent practical uses of this model.

Linearizability. We use the notion of linearizability [49], which is the standard correctness criteria for concurrent algorithms. A set of operations are ***linearizable*** if the result of a concurrent execution is the same as if the operations were applied at a distinct point in time (the ***linearization point***) between the operation's invocation and response. The sequential ordering of the operations based on their linearization points is its ***linearization order***.

Graph Connectivity and Related Problems. In this paper, we consider an unweighted, undirected graph $G = (V, E)$, with vertex set V and edge set E. A ***connected component (CC)*** in G is a maximal set of vertices, such that there is a path between any two vertices in the set. An algorithm for computing connected components returns a labeling L for each vertex, such that $L(u) = L(v)$ if and only if vertices u and v are in the same connected component. A ***connectivity query*** takes as input two vertex identifiers and returns *true* if their labels are the same and *false* otherwise. A ***spanning forest (SF)*** in $G = (V, E)$ contains one tree for each connected component in G that contains all vertices of that component. A ***breadth-first search (BFS)*** algorithm takes as input a graph $G = (V, E)$ and a source vertex $src \in V$, and traverses the vertices reachable from src in increasing order of their distance from src. A ***low-diameter decomposition (LDD)*** of a graph parameterized by $0 < \beta < 1$ and d is a partition of the vertices into sets $V_1, \ldots, V_k$, such that the shortest path between any two vertices in the same partition using only intra-partition edges is at most d, and the number of inter-partition edges is at most βm [75].

A ***union-find*** (or disjoint-set) data structure maintains a collection of sets where all elements in each set are associated with the same label. The data structure is represented as a forest where each element has a parent pointer (with the root of each tree pointing to itself), and the label of an element is the root of the tree it belongs to. A union-find data structure supports the MAKESET, UNION, and FIND operations [25]. MAKESET(e) creates a new set containing just the element e with a parent pointer to itself. UNION(e,e') merges together the trees of e and e' into a single tree, if they are not already in the same tree. FIND(e) returns the label of element e by looking up the root of the tree that e belongs to. The FIND and UNION operations can compress paths in the tree to speed up future operations. When used in a CC algorithm, the elements are vertices, and at the end of the algorithm, FIND(v) returns $L(v)$. There are various versions of union-find based on how the UNION and FIND operations are implemented, and this paper explores a large number of union-find implementations for the concurrent setting.

A ***root-based*** connectivity algorithm also maintains a set data structure, like union-find algorithms. A root-based algorithm only links sets together by adding a link from the root of one tree, to a node in another tree. A ***min-based*** connectivity algorithm is a round-based algorithm which repeatedly performs an "application" step on all edges. The algorithm maintains a parents array, P, containing a single entry per vertex. An edge application takes an edge (u, v) and may update $P[u]$ based on $P[v]$ and vice versa, where a parent value is updated if and only if the new value is strictly less than the previous value. The algorithm terminates once the parent values stop changing.

3 CONNECTIT FRAMEWORK

In this section, we define the components of the CONNECTIT framework and the specific ways that they can be combined. We start by presenting our algorithm for computing connectivity. The algorithm supports combining multiple sampling methods and finish methods,

which we describe in detail in Sections 3.2 and 3.3. Figure 2 illustrates how the ConnectIt framework works for a k-out sampling algorithm on an example graph.

Next, in Sections 3.4 and 3.5, we describe modifications required to adapt our framework for spanning forest, and for the parallel batch-incremental setting. We present correctness proofs for our algorithms in all of the aforementioned settings in Appendix B. Finally in Section 3.6, we describe details regarding our implementation.

3.1 Connectivity

Algorithm 1 ConnectIt Framework: Connectivity

```
1: procedure Connectivity(G(V, E), sample_f, finish_f)
2:     sampling ← GetSamplingAlgorithm(sample_opt)
3:     finish ← GetFinishAlgorithm(finish_opt)
4:     labels ← {i → i | i ∈ [|V|]}
5:     labels ← sampling.SampleComponents(G, labels)
6:     frequentid ← IdentifyFrequent(labels)
7:     labels ← finish.FinishComponents(G, labels, frequentid)
8:     return labels
```

A connectivity algorithm in ConnectIt is instantiated by supplying *sampling* and *finish* methods. Algorithm 1 presents the generic ConnectIt connectivity algorithm, parameterized by these user-defined functions. The algorithm first initializes a *connectivity labeling* by setting each vertex's label to be its own ID (Line 4). It then performs a sampling step using the provided *sampling* algorithm (Line 5). The sampling step results in partial connectivity information being computed, stored in the *labels* array. Next, it identifies the most frequently occurring component ID in the *labels* array (Line 6). The identified component is then supplied to the *finish* algorithm, which finishes computing the connected components of G. The finish method potentially saves significant work by avoiding processing vertices in the component with an ID of *frequentid*.

Before discussing the sampling methods and finish methods, let us discuss the properties that we require in ConnectIt to obtain a correct parallel connectivity algorithm.

Properties of Sampling Methods. First, we introduce the following desirable properties for sampling methods.

Definition 3.1. Given a graph $G(V, E)$, let C be the labeling produced by a sampling method $\mathcal{S}$, and let $C' = \text{Connectivity}(G[C])$ where $G[C]$ is the graph induced by contracting G using the labeling C.[2] We note that this contraction is only for the purpose of analysis. Then,

(1) For each vertex v, either $C[v] = v$, or $C[v] = r$ and $C[r] = r$.
(2) $C'' = \{C'[C[v]] \mid v \in V\}$ is a correct connectivity labeling.

The definition captures what is required by a finish algorithm to "extend" the sampled, partial connectivity information to the full connectivity information of G.

Properties of Finish Methods. Next, we define the correctness properties for finish methods in ConnectIt.

Definition 3.2. A connectivity algorithm is ***monotone*** if the algorithm updates the labels so that the new labeling can be represented as the union of two trees in the previous labeling.

[2]Contracting G with respect to C creates a new graph by merging all vertices v with the same label into a single vertex, and only preserving edges (u, v) such that $C(u) \neq C(v)$, removing duplicate edges.

In other words, once a vertex is connected to its parent in a tree, it will always be in the same tree as its parent. Finally, we define linearizable monotonicity, which is the relaxed correctness property possessed by most finish algorithms in our framework.

Definition 3.3. Given an undirected graph $G(V, E)$, we say that a connectivity algorithm operating on a connectivity labeling C is ***linearizably monotone*** if

(1) Its operations are linearizable.
(2) Every operation in the linearization preserves monotonicity.

All finish methods considered in ConnectIt are linearizably monotone, with only a few exceptions. The first is a subset of Rem's algorithms which use the splice primitive, which we analyze separately (Theorem 3), and the family of min-based connectivity algorithms which includes a subset of Liu-Tarjan algorithms, Stergiou's algorithm, and Label-Propagation (Theorem 4).

Theorem 1. *The connectivity algorithm in* ConnectIt *(Algorithm 1) applied with a sampling method satisfying Definition 3.1 and a linearizably monotone finish method is correct.*

3.2 Sampling Algorithms

This section introduces the sampling methods used in ConnectIt. We provide pseudocode for our sampling methods in Appendix D.1.

***k*-out Sampling.** The k-out method takes a single parameter, k, and selects k edges out of each vertex uniformly at random, and computes the connected components of this sampled graph. The following main result shown in a recent paper by Holm et al. [51] is that if nk edges are sampled in this way for sufficiently large k, only $O(n/k)$ inter-component edges remain after contraction, in expectation. The authors of [51] conjecture that this fact about the number of inter-component edges holds for any $k \geq 2$, although the proof holds for $k = \Omega(\log n)$. Sutton et al. [104] describe a sampling scheme that selects the *first k edges* incident to the vertex, which does not use randomization. However, on some graphs with poor orderings, this method can result in only a small fraction of the components being discovered (leaving up to several orders more inter-component edges), which results in a costly sampling step that provides little benefit. To improve our results for these poorly ordered graphs while achieving good performance on graphs where this heuristic performs well, we select the first edge incident to each vertex, and select the remaining $k - 1$ edges randomly. To the best of our knowledge, using randomness in this experimental setting has not been explored before. We provide a full evaluation of different options in Appendix C.4.

Breadth-First Search Sampling. The breadth-first search (BFS) sampling method is a simple heuristic based on using a breadth-first search from a randomly chosen source vertex. Assuming that the graph contains a massive component, containing, say, at least a constant fraction of the vertices, running a BFS from a randomly selected vertex will discover this component with constant probability. To handle the case where we are unlucky and pick a bad vertex, we can apply this process c times. Setting $c = O(\log n)$ would ensure that we find the massive component with high probability, but in practice we set $c = 3$, and in our experiments we found that one try was sufficient to discover the massive component on all of

the real-world graphs we test on. CONNECTIT terminates the sampling algorithm when a component containing more than 10% of the vertices is found or after c rounds, whichever happens first.

Low-Diameter Decomposition Sampling. As discussed in Section 2, running an LDD algorithm on a graph with parameter $\beta < 1$ partitions the graph into clusters such that the strong diameter (the shortest path using only edges inside the cluster) of each cluster is $O(\log n/\beta)$, and cuts $O(\beta m)$ edges in expectation [75]. Shun et al. [94] give a simple and practical work-efficient parallel connectivity algorithm based on recursively applying LDD and performing graph contraction to recurse on a contracted graph.

In this paper, we consider applying just a *single* round of the LDD algorithm of Miller, Peng, and Xu [75], without actually contracting the graph after performing the LDD. On low-diameter graphs, the hope is that much of the largest connected component will be contained in the most frequent cluster identified. Our approach is practically motivated by studying the behavior of the work-efficient connectivity algorithm of Shun et al. [94] on low-diameter real-world networks and noticing that after one application of LDD, the resulting clustering contains a single massive cluster, and the number of distinct clusters in the contracted graph is extremely small. We provide an empirical analysis of this sampling method for different values of β in Appendix C.4 Finally, similar to prior work [31, 94], our implementation simulates sampling from an exponential distribution by first randomly permuting the vertices, and adding vertices according to an exponential distribution with parameter β in order of the permutation.

In Appendix B.1, we prove that k-out Sampling, BFS Sampling, and LDD Sampling all produce connectivity labeling satisfying Definition 3.1.

3.3 Finish Algorithms

We now describe different methods that can be used as ***finish*** methods in our framework. All of the finish methods that we describe can be combined with any of the sampling methods described in Section 3.2. We provide implementations of several different algorithm classes, which internally have many options that can be combined to generate different instantiations of the algorithm. The algorithms that we consider as part of the framework are union-find (many different variants, described below), Shiloach-Vishkin [90], Liu-Tarjan [67], Stergiou [101], and Label-Propagation. An important feature of CONNECTIT is in modifying these finish methods to *avoid traversing* the vertices with the most frequently occuring ID as identified from sampling in Algorithm 1. We provide pseudocode for our CONNECTIT implementations of all of the algorithms described in this section in Appendix B.

3.3.1 Union-Find. We consider several different concurrent (asynchronous) union-find algorithms. All of these algorithms are linearizably monotonic for a set of concurrent union and find operations, with the exception of the concurrent Rem's algorithms, which are linearizable only for a set of concurrent union operations or a set of concurrent find operations, but not for mixed operations. All of these algorithms combine with sampling by simply skipping traversing the vertices with label equal to *frequentid* after sampling.

Asynchronous Union-Find. The first class of algorithms are inspired by a recent paper exploring concurrent union-find implementations by Jayanti and Tarjan [58]. We implement all of the variants from their paper, as well as a full path compression technique (also considered in [2]) which works better in practice in some cases. We refer to this union-find algorithm as **Union-Async** since it is the classic union-find algorithm directly adapted for an asynchronous shared-memory setting. The algorithm links from lower-priority to higher-priority vertices to avoid cycles, and only performs links on roots (thus implying that the algorithm is monotonic). This algorithm can be combined with the following implementations of the find operation: **FindNaive**, which performs no compression during the operation, **FindSplit** and **FindHalve**, which perform path-splitting and path-halving, respectively, and **FindCompress**, which fully compresses the find path. Jayanti and Tarjan show that this class of algorithms is linearizable for a set of concurrent union and find operations (although they do not consider FindCompress, it is relatively easy to show that it is linearizable). The fact that these algorithms are linearizably monotonic for a set of concurrent union and find operations follows from the observation that they only link roots, and thus all label changes made by the operations are the result of taking the union of trees.

We also consider two similar variants of the Union-Async algorithm: Union-Hooks and Union-Early. **Union-Hooks** is closely related to Union-Async, with the only difference between the algorithms is that instead of performing a CAS directly on the array storing the connectivity labeling, we perform a CAS on an auxiliary *hooks* array, and perform an uncontended write on the *parents* array. **Union-Early** is also similar to Union-Async, except that the algorithm traverses the paths from both vertices together, and tries to eagerly check and hook a vertex once it is a root. The algorithm can optionally perform a find on the endpoints of the edge after the union operation finishes, which has the effect of compressing the find path. The linearizability proof for Union-Async by Jayanti and Tarjan [58] applies to Union-Hooks and Union-Early, and shows that both algorithms are linearizable for a set of concurrent finds and unions. Thus, these algorithms are also linearizably monotonic for a set of concurrent union and find operations as they only link roots.

Randomized Two-Try Splitting. Next, we incorporate a more sophisticated randomized algorithm by Jayanti, Tarjan, and Boix-Adserà [59], and refer to this algorithm as **Union-JTB**. The algorithm either performs finds naively, without using any path compression (**FindSimple**), or uses a strategy called **FindTwoTrySplit**, which guarantees provably efficient bounds for their algorithm, assuming a source of random bits. We give a faithful implementation of the algorithm presented by Jayanti, Tarjan, and Boix-Adserà [59], and refer to their paper for the pseudocode and proofs of correctness. In particular, they show that the algorithm has low total expected work, and low cost per operation with high probability. Since the Union-JTB algorithm is linearizable, and only links roots, the algorithm is also linearizably monotonic. Extending Theorem 4.1 in [59] to the work-depth setting, we have:

Corollary 1. *The* Union-JTB *algorithm solves connectivity in* $O(m \cdot (\alpha(n, m/(np)) + \log(1 + np/m)))$ *expected work and* $O(\log n)$ *depth with high probability.*

Concurrent Rem's Algorithms. We implement two concurrent versions of Rem's algorithm, a lock-based version by Patwary et al. [84] (**Union-Rem-Lock**) and a lock-free compare-and-swap based implementation (**Union-Rem-CAS**). Our implementations of Rem's algorithm can be combined with the same rules for path compression described for Union above, with one exception which we discuss below. In addition to path compression strategies, our implementations of Rem's algorithms take an extra ***splice*** strategy, which is used when a step of the union algorithm operates at a non-root vertex. Specifically, our algorithms support the **HalveAtomicOne**, **SplitAtomicOne**, and **SpliceAtomic** rules. The first two rules perform a single path-halving or path-splitting, and the second rule performs the splicing operation described in Rem's algorithm. We note that combining the FindCompress option with the SpliceAtomic rule results in an incorrect algorithm, and so we exclude this single combination. All variants of Rem's algorithms which do not perform SpliceAtomic are linearizable for a set of concurrent union and find operations by simply following the linearizably proof of Jayanti and Tarjan [58]. Rem's algorithms combined with SpliceAtomic do not satisfy linearizability of both concurrent find and union operations. In Appendix B.2.2 (Theorem 3), we show that the algorithm is correct in the *phase-concurrent* setting, where union and find operations are separated by a barrier.

3.3.2 Min-Based Algorithms. Recently, Liu and Tarjan present a framework for simple concurrent connectivity algorithms based on several rules which manipulate an array of parent pointers using edges to transmit the connectivity information [67]. We note that these algorithms are not true concurrent algorithms, but really parallel algorithms designed for the Massively Parallel Computation (MPC) setting. We implement the framework proposed in their paper as part of CONNECTIT, and also include algorithms that can be generated by their framework but were not considered in their paper. Their framework ensures that the parent array is a *minimum labeling*, where each vertex's parent is the minimum value among candidates that it has observed.

Conceptually, each round of an algorithm in the framework processes all remaining edges and performs several rules on each edge. In each round, each vertex observes a number of candidates and updates its parent at the end of the round to the minimum of its current parent, and all candidates. Each round performs a *connect phase*, a *shortcut phase*, and possibly an *alter phase*. The *connect phase* updates the parents of edges based on different operations. A Connect uses the endpoints of an edge as candidates for both vertices. A ParentConnect uses the parents of the endpoints of an edge as candidates for both vertices. Lastly, an ExtendedConnect uses the parent values of an edge as candidates for both the edge endpoints as well as the parents of the endpoints. Within the connect phase, the operation can also choose to update the parent value of a vertex if and only if it is a tree-root at the start of the round, which we refer to as RootUp. In all cases, an update occurs if and only if the candidate parent is smaller than the current parent. Next, the algorithms performs the *shortcut phase*, which performs one step of path compression. Some of the algorithms perform a FullShortcut, which repeats shortcutting until no further parents change. The algorithm can then execute an optional *alter phase*. Alter updates the endpoints of an edge to be the current labels of the endpoints (this step is required for correctness when using Connect).

Liu and Tarjan prove that all of the algorithm combinations generated from their framework are correct, but only analyze five particular algorithms in terms of their parallel round complexity. In addition to the five original algorithms considered by Liu and Tarjan, we consider a number of algorithm combinations that were not explored in the original paper. In this paper, we evaluate all algorithm combinations that are expressible in the Liu-Tarjan framework, which we list in Appendix D. Note that only the RootUp-based algorithms in the Liu-Tarjan framework are monotonic. The remaining algorithms are not monotonic due to the fact that a non-root vertex can be moved to a completely different subtree (one where the previous tree and new trees are disconnected). The non-monotonic algorithms result in a correct connectivity algorithm due to the fact that edges which were previously applied continue to be applied in subsequent rounds of the algorithm [67].

Other Min-Based Algorithms. Stergiou et al. [101] recently proposed a min-based connectivity algorithm for the massively parallel computation setting. We implement the algorithm as part of the Liu-Tarjan framework, within which it can be viewed as a particular instantiation of the Liu-Tarjan rules [67]. CONNECTIT also includes the folklore Label-Propagation algorithm.

Sampling for Min-Based Algorithms. Lastly, we describe how to combine the min-based algorithms above which are *not linearizably-monotonic* with the sampling algorithms in CONNECTIT. Recall that the class of min-based algorithms are based on connecting vertices using the min operator. Therefore, if the largest component after sampling, which has ID *frequentid*, is relabeled such that all vertices in this component have the smallest possible ID, then these vertices will never change components, and we thus we never have to inspect edges oriented out of these vertices. We show that this modification produces correct composed algorithms by viewing the largest component as a single contracted vertex that only preserves its inter-cluster edges, and then applying the correctness proof for a connectivity algorithm in the Liu-Tarjan framework [67]. We provide a detailed proof of correctness in Theorem 4 in the Appendix B.3.3.

3.4 Spanning Forest

We also extend CONNECTIT to generate a spanning forest of the graph. We show that the class of root-based algorithms can be converted in a black-box manner from parallel connectivity algorithms to parallel spanning forest algorithms. This class consists of every finish algorithm discussed in Section 3.3, with the exception of the Liu-Tarjan algorithms that do not use RootUp and Stergiou's algorithm. We defer our description to Appendix B.4.

3.5 Streaming

We now discuss how CONNECTIT can be extended to support streaming graph connectivity in the parallel batch-incremental and wait-free asynchronous settings. In the parallel batch-incremental setting, we consider an algorithm that receives batches of operations containing a mix of updates and queries, one batch after the other. The updates within a batch are not ordered, and so the streaming algorithm can exploit parallelism to answer both the updates and the queries within the batch. We also show how a large subset of the algorithms

proposed in CONNECTIT can be extended to work in the stronger wait-free asynchronous setting where finds can occur concurrently with union operations. CONNECTIT supports the following types of algorithms in the streaming setting:

(i) The union-find algorithms in Section 3.3.1, excluding Rem's algorithms with the SpliceAtomic method. All of these algorithms are linearizable and wait-free for a batch of concurrent union and find operations.

(ii) Shiloach-Vishkin and the root-based Liu-Tarjan algorithms. For these algorithms, we guarantee that the find operations are wait-free and linearizable, although updates are processed synchronously.

(iii) Union-Rem-CAS and Union-Rem-Lock using SpliceAtomic. For this class of algorithms, we guarantee that the results are correct when the union and find operations are run *phase-concurrently*. If a batch of mixed queries and updates are given, the algorithm first separates the updates and queries, applies the updates, and then runs the queries in the batch.

Due to space constraints, we provide details about our streaming algorithm in CONNECTIT in Appendix B.

3.6 Implementation

CONNECTIT Implementation. Our implementation of CONNECTIT is written in C++, and uses template specialization to generate high-performance implementations while ensuring that the framework code is high-level and general. Using CONNECTIT, we can instantiate any of the supported connectivity algorithm combinations using one line of code. Implementing a new sampling algorithm is done by creating a new structure that implements the sampling method for connectivity (and if applicable, a specialized implementation for spanning forest). The sampling code emits an array containing the partial connectivity information. Additionally, for spanning forest the code emits a subset of the spanning forest edges corresponding to the partial connectivity information. Implementing a new finish algorithm is done by implementing a structure providing a FINISHCOMPONENTS method. If the finish algorithm supports spanning forest, it also implements a FINISHFOREST method. Finally, if it supports streaming, the structure implements a PROCESSBATCH method, taking a batch of updates, and returning results for the queries in the batch.

Graph Compression. We use the compression techniques provided by Ligra+ [31, 96] to process the large graphs used in our experiments. Storing the largest graph used in our experiments in the uncompressed format would require well over 900GB of space to store the edges alone. However, the graph requires only 330GB when encoded using byte codes. In this format, we store a vertex's neighbor list by difference encoding consecutive vertices, with the first vertex difference-encoded with respect to the source vertex. The differences are stored using a variable-length code called a byte code. Decoding is done by sequentially uncompressing each difference, and summing the differences into a running sum which gives the ID of the next neighbor. To enable parallelism, we use a blocked version of this format, which splits a vertex's adjacency information into independent blocks, so that different blocks can be decoded in parallel. These compression schemes are supported by the Graph Based Benchmark Suite [31], and we used this code base as the basis for implementing CONNECTIT.

Dataset	n	m	Diam.	Num. Comps.	Largest Comp.
road_usa	23.9M	57.7M	6,809	1	23.9M
LiveJournal	4.8M	85.7M	16	1,876	4.8M
com-Orkut	3.1M	234.4M	9	1	3.1M
Twitter	41.7M	2.4B	23*	1	41.7M
Friendster	65.6M	3.6B	32	1	65.6M
ClueWeb	978.4M	74.7B	132*	23,794,336	950.5M
Hyperlink2014	1.7B	124.1B	207*	129,441,050	1.57B
Hyperlink2012	3.6B	225.8B	331*	144,628,744	3.35B

Table 2: Graph inputs, including vertices and edges, graph diameter, the exact number of components, and the number of vertices in the largest component. We mark diameter values where we are unable to calculate the exact diameter with * and report the effective diameter observed during our experiments, which is a lower bound on the actual diameter.

4 EVALUATION

Overview of Results. We show the following experimental results in this section.

- Without sampling, the Union-Rem-CAS algorithm using the options {SplitAtomicOne, HalveAtomicOne} is the fastest CONNECTIT algorithm across all graphs (Section 4.1).
- Performance analysis of the union-find variants (Section 4.1.1).
- With sampling, the Union-Rem-CAS algorithm using the options {SplitAtomicOne, HalveAtomicOne} is consistently the fastest algorithm in CONNECTIT (Section 4.2).
- The fastest CONNECTIT algorithms using sampling significantly outperform existing state-of-the-art results (Section 4.3).
- CONNECTIT streaming algorithms achieve throughputs between 108M–7.16B edge insertions per second across all inputs. Our algorithms achieve high throughput even at small batch sizes, and have consistent latency (Section 4.4).
- CONNECTIT's streaming algorithms outperform the streaming connected components algorithm from STINGER, an existing state-of-the-art graph streaming system, by between 1,461–28,364x.

We show the additional experimental results in the appendix:

- An analysis of the three sampling schemes considered in this paper, showing how these schemes behave in practice as a function of their parameters, and different implementation choices. One of our main observations is that in practice, the number of inter-cluster edges left after k-out Sampling is significantly less than n/k (Appendix C.4).
- We find that CONNECTIT algorithms with sampling are as fast (or faster) than basic graph routines that perform one indirect read over every edge.
- We compare CONNECTIT's performance on very large graphs (Table 1), to the performance of state-of-the-art external-memory, distributed-memory, and shared-memory systems.

Experimental Setup. Our experiments are performed on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use a work-stealing scheduler that we implemented. The scheduler is implemented similarly to Cilk for parallelism. Our programs are compiled with the `g++` compiler (version 7.3.0) with the `-O3` flag.

We use the command `numactl -i all` to balance the memory allocations across the sockets. All of the numbers we report are based on our parallel implementations on 72 cores with hyper-threading.

Graph Data. To show how ConnectIt performs on graphs at different scales, we selected a representative set of real-world graphs of varying sizes. Most of our graphs are Web graphs and social networks—low-diameter graphs that are frequently used in practice. To test our algorithms on high-diameter graphs, we also ran our implementations on a road network.

Table 2 lists the graphs we use. We used a collection of graphs at different scales, including the largest publicly-available graphs. ***road_usa*** is an undirected road network from the DIMACS challenge [30]. ***LiveJournal*** is a directed graph of the LiveJournal social network [17]. ***com-Orkut*** is an undirected graph of the Orkut social network. ***Twitter*** is a directed graph of the Twitter network, where edges represent the follower relationship [66]. ***Friendster*** is an undirected graph describing friendships from a gaming network. ***ClueWeb*** is a directed Web graph from the Lemur project at CMU [17]. ***Hyperlink2012*** and ***Hyperlink2014*** are directed hyperlink graphs obtained from the WebDataCommons dataset where nodes represent web pages [74]. Others smaller graphs we consider also are derived from different application domains, and can be found at SuiteSparse [29]. We note that Hyperlink2012 is the *largest publicly-available real-world graph*.

Some of the inputs (such as the Hyperlink graphs) are originally directed. Like previous work on connectivity for these graphs [101], we symmetrize them before applying our algorithms and empirically find that this results in a single massive component for all graphs that we consider (the size of the largest component is shown in Table 2).

4.1 Performance Without Sampling

We start by studying the performance of different ConnectIt finish methods without sampling, since trends observed in this setting hold in the setting when sampling is applied. Table 3 shows the results of the fastest ConnectIt implementations across our inputs. For algorithms, that have many options, such as union-find, we report the fastest time out of all combinations of options.

Among all of our algorithms, we observe that Union-Rem-CAS is consistently the fastest finish algorithm. We found that the fastest variant of Union-Rem-CAS across all graphs uses the FindNaive option for find (i.e., it does not perform any additional compression after the union operation), and most frequently uses the SplitAtomicOne option to perform path compression during the union operation(see Algorithm 14). We also observed that the running times of the other splice options, SpliceAtomic and HalveAtomicOne, are almost identical to SplitAtomicOne.

The Union-Hooks algorithm also achieves high performance, and is between 1.01–1.51x slower than Union-Rem-CAS across all graphs (1.22x slower on average). The Union-Async algorithm also achieves consistent high performance, and is between 1.06–1.61x slower than Union-Rem-CAS across all graphs (1.41x slower on average). Union-JTB is consistently slower than Union-Rem-CAS, being between 1.83–4.86x slower across all graphs, and 3.68x slower on average. The fastest find instantiation for Union-JTB in this setting was always the two-try splitting option, with the exception of the road_usa graph where simple finds were slightly faster. We discuss the source of performance differences between these algorithms below, in Section 4.1.1.

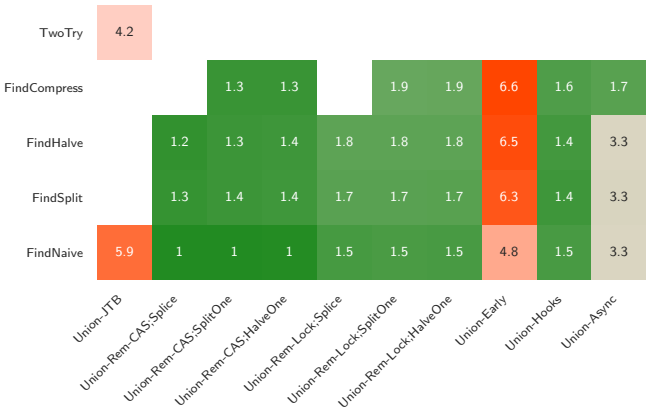


Figure 3: Relative performance of different union-find implementations on graphs used in our evaluation of ConnectIt in the *No Sampling* setting. Running times are obtained on a 72-core machine with 2-way hyper-threading, and numbers in the figure are slowdowns relative to the fastest implementation.

Compared to union-find algorithms, the Liu-Tarjan algorithms are much slower on our inputs. The fastest variants in this setting were one of {EF, PRF, PR, CRFA} (we define these Liu-Tarjan variants in Appendix D). The fastest Liu-Tarjan variant is still 1.96–8.37x slower than Union-Rem-CAS (5.23x slower on average). Stergiou's algorithm was always slower than the fastest variant from the Liu-Tarjan framework. Finally, our implementation of Shiloach-Vishkin is between 2.15–6.24x slower than Union-Rem-CAS (4.11x slower on average). The performance of Label-Propagation is between 2.41–3.36x slower on all graphs except for road_usa. On road_usa, its performance is 355x worse than that of Union-Rem-CAS because it requires a large number of rounds where most vertices are active due to the high diameter of the graph.

Takeaways. Without sampling, the Union-Rem-CAS algorithm using SplitAtomicOne, with no additional path-compression option is a robust algorithm choice, and consistently performs the best across all graphs compared with other ConnectIt implementations.

4.1.1 Union-Find Evaluation. Figure 3 shows the relative performance of different union-find variants from ConnectIt in the *No Sampling* setting, averaged across all graphs. We observe that Union-Rem-CAS implementations achieve consistently high performance in this setting, especially when using either the SplitAtomicOne or HalveAtomicOne options. The Splice option achieves good performance, but appears to be between 1.5–1.9x slower on average than using SplitAtomicOne or HalveAtomicOne to perform compression. The Union-Rem-Lock implementation is 1.4–1.8x slower than Union-Rem-CAS for all 6 variants of this algorithm. Union-Early, Union-Hooks, and Union-Async are all slower on average: between 3.8–5.6x, 1.2–1.3x, and 1.4–2.6x on average, respectively. Finally, Union-JTB is much slower, although the FindTwoTry option only incurs a (perhaps) more modest slowdown of 3.5x on average.

Performance Analysis. We annotated our experiments on union-find algorithms to measure the ***Max Path Length (MPL)***, or the

Group	Algorithm	road_usa	LiveJournal	com-Orkut	Twitter	Friendster	ClueWeb	Hyperlink2014	Hyperlink2012
No Sampling	Union-Early	3.61e-2	3.48e-2	8.63e-2	2.52	1.50	59.8	17.0	32.9
	Union-Hooks	3.37e-2	1.75e-2	2.69e-2	0.390	1.17	6.05	9.37	20.0
	Union-Async	4.02e-2	2.03e-2	3.12e-2	0.426	1.21	7.92	12.2	25.5
	Union-Rem-CAS	**2.80e-2**	1.27e-2	1.91e-2	0.316	0.902	4.04	6.64	13.9
	Union-Rem-Lock	5.07e-2	1.95e-2	2.84e-2	0.437	1.23	5.64	9.20	19.3
	Union-JTB	6.90e-2	4.49e-2	8.48e-2	0.965	2.76	22.5	36.4	72.1
	Liu-Tarjan	7.40e-2	5.18e-2	6.46e-2	2.78	6.60	30.1	67.1	142
	Shiloach-Vishkin	0.138	4.34e-2	5.70e-2	1.65	5.38	21.2	38.5	106
	Label-Propagation	13.4	4.66e-2	6.37e-2	1.24	4.37	13.4	20.7	46.5
k-out Sampling	Union-Early	3.25e-2	9.00e-3	8.61e-3	0.117	0.227	2.28	4.77	8.94
	Union-Hooks	3.62e-2	9.18e-3	9.16e-3	0.121	0.230	2.22	3.63	8.51
	Union-Async	3.33e-2	8.97e-3	**8.56e-3**	0.117	0.228	2.21	3.60	8.49
	Union-Rem-CAS	3.43e-2	**8.96e-3**	8.62e-3	0.117	0.227	2.15	3.51	**8.20**
	Union-Rem-Lock	4.45e-2	1.13e-2	1.01e-2	0.138	0.344	2.63	4.33	9.91
	Union-JTB	3.89e-2	9.77e-3	8.80e-3	0.125	0.237	2.43	4.05	9.58
	Liu-Tarjan	6.34e-2	9.90e-3	9.18e-3	0.129	0.374	2.61	6.74	11.5
	Shiloach-Vishkin	5.72e-2	9.72e-3	8.78e-2	0.124	0.237	2.70	5.03	12.5
	Label-Propagation	12.6	1.02e-2	9.63e-3	0.121	0.375	2.44	4.75	9.68
BFS Sampling	Union-Early	2.69	1.07e-2	9.26e-3	9.42e-2	0.186	2.27	4.02	9.33
	Union-Hooks	2.65	1.09e-2	9.71e-3	9.53e-2	0.186	2.29	2.94	9.40
	Union-Async	2.69	1.08e-2	9.12e-3	9.31e-2	0.189	2.23	2.87	9.23
	Union-Rem-CAS	2.66	1.06e-2	9.19e-3	**9.24e-2**	**0.183**	2.21	**2.83**	9.11
	Union-Rem-Lock	2.67	1.13e-2	1.07e-2	0.113	0.219	2.69	3.68	10.8
	Union-JTB	2.75	1.14e-2	9.52e-3	9.80e-2	0.195	2.38	3.22	9.88
	Liu-Tarjan	2.68	1.17e-2	9.80e-3	9.61e-2	0.383	2.85	7.61	13.4
	Shiloach-Vishkin	2.54	1.12e-2	9.72e-3	9.87e-2	0.196	2.59	4.13	12.2
	Label-Propagation	2.58	1.19e-2	1.03e-2	9.47e-2	0.446	2.31	3.21	9.91
LDD Sampling	Union-Early	0.117	1.32e-2	8.63e-3	0.124	0.193	1.74	4.63	8.52
	Union-Hooks	0.112	1.33e-2	8.81e-3	0.127	0.197	1.75	3.58	8.46
	Union-Async	0.103	1.32e-2	8.49e-3	0.123	0.193	1.71	3.48	8.31
	Union-Rem-CAS	9.86e-2	1.29e-2	8.48e-3	0.122	0.193	**1.69**	3.46	8.28
	Union-Rem-Lock	0.126	1.54e-2	1.03e-2	0.144	0.226	2.16	4.31	9.97
	Union-JTB	0.148	1.35e-2	8.98e-3	0.131	0.202	1.85	3.84	9.13
	Liu-Tarjan	0.178	1.45e-2	8.73e-3	0.130	1.24	2.32	8.33	12.5
	Shiloach-Vishkin	0.250	1.36e-2	8.81e-3	0.131	0.197	2.07	4.70	11.2
	Label-Propagation	14.3	1.41e-2	8.99e-3	0.127	2.03	1.76	3.79	9.06
Other Systems	BFSCC [92]	2.60	1.94e-2	1.05e-2	0.169	1.34	5.56	61.6	62.5
	WorkefficientCC [94]	0.41	0.247	2.78e-2	0.587	2.18	5.97	11.4	25.8
	MultiStep [98]	29.6	0.272	0.138	x	1.76	—	—	—
	Galois [79]	6.10e-2	2.55e-2	3.40e-2	1.167	1.77	—	—	—
	PatwaryRM [84]	6.81e-2	3.65e-2	3.93e-2	0.428	1.15	—	—	—
	GAPBS (Shiloach-Vishkin) [11]	0.103	0.134	0.150	5.669	7.01	—	—	—
	GAPBS (Afforest) [104]	4.29e-2	5.30e-2	7.32e-2	0.172	0.306	—	—	—

Table 3: Running times of CONNECTIT algorithms and state-of-the-art static connectivity algorithms in seconds on a 72-core machine (with 2-way hyper-threading enabled). We report running times for the *No Sampling* setting, as well as the *k-out Sampling*, *BFS Sampling*, and *LDD Sampling* schemes considered in this paper. Within each group, we display the running time of the fastest CONNECTIT variant in green. Additionally, for each graph we display the fastest running time achieved by any algorithm combination in bold. We mark entries that did not successfully solve the problem with a x, and mark entries that are inapplicable with —.

longest path length experienced by the union-find algorithm during the execution of any UNION operation, and the ***Total Path Length (TPL)***, which is just the sum of all path lengths observed during all UNION executions. We also measured the number of LLC misses, and the total amount of bytes transferred to the memory controller (both reads and writes). We note that adding this instrumentation affects the overall running times between 10–20%. We report the detailed results of the analysis in Appendix C.1. We find that the TPL is highly relevant for predicting the running time—the TPL has a Pearson correlation coefficient of 0.738 with running time (the MPL has a weaker coefficient of 0.344).

Takeaways. Based on this discussion, we conclude that minimizing both the total amount of data written to and read from the memory controller, and improving the locality of these accesses thereby reducing the number of LLC misses is critical for high performance. One way of achieving these objectives is by minimizing the TPL traversed, although optimizing to minimize the TPL does not by itself guarantee the fastest performance.

4.2 Performance With Sampling

We defer a detailed analysis of our performance under different sampling schemes to Appendix C.2, and list our main findings here. Our results for combining CONNECTIT algorithms with different sampling schemes suggest the following takeaways when selecting algorithms and sampling schemes for different graphs:

- For low-diameter graphs, such as social networks and Web graphs, the fastest finish algorithms combined with all three sampling schemes explored in this paper provide significant speedups over the fastest CONNECTIT algorithms that do not use sampling, ranging from 1.4–3.9x speedup using *k*-out Sampling (2.2x on average), 1.1–4.9x speedup using BFS Sampling (2.4x on average), and 0.98–4.6x speedup for LDD Sampling (2.3x on average).
- If the graph has high diameter, using *k*-out Sampling with any union-find method seems to be the best fit, unless using Label-Propagation, in which case BFS Sampling is the best choice.
- For large Web graphs, like ClueWeb and the two Hyperlink graphs studied in this paper, all three sampling schemes result in very high speedups over the fastest unsampled algorithms in CONNECTIT,

ranging from 1.7–1.9x for k-out Sampling, 1.5–2.3x for BFS Sampling, and 1.7–2.4x for LDD Sampling. Furthermore, each of these schemes combined with Union-Rem-CAS results in the fastest ConnectIt algorithm for one of these graphs.

4.3 Comparison with State-Of-The-Art

Table 3 reports the performance of these systems on the inputs used in our evaluation. Aside from BFSCC and WorkefficientCC, which we implemented as part of our system, we were unable to run the other systems on the large inputs due to the fact that these systems do not support compression, which is required for systems to compactly store and process our largest graph inputs on our machine without using an exorbitant amount of disk and memory. We list the main takeaways of the experimental results here, and present the full results of the comparison in Appendix C.2.1.

- **BFSCC**: The BFS-based connectivity implementation available from Ligra [92]. This algorithm computes each connected component by running a parallel BFS from the first uncovered vertex within it. We find that using sampling, ConnectIt is between 1.22–21.7x faster than BFSCC, and 6.46x faster on average.
- **WorkefficientCC**: The work-efficient connectivity implementation by Shun et al. [94], which recursively computes LDD, which is publicly-available as part of the Graph Based Benchmark Suite [31]. Our implementations without sampling are between 1.45–19.4x faster, and 5.6x faster on average. Our implementations with sampling are between 3.1–27.5x faster, and 9.3x faster on average. This algorithm previously held the record time for solving connectivity on the Hyperlink2012 graph on any system, running in just 25 seconds on a 72-core shared-memory machine. The fastest ConnectIt algorithm, Union-Rem-CAS with SplitAtomicOne breaks this record, running 3.2x faster on a similar machine.
- **Multistep**: The hybrid BFS/Label-Propagation method by Slota et al. [98]. Our codes without sampling are between 1.95–21.4x faster, and 10.1x faster on average. Using sampling, our codes are between 9.6–30.3x faster, and 18.6x faster on average. Our codes are much faster on large-diameter networks (over 1,000x faster).
- **Galois**: Galois is a state-of-the-art shared-memory parallel programming library for C++ [78]. Several connectivity implementations are available in their code base. We found that their label propagation algorithm is consistently the fastest, and report running times for this implementation. Excluding the road_usa graph where their code performs poorly, our codes without sampling are between 1.78–3.69x faster than theirs (2.36x faster on average), and our codes using sampling are between 2.84–12.6x faster than theirs (7.27x faster on average). Our codes are 2.17x faster than their topology-driven label propagation algorithm on the road_usa graph, which was their fastest algorithm for this graph.
- **PatwaryRM**: We use the multicore union-find codes by Patwary et al. [84], which use Rem's algorithm. Our fastest implementations without sampling achieve between 1.27–2.87x speedup over their implementation (1.99x speedup on average), and our fastest implementations with sampling achieve between 2.43–6.28x speedup over their implementation (4.4x speedup on average).
- **GAPBS**: We compared with codes from the GAP Benchmark Suite, a state-of-the-art shared-memory graph processing benchmark [11], which implements the Afforest algorithm [104]. Our fastest implementations without sampling are between 0.33–4.17x faster than their Afforest [104] implementation (2.08x faster on average). Our fastest implementations with sampling are between 1.53–8.55x faster than their Afforest implementation (3.9x faster on average).

4.4 Streaming Parallel Graph Connectivity

In this section, we evaluate ConnectIt algorithms that support streaming. Our evaluation consists of three parts: (i) evaluating the throughput achieved by our streaming algorithms across a range of graph sizes and classes; (ii) evaluating the latency of applying batches of different sizes, and at different ratios of insertions to queries within a batch; and (iii) comparing our streaming algorithms with existing state-of-the-art multicore streaming connectivity methods. We provide a full analysis of these topics in Appendix C.3.

Experiment Design. We start by describing the design of our experiments. We run two types of streaming experiments. The first type of experiment generates a stream of edge updates based on a static input graph. The updates are generated by sampling a fraction of the edges, f_u, to use as updates. Unless otherwise mentioned, we use all edges as updates ($f_u = 1$). The second type of experiment uses synthetic graph generators to sample edge updates. We consider the RMAT and Barabasi-Albert (BA) graph generators in these experiments. We generate RMAT graphs using $(a, b, c) = (0.5, 0.1, 0.1)$, and create a BA graph with $m = 10n$ edges. The number of vertices in both graphs is $n = 2^{30}$, and the number of edges is $10n$ in the RMAT and BA graph respectively. In both cases, the batches used by our streaming algorithms are represented in the COO format. For the ClueWeb, Hyperlink2014, and Hyperlink2012 graphs, we are unable to represent the entire graph in the COO format, and sub-sample 10% of the edges to use as updates.

4.4.1 Streaming Throughput. We first consider the maximum throughput achieved by each algorithm family in the setting where only updates are applied. Table 4 reports the streaming throughput achieved by the fastest variant of each algorithm on each of the input graphs. Note that no sampling is applied in these streaming experiments. The updates are streamed as part of a single, large batch, which is applied by the algorithm in parallel for Type (i) and (ii), and separately applied for updates first, then queries for Type (iii) algorithms (see the discussion on these types in Section 3.5).

We observe that the performance of most of our union-find algorithms is consistently high on these graphs, in particular the performance of Union-Hooks, Union-Async, Union-Rem-CAS, and Union-Rem-Lock. As in the static setting, the Union-Rem-CAS algorithm consistently performs the best on all input graphs, achieving a maximum throughput of over 7 billion updates per second on the com-Orkut graph. The other two union-find algorithms, Union-Early and Union-JTB, achieve somewhat inconsistent performance, which is consistent with our findings from Section 4.1. For Union-Early, the version of the algorithm with no extra compression performed best compared with the versions that perform additional path compression algorithms, as in the static case in Figure 3 (comparing FindNaive

Algorithm	road_usa	LiveJournal	com-Orkut	Twitter	Friendster	RMAT	BA	ClueWeb	Hyperlink2014	Hyperlink2012
Union-Early	1.48e9	9.23e8	1.38e9	4.31e8	1.05e9	3.49e8	5.16e8	4.00e8	3.15e9	2.80e9
Union-Hooks	3.12e9	4.21e9	5.94e9	2.79e9	1.49e9	7.27e8	1.18e9	4.69e9	5.17e9	4.48e9
Union-Async	3.49e9	3.36e9	5.29e9	2.73e9	1.41e9	8.05e8	1.13e9	4.86e9	5.92e9	4.69e9
Union-Rem-CAS	**3.98e9**	**5.28e9**	**7.16e9**	**3.85e9**	**2.01e9**	**8.78e8**	**1.46e9**	**5.73e9**	**6.64e9**	**5.64e9**
Union-Rem-Lock	1.56e9	3.68e9	5.95e9	3.36e9	1.74e9	7.67e8	1.42e9	3.56e9	2.99e9	3.21e9
Union-JTB	1.15e9	1.06e9	2.68e9	1.42e9	7.33e8	2.88e8	5.27e8	2.15e9	2.26e9	1.79e9
Liu-Tarjan	2.87e8	4.31e8	5.98e8	3.77e8	1.84e8	1.11e8	1.98e8	3.02e8	2.80e8	2.62e8
Shiloach-Vishkin	1.79e8	4.56e8	1.13e9	2.89e8	1.76e8	1.06e8,	2.43e8	3.34e8	2.65e8	2.24e8

Table 4: Maximum parallel streaming throughput (edge updates per second) achieved by ConnectIt streaming algorithms on our graph inputs, and two synthetic graph inputs from the RMAT and Barabasi-Albert (BA) families. Due to memory constraints, we were unable to materialize a COO representation of our three largest graphs, and instead, sample of 10% of the edges for the batch. Otherwise, the entire graph is used as part of a single batch, which is not permuted. All edges in the batch are updates. For each graph we display the algorithm with the highest throughput in bold.

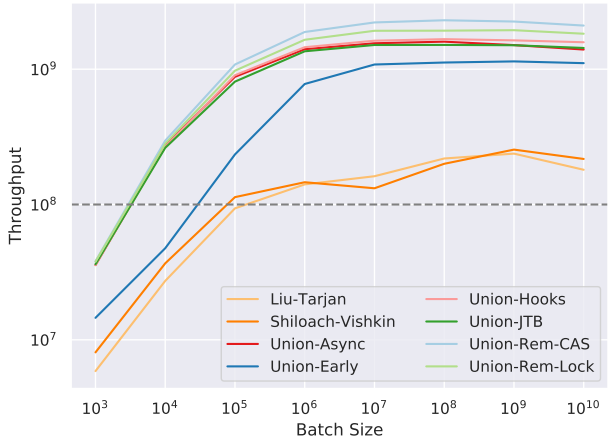


Figure 4: Throughput vs. batch size of different ConnectIt incremental connectivity implementations on the Friendster graph.

performance to that of non-trivial path compression options). The performance of Shiloach-Vishkin seems to depend mostly on the number of rounds required (more rounds on larger diameter networks like road_usa). We note that the algorithm requires just two rounds on com-Orkut. The performance of the fastest Liu-Tarjan algorithm is similar to that of the Shiloach-Vishkin algorithm in the streaming setting across all graphs. Both of these methods are significantly slower than the fastest union-find methods. We note that the fastest Liu-Tarjan algorithm in this setting is consistently the version using Connect, RootUp, FullShortcut, and Alter (the CRFA algorithm in Appendix D).

Throughput vs. Batch Size. Figure 4 shows the throughput obtained by our algorithms as a function of the batch size for the Friendster graph (plots for other graphs are in Appendix C.3. We observe that all ConnectIt algorithms achieve consistently high throughput, from very small, to very large batch sizes. Our fastest union-find algorithms achieve a throughput of over 10M updates per second at a batch size of 100, and exceed 100M updates per second for batches of size 1000 on all graphs.

4.4.2 Streaming Latency. Next, we evaluate the latency of applying batches of pure update operations. Due to space constraints, the figure of this experiment is provided in Figure 18 in Appendix C.3, and we describe our main findings here. The experiment studies the per-batch latency of different algorithm variants when processing a 10% edge sample of the Hyperlink2012 graph. We observed that the latency for all of our algorithms is highly regular, with the median time per batch for different batch sizes being within 1–2% of the average batch time across all algorithms. The per-batch latency also grows linearly with the batch-size, as expected, and the lowest per-batch latencies are achieved by the Union-Rem-CAS algorithm using either the SplitAtomicOne or HalveAtomicOne option.

4.4.3 Streaming Comparison with STINGER. STINGER is built around a dynamic data structure for representing dynamic graphs in the streaming setting [35]. The data structure is based on adapting the CSR format to support dynamic updates. The structure chunks the edges into a number of blocks, which are chained together as a linked list. Updates traverse the list to find an empty slot for a new edge, using fine-grained locking to perform concurrent insertions. STINGER supports a streaming connected components algorithm by McColl et al. [71]. Their algorithm also supports edge deletions, which makes it more general than our streaming algorithms. Since our codes do not update the graph structure, for the sake of comparing Stinger with our codes, we only report the time required for Stinger to update its connectivity labeling, which is strictly less than the overall update time of Stinger. As far as we know, the only existing parallel algorithm designed for incremental connectivity is by Simsiri et al. [97], but unfortunately we were unable to obtain the code from the authors.

Comparison. The STINGER code takes one parameter, which trades off the space usage of the algorithm vs. the amount of re-computation that has to be done upon a deletion. We set it to the lowest possible value, which gave the best performance. We note that the STINGER streaming connectivity algorithm has an unusually long initialization period, which depends on the number of vertices. We were unable to initialize the streaming connectivity algorithm within several hours for graphs with more than 1 million vertices, and were only able to evaluate batches of size up to 2 million due to limitations in their system. Based on these considerations, we opted to generate batches from an RMAT graph generator using 2^{20} vertices. In the experiment, we insert batches of varying sizes and measure the time required by the streaming connectivity algorithm to process the batch. Note that the times we report ignore the time taken by STINGER to update adjacency information.

Table 5 reports the results of the experiment of STINGER, and results for the same experiment for ConnectIt's Union-Rem-CAS algorithm with the SplitAtomicOne option. Both systems are run on the same machine using all cores with hyper-threading. We observe

Batch Size	STINGER	Updates/sec	CONNECTIT	Updates/sec
10	6.07e-2	164	2.14e-6	4.67M
10^2	9.87e-2	1013	1.19e-5	8.40M
10^3	0.171	5,847	2.19e-5	45.6M
10^4	0.137	72,992	5.19e-5	192M
10^5	0.503	198,807	3.25e-4	307M
10^6	3.99	250,626	2.73e-3	366M
$2 \cdot 10^6$	6.52	306,748	4.313e-3	463M

Table 5: Running times and update rates (directed edges/second) for STINGER's streaming connected components algorithm and CONNECTIT's Union-Rem-CAS algorithm (with SplitAtomicOne) when performing batch edge updates on an empty graph with varying batch sizes. Inserted edges are sampled from the RMAT graph generator.

that CONNECTIT significantly outperforms STINGER in this setting, achieving between *1,461–28,364x* speedup across different batch sizes. Surprisingly, CONNECTIT applied with a batch size of 10 achieves significantly higher throughput than STINGER for a batch size of 2M. We realize that our comparison is somewhat unfair toward STINGER, since although this implementation is one of the fastest streaming connectivity implementations currently publicly available, it is designed for dynamic updates, and must perform extra work in anticipation of edge deletions, which our algorithms do not handle. It would be interesting to extend CONNECTIT to support fully dynamic updates, and to revisit this comparison in that setting.

5 CONCLUSION

In conclusion, we have introduced the CONNECTIT framework, which provides orders of magnitude more parallel static and incremental connectivity and spanning forest implementations than what currently exist today. We have found that the fastest multicore implementations in CONNECTIT significantly outperform state-of-the-art parallel solutions. We believe that this paper is one of the most comprehensive evaluation of multicore parallel connectivity implementations to date. For future work, we are interested in identifying practical parallel algorithms that support edge deletions and integrating them into CONNECTIT.

ACKNOWLEDGEMENT

Thanks to Edward Fan for initial implementations of Rem's algorithm. Thanks to Guy Blelloch, Siddartha Jayanti, Jakub Lacki, and Yuanhao Wei for for helpful discussions and suggestions. This research was supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. 2019. Parallel Batch-Dynamic Graph Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 381–392.

[2] Dan Alistarh, Alexander Fedorov, and Nikita Koval. 2019. In Search of the Fastest Concurrent Union-Find Algorithm. *CoRR* abs/1911.06347 (2019). http://arxiv.org/abs/1911.06347

[3] Hazim Almuhimedi, Shomir Wilson, Bin Liu, Norman Sadeh, and Alessandro Acquisti. 2013. Tweets are forever: a large-scale quantitative analysis of deleted tweets. In *Proceedings of the 2013 conference on Computer supported cooperative work*. 897–908.

[4] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. 2018. Parallel graph connectivity in log diameter rounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 674–685.

[5] B. Awerbuch and Y. Shiloach. 1987. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. *IEEE Trans. Comput.* C-36, 10 (1987), 1258–1263.

[6] David A. Bader and Guojing Cong. 2005. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of 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 *International Conference on Parallel Processing (ICPP)*. 547–556.

[8] David A. Bader and Joseph JaJa. 1996. Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study. *J. Parallel Distrib. Comput.* 35, 2 (1996), 173–190.

[9] Dip Sankar Banerjee and Kishore Kothapalli. 2011. Hybrid Algorithms for List Ranking and Graph Connected Components. In *International Conference on High Performance Computing (HiPC)*. 1–10.

[10] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Article 12, 12:1–12:10 pages.

[11] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). http://arxiv.org/abs/1508.03619

[12] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab Mirrokni. 2019. Near-optimal massively parallel graph connectivity. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1615–1636.

[13] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab Mirrokni, and Warren Schudy. 2019. Massively Parallel Computation via Remote Memory Access. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*. ACM, 59–68.

[14] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 235–248.

[15] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 253–264.

[16] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally Deterministic Parallel Algorithms Can Be Fast. In *ACM SIGPLAN Symposium on Proceedings of Principles and Practice of Parallel Programming (PPoPP)*. 181–192.

[17] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*.

[18] Libor Bus and Pavel Tvrdik. 2001. A Parallel Algorithm for Connected Components on Distributed Memory Machines. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. 280–287.

[19] E.N. Caceres, F. Dehne, H. Mongelli, S.W. Song, and J.L. Szwarcfiter. 2004. A Coarse-Grained Parallel Algorithm for Spanning Tree and Connected Components. In *Euro-Par*.

[20] Francis Y. Chin, John Lam, and I-Ngo Chen. 1982. Efficient Parallel Algorithms for Some Graph Problems. *Commun. ACM* 25, 9 (Sept. 1982), 659–665.

[21] Laukik Chitnis, Anish Das Sarma, Ashwin Machanavajjhala, and Vibhor Rastogi. 2013. Finding Connected Components in Map-reduce in Logarithmic Rounds. In *IEEE International Conference on Data Engineering (ICDE)*. 50–61.

[22] K.W. Chong and T.W. Lam. 1995. Finding Connected Components in $O(\log n \log\log n)$ Time on the EREW PRAM. *Journal of Algorithms* 18, 3 (1995), 378–402.

[23] 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.

[24] Guojing Cong and Paul Muzio. 2014. Fast Parallel Connected Components Algorithms on GPUs. In *Revised Selected Papers, Part I, of the Euro-Par 2014 International Workshops on Parallel Processing - Volume 8805*. 153–164.

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3. ed.)*. MIT Press.

[26] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *USENIX Conference on File and Storage Technologies (FAST)*. 45–58.

[27] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *PLDI*. ACM, 752–768.

[28] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 752–768.

[29] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (Nov. 2011), 1:1–1:25.

[30] Camil Demetrescu, Andrew Goldberg, and David Johnson. 2019. 9th DIMACS implementation challenge - shortest paths. http://www.dis.uniroma1.it/challenge9/.

[31] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 393–404.

[32] Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. 2020. Parallel Batch-Dynamic Graphs: Algorithms and Lower Bounds. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1300–1319.

[33] Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *PVLDB* 13, 9 (2020), 1598–1613.

[34] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC)*. 1–5.

[35] David Ediger, Robert McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC)*. 1–5.

[36] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *International Conference on Knowledge Discovery and Data Mining (KDD)*. 226–231.

[37] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1233âĂŞ1244.

[38] Xing Feng, Lijun Chang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Long Yuan. 2018. Distributed computing connected components with linear communication cost. *Distributed and Parallel Databases* 36, 3 (01 Sep 2018), 555–592.

[39] Junhao Gan and Yufei Tao. 2015. DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation. In *ACM International Conference on Management of Data (SIGMOD)*. 519–530.

[40] Hillel Gazit. 1991. An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph. *SIAM J. Comput.* 20, 6 (Dec. 1991), 1046–1067.

[41] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *PVLDB* 13, 8 (2020), 1304–13.

[42] Giraph 2012. Giraph. http://giraph.apache.org.

[43] John Greiner. 1994. A Comparison of Parallel Algorithms for Connected Components. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 16–25.

[44] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 24–34.

[45] Shay Halperin and Uri Zwick. 1994. An Optimal Randomized Logarithmic Time Connectivity algorithm for the EREW PRAM (Extended Abstract). In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[46] S. Hambrusch and L. TeWinkel. 1988. A study of connected component labeling algorithms on the MPP. In *International Conference on Supercomputing (ICS)*. 477–483.

[47] Yujie Han and Robert A. Wagner. 1990. An Efficient and Fast Parallel-connected Component Algorithm. *J. ACM* 37, 3 (July 1990), 626–642.

[48] K. A. Hawick, A. Leist, and D. P. Playne. 2010. Parallel Graph Component Labelling with GPUs and CUDA. *Parallel Comput.* 36, 12 (Dec. 2010), 655–678.

[49] Maurice P. Herlihy and Jeanette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

[50] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. 1979. Computing Connected Components on Parallel Computers. *Commun. ACM* 22, 8 (Aug. 1979), 461–464.

[51] Jacob Holm, Valerie King, Mikkel Thorup, Or Zamir, and Uri Zwick. 2017. Random k-out subgraph leaves only $O(n/k)$ inter-component edges. *CoRR* abs/1909.11147 (2017). http://arxiv.org/abs/1909.11147

[52] Tsan-Sheng Hsu, Vijaya Ramachandran, and Nathaniel Dean. 1997. Parallel Implementation of Algorithms for Finding Connected Components in Graphs. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge*. 23–41.

[53] J. Iverson, C. Kamath, and G. Karypis. 2015. Evaluation of Connected-component Labeling Algorithms for Distributed-memory Systems. *Parallel Comput.* 44, C (May 2015), 53–68.

[54] Kazuo Iwama and Yahiko Kambayashi. 1994. A Simpler Parallel Algorithm for Graph Connectivity. *J. Algorithms* 16, 2 (March 1994), 190–217.

[55] Jayadharini Jaiganesh and Martin Burtscher. 2018. A High-performance Connected Components Implementation for GPUs. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 92–104.

[56] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru. 2017. An Adaptive Parallel Algorithm for Computing Connected Components. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2428–2439.

[57] J. Jaja. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[58] Siddhartha V. Jayanti and Robert E. Tarjan. 2016. A Randomized Concurrent Algorithm for Disjoint Set Union. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 75–82.

[59] Siddhartha V. Jayanti, Robert E. Tarjan, and Enric Boix-Adserà. 2019. Randomized Concurrent Set Union and Generalized Wake-Up. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 187–196.

[60] 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.

[61] David R. Karger, Noam Nisan, and Michal Parnas. 1999. Fast Connected Components Algorithms for the EREW PRAM. *SIAM J. Comput.* 28, 3 (Feb. 1999), 1021–1034.

[62] 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 (SOCC)*. Article 18, 18:1–18:13 pages.

[63] Vaclav Koubek and Jana Krsnakova. 1985. Parallel algorithms for connected components in a graph. In *Fundamentals of Computation Theory*. 208–217.

[64] Arvind Krishnamurthy, Steven S. Lumetta, David E. Culler, and Katherine Yelick. 1994. Connected Components on Distributed Memory Machines. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge*. 1–21.

[65] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. 1990. Efficient parallel algorithms for graph problems. *Algorithmica* 5, 1-4 (1990), 43–64.

[66] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *WWW*.

[67] Sixue Liu and Robert E. Tarjan. 2019. Simple Concurrent Labeling Algorithms for Connected Components. In *Symposium on Simplicity in Algorithms (SOSA)*. 3:1–3:20.

[68] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *European Conference on Computer Systems (EuroSys)*. 527–543.

[69] Kamesh Madduri and David A. Bader. 2009. Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1–11.

[70] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *ACM Conference on Management of Data (SIGMOD)*. 135–146.

[71] R. McColl, O. Green, and D. A. Bader. 2013. A new parallel algorithm for connected components in dynamic graphs. In *IEEE International Conference on High Performance Computing (HiPC)*. 246–255.

[72] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 201–213.

[73] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The Graph Structure in the Web–Analyzed on Different Aggregation Levels. *The Journal of Web Science* 1, 1 (2015), 33–47.

[74] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The Graph Structure in the Web–Analyzed on Different Aggregation Levels. *The Journal of Web Science* 1, 1 (2015).

[75] Gary L. Miller, Richard Peng, and Shen Chen Xu. 2013. Parallel Graph Decomposition Using Random Shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 196–203.

[76] Dhruva Nath and S. N. Maheshwari. 1982. Parallel Algorithms for the Connected Components and Minimal Spanning Tree Problems. *Inf. Process. Lett.* 14, 1 (1982), 7–11.

[77] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*. 456–471.

[78] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*. 456–471.

[79] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, Portable and Parameterless. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 499–512.

[80] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1–19.

[81] M.M.A. Patwary, P. Refsnes, and F. Manne. 2012. Multi-core Spanning Forest Algorithms using the Disjoint-set Data Structure. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 827–835.

[82] Md. Mostofa Ali Patwary, Jean Blair, and Fredrik Manne. 2010. Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure. In *International Symposium on Experimental Algorithms (SEA)*. 411–423.

[83] M. M. A. Patwary, D. Palsetia, A. Agrawal, W. k. Liao, F. Manne, and A. Choudhary. 2012. A new scalable parallel DBSCAN algorithm using the disjoint-set data structure. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 62:1–62:11.

[84] Md. Mostofa Ali Patwary, Peder Refsnes, and Fredrik Manne. 2012. Multi-core Spanning Forest Algorithms using the Disjoint-set Data Structure. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 827–835.

[85] C. A. Phillips. 1989. Parallel Graph Contraction. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 148–157.

[86] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. Das Sarma. 2013. Finding connected components in map-reduce in logarithmic rounds. In *IEEE International Conference on Data Engineering (ICDE)*. 50–61.

[87] J. H. Reif. 1985. Optimal Parallel Algorithms for Integer Sorting and Graph Connectivity. *TR-08-85, Harvard University* (1985).

[88] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[89] Dipanjan Sengupta and Shuaiwen Leon Song. 2017. EvoGraph: On-the-Fly Efficient Mining of Evolving Graphs on GPU. In *High Performance Computing*. 97–119.

[90] Yossi Shiloach and Uzi Vishkin. 1982. An $O(\log n)$ Parallel Connectivity Algorithm. *J. Algorithms* 3, 1 (1982), 57–67.

[91] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 135–146.

[92] J. Shun and G. E. Blelloch. 2014. Phase-Concurrent Hash Tables for Determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 96–107.

[93] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing Contention Through Priority Updates. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 152–163.

[94] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2014. A Simple and Practical Linear-Work Parallel Algorithm for Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 143–153.

[95] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2014. A Simple and Practical Linear-work Parallel Algorithm for Connectivity. In *SPAA*.

[96] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *IEEE Data Compression Conference (DCC)*. 403–412.

[97] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and KunâĂŘLung Wu. 2017. Work-efficient parallel union-find. *Concurrency and Computation: Practice and Experience* 30, 4 (2017).

[98] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 550–559.

[99] G. M. Slota, S. Rajamanickam, and K. Madduri. 2016. A Case Study of Complex Graph Analysis in Distributed Memory: Implementation and Optimization. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 293–302.

[100] J. Soman, K. Kishore, and P. J. Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1–8.

[101] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting Label Propagation for Distributed Connected Components. In *ACM International Conference on Web Search and Data Mining (WSDM)*. 540–546.

[102] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *PVLDB* 13, 2 (2019), 211–225.

[103] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. 2018. PAM: Parallel Augmented Maps. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 290–304.

[104] M. Sutton, T. Ben-Nun, and A. Barak. 2018. Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 12–21.

[105] Robert E. Tarjan and Uzi Vishkin. 1985. An Efficient Parallel Biconnectivity Algorithm. *SIAM J. Comput.* 14, 4 (1985), 862–874.

[106] Uzi Vishkin. 1984. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics* 9, 2 (1984), 197–207.

[107] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[108] Yiqiu Wang, Yan Gu, and Julian Shun. 2020. Theoretically-Efficient and Practical Parallel DBSCAN. In *ACM SIGMOD*.

[109] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.* 4, 1, Article 3 (Aug. 2017), 3:1–3:49 pages.

[110] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Efficient structural graph clustering: an index-based approach. *The VLDB Journal* 28, 3 (2019), 377–399.

[111] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 824–833.

[112] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. *Proc. VLDB Endow.* 7, 14 (Oct. 2014), 1821âĂŞ1832.

[113] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. 2019. FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence. *CoRR* abs/1910.05971 (2019). http://arxiv.org/abs/1910.05971

A APPENDIX OVERVIEW

The appendix is structured as follows:

(1) Section B contains deferred content and proofs about the ConnectIt framework. Section B.2 contains additional content about our finish algorithms, including our implementation of Shiloach-Vishkin, Stergiou, and Label-Propagation. Section B.4 describes the ConnectIt framework for spanning forest. Finally, Section B.5 describes the ConnectIt framework for streaming in the batch-incremental and wait-free asynchronous settings.
(2) Section C contains deferred experimental results. Specifically, we present the following results:
   (i) Results about ConnectIt algorithms in the setting *without sampling* (Section C.1). This includes a detailed performance characterization of our union-find and Liu-Tarjan algorithms based on performance counters, and other metrics.
   (ii) Results about ConnectIt algorithms in the setting *with sampling* (Section C.1).
   (iii) Additional details about the experimental comparisons to state-of-the-art algorithms done in this paper (Section C.2.1).
   (iv) A detailed experimental evaluation of the sampling schemes considered in this paper, analyzing how adjusting their parameters affects their running time, and the sampling quality (Section C.4).
   (v) A comparison between ConnectIt algorithms, and basic graph processing primitives that map, and perform indirect reads across all edges (Section C.4.1).
   (vi) A comparison between ConnectIt algorithms on very large graphs, and existing state-of-the-art results for such large graphs (Section C.5).
(3) For completeness, we provide pseudocode for our algorithms (Section D).

Additional Primitives. Before starting, we define an additional primitive used in the following sections: a ***writeMin*** takes two arguments, a memory location x and a value *val*. If *val* is less than the value stored at x, then ***writeMin*** atomically updates the value at x to *val* and returns *true*; otherwise it returns *false*. A ***writeMin*** can be implemented with a loop that attempts a CAS as long as *val* is less than the value at x [93].

B FRAMEWORK

In this section we provide deferred correctness proofs, as well as descriptions of algorithms deferred from the main body of the paper.

B.1 Correctness for Sampling Algorithms

Theorem 2. *k-out Sampling, BFS Sampling, and LDD Sampling all produce connectivity labeling satisfying Definition 3.1.*

Proof. Consider requirement (1) first. It is easy to check that both BFS Sampling and LDD Sampling satisfy this requirement, since both methods produce trees rooted either at the source for BFS Sampling, or at cluster centers in LDD Sampling. For k-out Sampling, this requirement is satisfied since this sampling method only hooks roots, and since the last step of the sampling method performs full path compression. Therefore, the labeling output by k-out Sampling consists exactly of roots, and vertices mapped to roots as required.

Now, consider requirement (2). Observe that this requirement is equivalent to the fact that the labeling C emitted by the sampling method is a valid partial connectivity labeling (i.e., if two vertices $u, v \in V$ are in the same tree in C, then they are connected in G). BFS Sampling clearly satisfies this requirement since the labeling represents a single connected component which is clearly a valid partial labeling. Similarly, LDD Sampling satisfies this definition since the labeling is precisely the contraction induced by the low-diameter decomposition clustering, and it is easy to check that this is a valid partial labeling. Finally, since we use a linearizable union-find algorithm as our sub-routine in k-out Sampling, a proof almost identical to that of Theorem 1 shows that the labeling induced by k-out Sampling is a correct partial labeling of G. □

B.2 Finish Algorithms

We now finish describing our ConnectIt finish methods and providing deferred correctness proofs for our finish methods.

B.2.1 Correctness for Linearizably Monotonic Methods.

Theorem 1. *The connectivity algorithm in* ConnectIt *(Algorithm 1) applied with a sampling method satisfying Definition 3.1 and a linearizably monotone finish method is correct.*

Proof. Let C be the partial labeling produced by the sampling method $\mathcal{S}$. Since $\mathcal{S}$ satisfies Definition 3.1, we have that the labeling C consists of a set of rooted trees. Now, consider $\mathcal{F}$, which is a linearizably monotone finish method operating on the partial connectivity labeling C. Since we are handling an abstract connectivity algorithm $\mathcal{F}$, it is helpful to think about operations of two types: (i) those that try to "apply" an edge (u, v) (i.e., it checks whether u and v are in different trees, and if so changes the labeling so that u and v are in the same tree in the new labeling) and (ii) lower-level operations that compress trees. Furthermore, note that $\mathcal{F}$ will avoid applying (directed) edges coming out of *frequentid*, which could be problematic since there may be many edges of the form (u, v) where $u \in$ *frequentid* and $v \notin$ *frequentid* that must be examined to compute a correct labeling. However, since the graph is symmetric, notice that $\mathcal{F}$ does apply all such inter-cluster edges, since this edge will be applied while processing vertex v, which is contained in an initial component $c \neq$ *frequentid*. The algorithm must process this component, since it only skips applying the directed edges incident to vertices in component *frequentid*.

Consider the linearizatiaon order of the finish algorithm $\mathcal{F}$. Before running $\mathcal{F}$, the labeling C represents some set of intermediate clusters that is a partial connectivity labeling of G (i.e., if there is a path between two vertices u and v in the trees represented by the labeling C, then this implies that there is a path between u and v in G). We will show by induction on the linearization order that each operation preserves the fact that C is a partial connectivity labeling of G. Consider the two types of operations:

- Operations of type (ii) described above are trivial to handle since the operations only change the structure of the labeling, and not the connectivity information represented by it. Thus, the labeling is still a valid partial connectivity labeling of G.

- Next, consider an operation of type (i), which applies an edge (u, v). If u and v are in the same tree before the operation, the operation does not modify the structure of the trees and thus the new connectivity labeling (if any labels changed) is equivalent to the previous labeling (and is thus valid). Otherwise, if u and v are in different trees, the operation will produce a new labeling where u and v are in the same tree, and the labeling for any vertices z not in the same tree as either u or v is unmodified. Thus, the new labeling is a partial connectivity labeling of G, since the algorithm only changes labels of vertices in the trees containing u and v, and by induction all vertices in each of u's and v's trees were connected by paths in G, and due to the (u, v) edge, all vertices in the union of these trees have paths between each other in G.

Therefore, at the end of the execution, the algorithm has computed a partial connectivity labeling, and has linearized the apply operations of all (u, v) edges where u and v are not both in *frequentid*. Note that all edges (u, v) where both u and v are initially in *frequentid* are initially in the same tree. Thus, the algorithm has computed a connectivity labeling, having applied all edges $(u, v) \in E$, and so for any $u, v \in V$, u and v are in the same tree in C if and only if there is a path between u and v in G, completing the proof. □

B.2.2 *Correctness Properties of Rem's Algorithms.*

THEOREM 3. Union-Rem-CAS *and* Union-Rem-Lock *are correct connectivity algorithms in the phase-concurrent setting, and result in linearizable results for a set of phase-concurrent union and find operations.*

PROOF. We provide a high-level proof sketch. The proof is based on building a graph of completed and currently active union operations. Let this graph be G_C. The graph (roughly) corresponds to the trees concurrently maintained by the algorithm, and includes extra edges corresponding to the currently executing UNION operations.

- When a UNION(u, v) operation starts (is invoked), we place a *shadow edge* between the root of the tree currently containing u and the tree currently containing v.
- When a UNION(u, v) operation completes (we receive its response), we first check if it returns True due to atomically hooking a tree root r_u to a vertex h_v. If so, we link the tree containing r_u to h_v in the graph. Then, we remove the shadow edge corresponding to this Union operation from the graph.

The proof is by induction on the completion times of the operations that (i) once a UNION(u, v) call completes, u and v always remain connected in G_C and (ii) any vertices u and v in different components in G are never connected in G_C. The proof is similar to that of Theorem 1. □

B.3 Counter-example for Rem's Algorithm with SpliceAtomic and FindCompress

As mentioned in Section 3.3, combining the FindCompress option with the SpliceAtomic rule results in incorrect Rem's algorithms (Union-Rem-Lock and Union-Rem-CAS), which is shown in Figure 5 as a counter-example. Without loss of generality, we will explain why Union-Rem-Lock can lead to the incorrected result with FindCompress and SpliceAtomic.

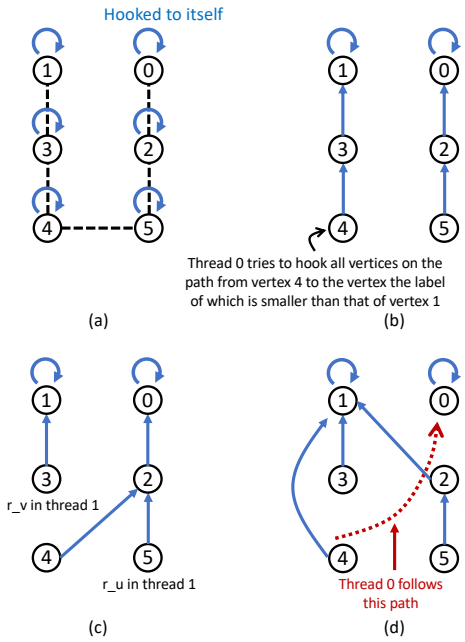


Figure 5: A counter example of Rem's algorithm with SpliceAtomic and FindCompress.

Suppose $V = \{0, 1, 2, 3, 4, 5\}$ and $E = \{(0, 2), (1, 3), (2, 5), (3, 4), (4, 5)\}$ (Figure 5 (a)). Assume Rem's algorithm is applied to the edges $(2, 5)$ and $(0, 2)$ sequentially. After that, thread 0 applies Union-Rem-Lock to the edge $(3, 4)$, and is about to execute Line 11 in Algorithm 8 (i.e., thread 0 is executing Line 17 in Algorithm 13). Thereafter, another thread is done with applying Union-Rem-Lock to the edge $(1, 3)$, the state of which is shown in Figure 5 (b).

Now, thread 1 applies Union-Rem-Lock to the edge $(4, 5)$: after one iteration of while statement (Line 6 in Algorithm 13), $P[4]$ is changed from 3 to 2, and $r_u = 5$, $r_v = 3$ (Figure 5 (c)).

After that, thread 0 executes the while loop (Line 11 in Algorithm 8), which hooks $P[4]$ and $P[2]$ to 1 (Figure 5 (d)); $P[0]$ is not hooked as $P[0] < P[1]$.

Now, vertex 0 is isolated, and thread 1 cannot link vertex 1 to any other vertices during Union-Rem-CAS. Thus, the number of connected components after applying Union-Rem-Lock is 2, which is wrong.

B.3.1 Shiloach-Vishkin. We include Shiloach-Vishkin's algorithm in CONNECTIT, which is a classic parallel connectivity algorithm. The algorithm works by combining vertices into trees over a series of synchronous rounds using linking rules. Since only roots of trees can be linked, the algorithm is naturally monotonic. We provide pseudocode for our implementation of Shiloach-Vishkin in Algorithm 15 in Appendix D. The algorithm iteratively hooks roots of trees onto each other and compresses these trees by linking all vertices to the root of the tree. Each *round* of the algorithm maps over all edges. If an edge goes between two tree roots, the algorithm tries to *hook* the larger root to the smaller one. This operation can

either be done using a plain write, which is what existing implementations of the algorithm use, or using a ***writeMin*** to hook to the lowest incident root, which is what our algorithm does. A round ends by making the parent pointers of all vertices point to the root of their tree using pointer jumping. Finally, by considering the hook operations the algorithm tries to apply to each edge as a set of concurrent union operations, it is easy to show that this algorithm is linearizable, and thus linearizably monotonic. Overall, our implementation of Shiloach-Vishkin requires $O(\log n)$ rounds, and each round can be implemented in $O(n + m)$ work and $O(\log n)$ depth.

B.3.2 Stergiou. Stergiou et al. [101] presented a simple connectivity algorithm for the Bulk Synchronous Parallel (BSP) model. Stergiou's algorithm is nearly expressible in the Liu-Tarjan framework, with the exception that the algorithm uses two parents arrays, *prevParent* and *curParent*, instead of just one array as in the Liu-Tarjan algorithms. In each round, the algorithm first updates *prevParent* to be equal to *curParent*. It then performs a ParentConnect, updating *curParent* based on the parent information in *prevParent*. Finally, it applies Shortcut on *curParent* and repeats until the parent array no longer changes. This algorithm is similar to the PUS algorithm of Liu-Tarjan, except that it uses two parent arrays instead of a single array. Stergiou's algorithm is not monotonic since the algorithm can relabel non-root vertices in the parents array. The correctness proof for this algorithm is similar to the one for the Liu-Tarjan framework. We compose Stergiou's algorithm with sampling methods in the same way as with the Liu-Tarjan framework.

B.3.3 Label-Propagation. Label-Propagation is arguably the most frequently implemented parallel graph connectivity today, and has been implemented by many existing popular graph processing systems, including Pregel, Giraph, and other frameworks [26, 42, 70, 79, 91]. The Label-Propagation algorithm can be viewed as iterative sparse-matrix vector multiplication (SpMV), where the product is done over the $(\min, \min)$ semiring.

The implementation of this algorithm works as follows. Initially, each vertex is its own parent (we interpret the parents in the algorithm as the labels). In each round, the algorithm maintains a subset of vertices called a ***frontier*** that had their parent change in the previous round (at the start of the algorithm, this set contains all vertices). Then, in each round, the algorithm processes all edges incident to vertices in the current frontier, writing the minimum parent ID to neighbors using a WRITEMIN. The algorithm terminates once the array of parents no longer changes, which will occur within D rounds where D is the diameter of the graph. In the worst case, the algorithm will process all edges in each round giving overall work $O(mD)$ and depth $(D \log n)$. Note that the Label-Propagation algorithm is not monotonic.

The correctness of this algorithm is folklore, but we provide a generic proof of correctness for it, and other min-based algorithms like Liu-Tarjan and Stergiou's algorithm in Theorem 4 below. We compose Label-Propagation with sampling methods by using the same idea as with our sampled Liu-Tarjan framework, since relabeling the largest component to have the smallest ID ensures that vertices in this component never change their ID in subsequent rounds.

Correctness for Min-Based Methods

We start by defining min-based finish methods (also defined in Section 2), which capture Liu-Tarjan's algorithms, Label-Propagation, and Stergiou's algorithm.

Definition B.1. A ***min-based finish method*** is a round-based algorithm that repeatedly performs an "application" step on all edges. An edge application takes an edge (u, v) and may update $P[u]$ (resp. $P[v]$) based on $P[v]$ (resp. $P[u]$), where a parent value is updated if and only if the new value is strictly smaller than the previous value. The algorithm terminates once the parent values stop changing.

Label-Propagation, Liu-Tarjan's algorithms, and Stergiou's algorithms all clearly satisfy this definition. The following theorem states that composing min-based algorithms with sampling algorithms produces correct results in CONNECTIT.

THEOREM 4. *The connectivity meta-algorithm in* ConnectIt *(Algorithm 1) applied with a sampling method $\mathcal{S}$ satisfying Definition 3.1 and a min-based finish method $\mathcal{F}$ produces a correct connectivity labeling.*

PROOF. The proof is by contradiction. Let C be the labels produced by $\mathcal{S}$ satisfying Definition 3.1, that is they are consistent with some contraction of G such that composing the contraction with the connectivity induced by inter-component edges results in a correct connectivity labeling of G. The proof only relies on the fact that the sampling method only maps vertices to the same label in C if they are in the same component. Let C' be the final connectivity labeling after running $\mathcal{F}$.

Suppose that two vertices u and v in the same component in G are assigned different labels in C'. Since u and v are connected, there is a path between them in G, and on this path there must be an edge (x, y) where $C'[x] \neq C'[y]$. However, since the min-based method applies all edges in each round, on the last round it must have applied the (x, y) edge, and since $C'[x] \neq C'[y]$, one of these values must have changed in this round, contradicting the assumption that this was the last round of the algorithm.

Furthermore, the min-based algorithms composed with sampling methods can skip the edges incident to the largest sampled component, since the largest sampled component is relabeled to have the smallest ID, and thus by the preceding argument, all vertices that are connected to the largest sampled component in G will eventually acquire its ID. □

B.4 CONNECTIT: Spanning Forest

In this section, we show how to extend CONNECTIT to solve the problem of generating a spanning forest of the graph. Algorithm 2 shows the generic CONNECTIT algorithm for this problem. Like our connectivity algorithm, the algorithm can be supplied with different combinations of sample and finish methods and generate correct spanning forest algorithms. The main idea in the algorithm is to map each edge in the spanning forest to a unique vertex that is one of the edge's endpoints.

Algorithm Description. The main changes between the CONNECTIT connectivity algorithm and spanning forest algorithm are as follows. The spanning forest algorithm requires (i) supplying an *edges* array to the algorithm, which is an array of size $|V|$ where

each entry is initially an edge pair of (∞, ∞) (Line 3); (ii) the sampling method generates a subset of the spanning forest edges in addition to computing partial connectivity information (Line 4); and (iii) the *finish* procedure now takes both the partially computed components and the partially computed forest from the sampling step, and finishes computing the spanning forest of the graph (Line 6). Finally, since some of the vertices may not have a corresponding spanning forest edge, we filter the EDGES array to remove any pairs which are not edges (Line 7). Our goal when building this interface was to allow as many finish methods as possible to work with it, and so we did not consider optimizations, such as replacing the edges array, which is an array of edge pairs, by a parents array which would only store a single endpoint per vertex (such a modification could be made to work for a small subset of the finish methods, but would exclude many other finish methods from working in the interface).

We show in the following subsections that any sampling method can be combined with any *root-based finish method* to generate correct spanning forest algorithms (Theorem 6). Our framework currently only works for root-based algorithms, and thus excludes algorithms such as the class of non-root-based algorithms in Liu-Tarjan's framework.

Algorithm 2 CONNECTIT Framework: Spanning Forest

```
1: procedure SPANNINGFOREST(G(V, E), sampling, finish)
2:     labels ← {i → i | i ∈ [|V|]}
3:     edges ← {i → (∞, ∞) | i ∈ [|V|]}
4:     (labels, edges) ← sampling.SAMPLEFOREST(G, labels, edges)
5:     frequentid ← IDENTIFYFREQUENTCOMP(labels)
6:     (labels, edges) ← finish.FINISHFOREST(G, labels, edges, frequentid)
7:     return FILTER(edges, FN(u, v) → ((u, v) ≠ (∞, ∞)))
```

B.4.1 Sampling Methods for Spanning Forest. We first introduce the following correctness definition for spanning forest sampling methods, which ensures that the sample methods compose correctly with all finish methods that are subsequently applied.

Definition B.2 (Sampling for Spanning Forest)**.** Given an undirected graph $G(V, E)$ let $\mathcal{E}_S$ denote the set of forest edges implicitly generated by a sampling method $\mathcal{S}$, and C denote the connectivity labeling after applying S. The sampling method $\mathcal{S}$ is correct if

(1) The sampling method satisfies Definition 3.1.
(2) $\text{CONNECTIVITY}(V, \mathcal{E}_S) = C$, i.e., finding the connected components on the spanning forest edges induces the connectivity labeling C.
(3) Each edge $e = (u, v) \in \mathcal{E}_S$ is assigned to exactly one vertex, and no vertex has more than one edge assigned to it.

In other words, requirement (2) corresponds to the intuitive notion that the connectivity labeling generated by the sampling method corresponds exactly to contracting the subset of spanning forest edges returned by the method. Requirement (3) is required to correctly compose the output of our sampling methods with our finish methods. We now discuss how to adapt each of the sampling methods presented in Section 3.2 to satisfy Definition B.2.

k-Out Sampling. For k-Out sampling, we require the following modifications. First, we include an edge $e = (u, v)$ as a forest edge if it was successfully used to hook a root to a different vertex in the underlying union-find algorithm used by the k-Out sampling method. Let $r(u)$ and $r(v)$ be the roots of u and v, respectively, at the time this edge is hooked. Without loss of generality, suppose $r(u)$ is the root we hook. We store this edge in the EDGES array by setting $\text{EDGES}[r(u)] = e$.

BFS Sampling and LDD Sampling. We modify BFS sampling to take all edges in the BFS tree as spanning forest edges. Consider a tree edge $e = (u, v)$ in the BFS tree where u is the parent of v in the tree. Then, we assign e to v as part of the sampling method. Finally, for low-diameter decomposition sampling we take the edges traversed by the searches in the LDD as spanning forest edges. As in BFS sampling, we assign each tree edge $e = (u, v)$ to the child in the BFS tree rooted at the cluster containing u and v.

THEOREM 5. *k-out Sampling, BFS Sampling, and LDD Sampling all satisfy Definition B.2.*

PROOF. For k-out Sampling, since the union-find algorithm is monotonic, it is easy to verify that the set of forest edges we include from the sampling produces exactly the trees represented by the parent array after k-out sampling, and thus this method satisfies requirement (1) of Definition B.2. Requirements (2) and (3) can also be satisfied since the k-out sampling algorithm only hooks roots. Recall that when the algorithm applies an edge (u, v) that hooks a root $r(u)$ to $r(v)$, it sets $\text{EDGES}[r(u)] = e$. This assignment satisfies (3) since e is included in the forest, and is only assigned to exactly one vertex, namely $r(u)$. It also satisfies the second part of (3) since only vertices that are roots at some point in the algorithm have edges assigned to them, and every root is hooked at most once. Therefore, a vertex has at most one edge assigned to it, as required.

Let T be the BFS tree of the sampled component. Since a BFS tree is clearly connected, the sampling method clearly satisfies requirement (1) of Definition B.2. Now, consider a tree edge $e = (u, v)$ in the BFS tree where u is the parent of v in the tree, and recall that we assign e to v. Since each vertex in the tree has exactly one parent, this assignment satisfies both requirements (2) and (3) of Definition B.2.

The proof for LDD Sampling is identical to that of BFS Sampling since each cluster in the LDD is explored using a BFS rooted at the center. We described the algorithm used to compute LDD earlier in Section 3.2, and recall that it works by sampling a start time from an exponential distribution for each vertex, and performing a simultaneous breadth-first search from the vertices. Vertices join the search at their start time if they are not already covered by a cluster. We refer to Miller et al. [75] and Shun et al. [94] for more details on LDD. □

B.4.2 Finish Methods for Spanning Forest. We first apply our framework to handle a class of ***root-based spanning-forest algorithms***. As the name suggests, a root-based spanning-forest algorithm is one that adds an edge to a spanning forest if the edge successfully hooks the root of the tree containing one of its endpoints to a smaller vertex in the other endpoint's tree. It is easy to see that the entire class of union-find algorithms considered in this paper satisfy this definition, since each of the union steps in these algorithms only operates on roots, and atomically hooks a root to a smaller vertex in the other tree. The main correctness property of our framework for spanning forest is summarized by the following theorem:

THEOREM 6. *Given an undirected graph $G(V, E)$, composing a valid sampling method $\mathcal{S}$ (as per Definition B.2) with a root-based spanning forest method $\mathcal{F}$ using Algorithm 2 outputs a correct spanning forest.*

PROOF. We provide a high-level proof sketch. The parents array, P, emitted by the sampling method represents a set of rooted trees, where the forest edges discovered by the sampling method, E_S, induce exactly the connectivity information stored in the trees. Applying a root-based spanning forest method to this configuration (the parents array, P, and the edge array, E_S) can be viewed as applying a root-based spanning forest algorithm to $G[P]$, the graph induced by the connectivity mapping generated by the sampling method.

The finish method by definition will hook a number of roots during its execution. Since each root can be hooked at most once, and does not have an edge written into its index in E_S yet (by Definition B.2), the finish method can successfully store this edge in E_S. Since the finish method is correct, the algorithm outputs a correct spanning forest for $G[P]$. Applying the fact that the sampling method also satisfies Definition 3.1 shows that the union of the forest generated by the sampling method and the forest generated on $G[P]$ is a spanning forest for G. □

Theorem 6 provides correctness guarantees for the following algorithms in our framework, which can be composed with all three sampling methods:

- All union-find variants, including Rem's algorithms, and all internal parameter combinations for these algorithms
- Shiloach-Vishkin's algorithm
- The root-based algorithms in the Liu-Tarjan framework

B.5 CONNECTIT: Streaming

Algorithm Description. We first describe the generic CONNECTIT algorithm for computing streaming connectivity. The pseudocode for our algorithm is described in Algorithm 3.

Algorithm 3 is initialized similarly to the CONNECTIT connectivity algorithm (Algorithm 1) on an initial graph (which is possibly empty). Before batch-processing starts, the framework calls the INITIALIZE method on the initial graph (Line 9). The PROCESSBATCH function provided by the framework can then be called by clients, where each PROCESSBATCH call is executed internally in parallel by calling the underlying finish algorithm's batch-processing mechanism (Line 11). The algorithm then returns the results of the queries (Line 12).

Algorithm 3 CONNECTIT Framework: Streaming

```
1:  procedure INITIALIZE(G(V, E), sampling_opt, finish_opt)
2:      sampling ← GETSAMPLINGALGORITHM(sample_opt)
3:      finish ← GETFINISHALGORITHM(finish_opt)
4:      components ← {i → i | i ∈ [|V|]}
5:      sampling.SAMPLECOMPONENTS(G, components)
6:      frequentid ← IDENTIFYFREQUENT(components)
7:      finish.FINISHCOMPONENTS(G, components, frequentid)
8:      return finish
9:  let BC ← INITIALIZE(G, sampling_opt, finish_opt)
10: procedure PROCESSBATCH(B = {updates, queries})
11:     query_results ← BC.PROCESSBATCH(B)
12:     return query_results
```

Correctness. What do we mean by an algorithm being correct in the parallel batch-incremental setting? A natural definition is to enforce that the algorithms are linearizable for a set of union and find operations, and that the linearization points of operations that occur in previous batches are fixed, i.e., operations in future batches cannot affect the linearization points of operations in previous batches that have already happened. If union and find operations within a batch are run concurrently, we require these operations to be linearizable with respect to the connectivity information at the start of the batch. We use this definition of correctness in what follows, which we believe is intuitive, and captures the desired properties of parallel batch-incremental algorithms.

Recall the different types of streaming algorithms introduced in Section 3.5. The reason we categorize algorithms as Type (iii), and apply the updates and finds phase concurrently is because these algorithms are not linearizable when applying finds concurrently with updates.

The proof that the algorithms of Type (i) and Type (ii) are correct according to this definition follows due to the fact that these algorithms are linearizable for a collection of concurrent unions and finds. The proof that algorithms of Type (iii) are correct in the parallel batch-incremental setting when run phase-concurrently follows from their correctness for a collection of edge updates in the static setting. After applying all updates, the parents array stores the correct connectivity information such that two vertices are in the same component if and only if they have the same root in the parents array. Therefore, we have the following theorem:

THEOREM 7. *Both Type (i) and Type (iii) streaming algorithms implemented in* ConnectIt *are correct when run in the parallel batch-incremental setting.*

C CONNECTIT EVALUATION

In this section we provide additional experimental results that are deferred from the main body of the paper.

C.1 Parallel Connectivity without Sampling

This section reports additional results, and performance analyses about CONNECTIT in the setting where no sampling is performed.

Union-Find Algorithms. We start by analyzing how quantities such as the Max Path Length and Total Path Length for union-find algorithms affect the algorithm performance, as well as how hardware counters can explain performance differences between union-find algorithms.

Maximum Path Length. Figure 6 plots the ***Max Path Length*** statistic vs. the total running time for different union-find variants on the road_usa and Hyperlink2012 graphs, which are the smallest and largest graphs used in our evaluation (the plots for other graphs are elided in the interest of space; we note that they are very similar). We observed that this value is consistently small across all executions for both the smallest and largest graph we test on, reaching a maximum of 36 on the road_usa graph, and a maximum of less than 100 on the Hyperlink2012 graph. Perhaps surprisingly, this quantity does not seem to be correlated with the running time—on the Hyperlink2012 graph the Union-Rem-CAS implementation has one of the largest

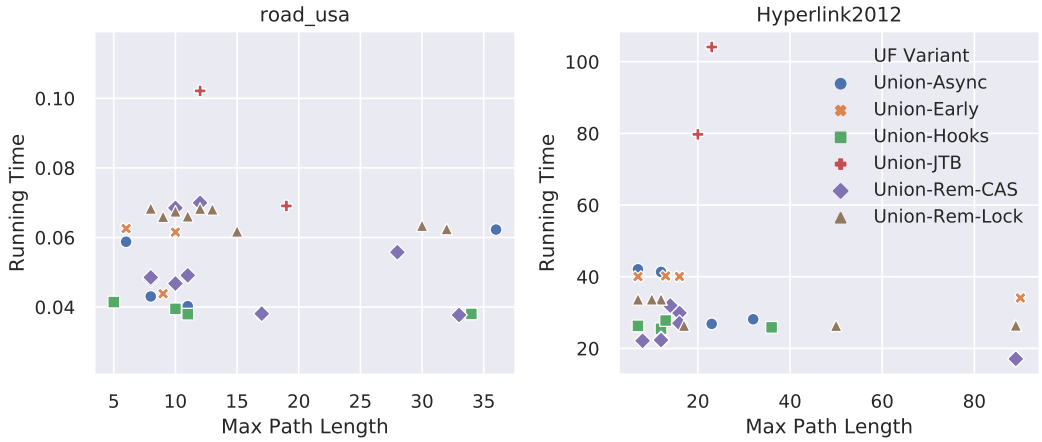


Figure 6: Plot of the Max Path Length encountered during algorithm execution vs. running time in seconds for the road_usa and Hyperlink2012 graphs. Running times are obtained on a 72-core machine with 2-way hyper-threading enabled.

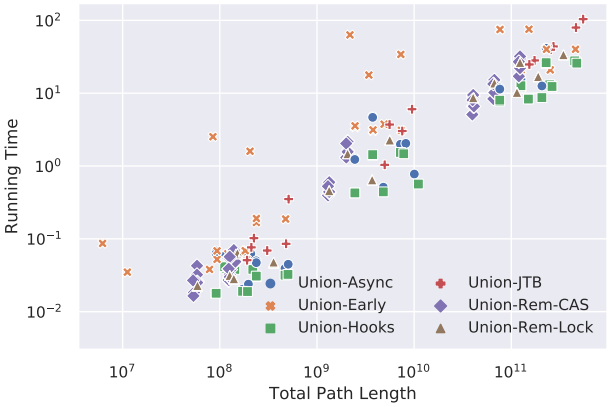


Figure 7: Plot of the Total Path Length traversed during algorithm execution (log-scale) vs. running time in seconds (log-scale) for all union-find experiments on all of our inputs. Running times are obtained on a 72-core machine with 2-way hyper-threading enabled.

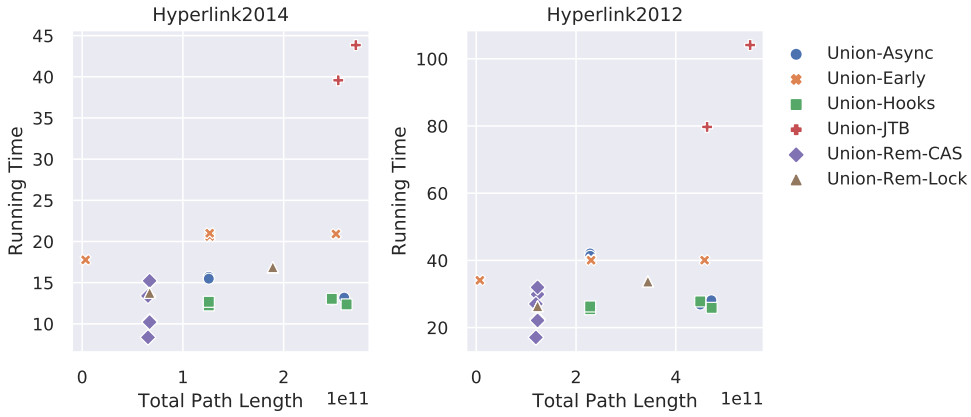
Max Path Lengths, but still achieves the lowest running time in this setting.

Total Path Length. Figure 7 plots the Total Path Length statistic vs. the total running time across all graphs for different union-find variants (both axes are shown in log-scale). The plot shows that the ***Total Path Length*** is highly relevant for predicting the running time—the Total Path Length has a Pearson correlation coefficient of 0.738 with the running time (the Max Path Length has a much weaker coefficient of 0.344). Figure 8a plots the same statistic for the Hyperlink2014 and Hyperlink2012 graphs, where we see a similar trend. The poor performance of the Union-JTB algorithms under both find configurations is explained largely by the large total path lengths traversed by these algorithms (the algorithm with larger Total Path Length, and running time was always the FindNaive implementation). We note that the Union-Early algorithm exhibits a single data point which experiences significantly lower Total Path Length than any of the other algorithms. This data point is for the version of the Union-Early algorithm combined with the FindNaive strategy. Since the algorithm does not perform any finds, the only contribution to the path length is from the path length traversed by the Union method. Despite this fact, it is interesting that the Union-Early code with the FindNaive option incurs such a low Total Path Length. We note that FindNaive is always the fastest option for this algorithm.

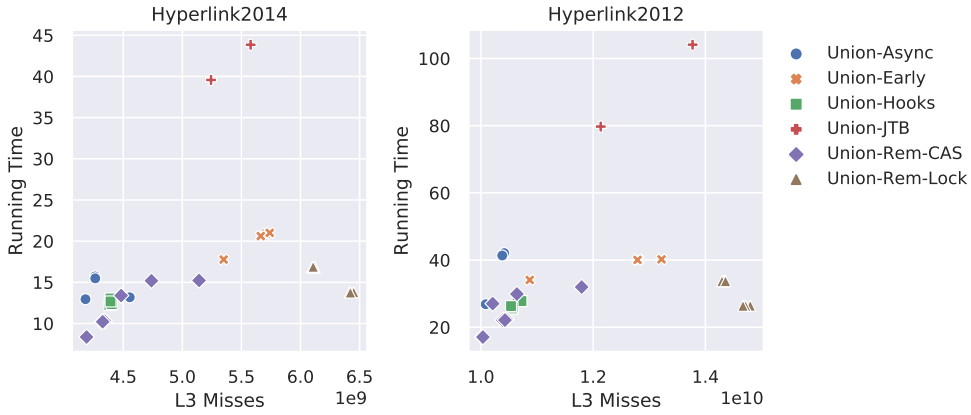
LLC Misses. In Figure 9, we plot the number of LLC misses vs. running time across all graphs. Figure 8b plots the same quantities for the Hyperlink2014 and Hyperlink2012 graphs. We see that the Union-Early implementations, which perform poorly on the Hyperlink2014 graph incur a large number of LLC misses. The Pearson correlation coefficient between the number of LLC misses and the overall running time is 0.797 across all graphs, which is comparable with the correlation between the Total Path Length and the running time. Lastly, we observe that the fastest algorithms incur the fewest number of LLC misses, with Union-Rem-CAS with the FindAtomicSplit option achieving the point in the bottom left corner of the plot for the Hyperlink graphs.

Total Bytes Written/Read. In Figure 10, we plot the total gigabytes written to and read from the memory controller vs. running time, across all graphs. Figure 8c plots the same quantities for the Hyperlink2014 and Hyperlink2012 graphs. As one might expect, the trends appear to be similar to the number of LLC misses. Perhaps surprisingly, although the Union-Rem-Lock requires the largest amount of memory traffic, these algorithms are much faster than algorithms that read and write fewer bytes across the memory controllers (Union-Early and Union-JTB). This quantity has a Pearson correlation coefficient with the running time of 0.774, which is similar to that of the LLC misses statistic. Finally, we observe that the Union-Rem-CAS and Union-Async algorithms perform require reading and writing very little data.

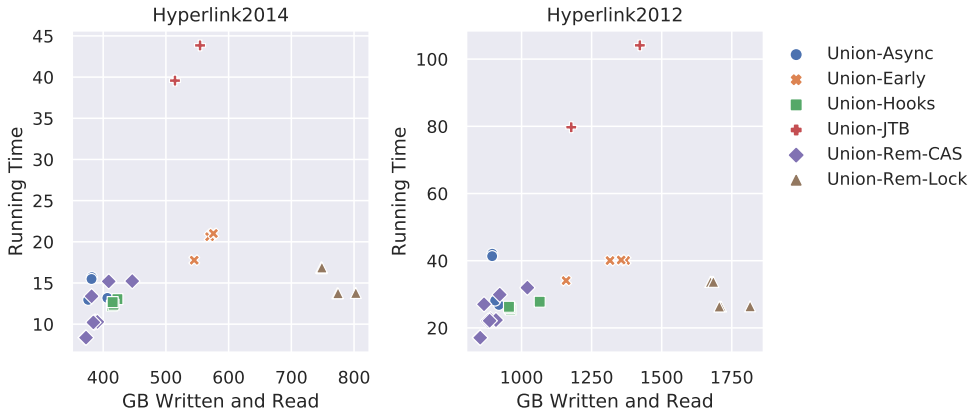
C.1.1 Liu-Tarjan Algorithms. Figure 11 shows the relative performance of different Liu-Tarjan implementations from CONNECTIT in the *No Sampling* setting. These results are averaged across all graphs used in our evaluation, and are computed the same way as in our union-find evaluation previously.



(a) Plot of the Total Path Length traversed during algorithm execution (log-scale) vs. running time in seconds (log-scale) for all union-find experiments on the Hyperlink2014 and Hyperlink2012 graphs.



(b) Plot of the number of LLC misses during algorithm execution vs. running time in seconds for all union-find experiments on the Hyperlink2014 and Hyperlink2012 graphs.



(c) Plot of the total GB written and read during execution vs. running time in seconds for all union-find experiments on the Hyperlink2014 and Hyperlink2012 graphs.

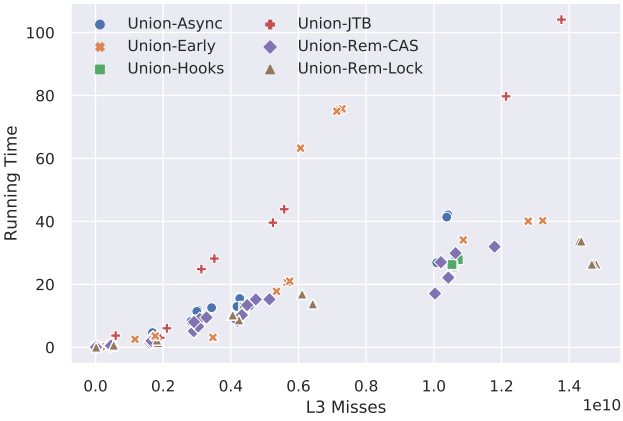


Figure 9: Plot of the number of LLC misses incurred during algorithm execution vs. running time in seconds for all union-find experiments on all of our inputs. Running times are obtained on a 72-core machine with 2-way hyper-threading enabled.

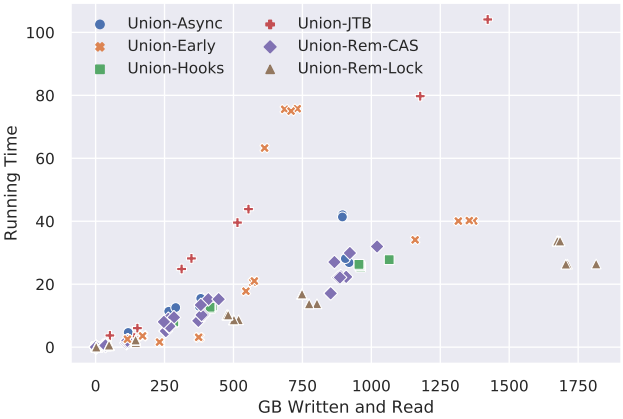


Figure 10: Plot of the total number of bytes written and read (in GB) during algorithm execution vs. running time for all union-find experiments on all of our inputs. Running times are obtained on a 72-core machine with 2-way hyper-threading enabled.

We observed that the four fastest Liu-Tarjan variants in this setting are the new variants we implemented that perform a FullShortcut, namely PF, EF, PRF, and ERF. The basic implementation (P) that performs a Update, Shorcut and ParentConnect is also highly competitive across all graphs. The remaining methods achieve between 1.3x–1.5x slowdown on average across all graphs, with the slowest algorithms on average being those using the Alter method.

Figure 12 analyzes how the number of LLC misses affects the running time. We see that the algorithm performance of the Liu-Tarjan variants is significantly determined by the number of LLC misses (the Pearson correlation coefficient is 0.982). We note that the algorithms using the Alter method are sometimes the fastest Liu-Tarjan variants, which can be seen in Figure 12.

C.2 Parallel Connectivity with Sampling

This section reports additional results, and performance analyses about CONNECTIT in the setting where sampling is performed. In particular, we evaluate how CONNECTIT algorithms perform when

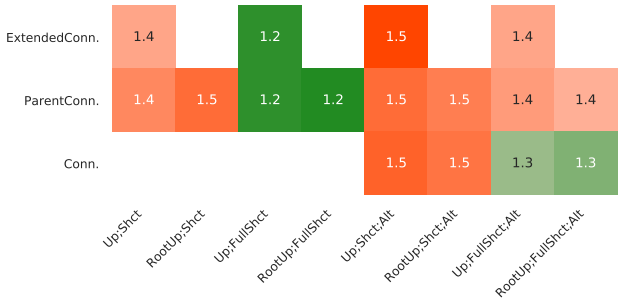


Figure 11: Relative performance of different Liu-Tarjan implementations on graphs used in our evaluation of CONNECTIT in the *No Sampling* setting. Running times are obtained on a 72-core machine with 2-way hyper-threading enabled.

combined with the k-out Sampling, BFS Sampling, and LDD Sampling strategies, and discuss the speedups of the sampling-based algorithms compared to the performance numbers without sampling presented in Section 4.1.

CONNECTIT Using k-out Sampling. The second group of rows in Table 3 shows the performance of CONNECTIT algorithms using the k-out Sampling scheme on our graph inputs. We immediately observe that the performance of CONNECTIT algorithms combined with k-out Sampling is consistently fast, and produces the fastest overall algorithm on almost half of our graph inputs (across both small and large graphs). Like in the no-sampling setting, the union-find algorithms achieve very high performance, with Union-Async and Union-Rem-CAS achieving extremely similar performance: their running times are within a few percent of each other on all of the graphs in this setting. Unlike in the no-sampling setting, the performance of other union-find variants is very close to that of the fastest union-find variants, with the slowest overall variant on average being Union-Rem-Lock. On average, the union-find implementations achieve a 1.94x speedup over the fastest connectivity implementations without sampling, with the highest average speedup obtained by Union-Rem-CAS (2.07x faster than the fastest algorithms without sampling, on average). Our Liu-Tarjan, Shiloach-Vishkin, and Label-Propagation algorithms also achieve high performance when compared with the fastest algorithms for each graph. Liu-Tarjan achieves 1.55x average speedup over the fastest algorithm in the no sampling setting, Shiloach-Vishkin achieves 1.53x average speedup, and Label-Propagation achieves 1.59x average speedup. This indicates that these algorithms compose efficiently with the k-out Sampling scheme.

Across all of our graphs, the one exception where k-out Sampling does not provide significant speedups is for the road_usa graph. The main reason is that the road_usa graph has very low average degree, with most vertices having between 2–5 edges. Therefore, the cost of simply going over this small set of edges twice during two-phase execution outweighs the additional benefit provided by sampling, since we almost finish computing connectivity after applying k-out

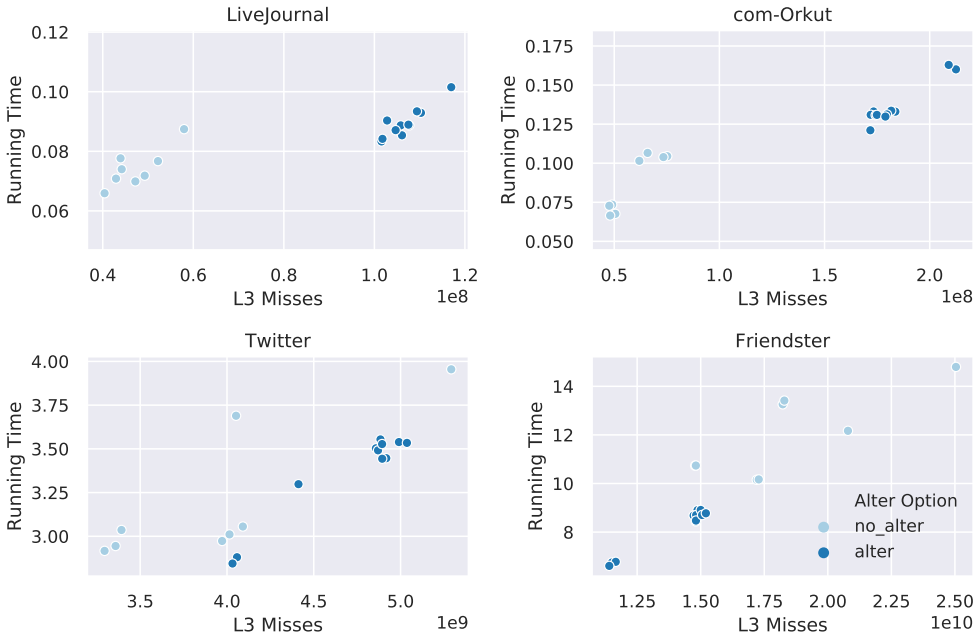


Figure 12: Plot of the number of LLC misses for each algorithm execution vs. running time in seconds for the LiveJournal, com-Orkut, Twitter, and Friendster graphs. Running times are obtained on a 72-core machine with 2-way hyper-threading enabled.

Sampling (see Section C.4 for more experimental analysis of *k*-out Sampling on this graph).

Next, we compare the effectiveness of *k*-out Sampling compared with the fastest results obtained by any combination of sampling and finish methods in the ConnectIt framework. We can see from Table 3 that the Union-Rem-CAS algorithm achieves the most consistent performance relative to the fastest algorithms, being at most 27% slower than the fastest implementation for all graphs, and within 14% on average. Union-Async and Union-Hooks are similarly competitive, being at most 30% and 31% slower, and within 19% and 18% of the fastest on average respectively. We conclude that applying one of these three union-find algorithms in addition to *k*-out Sampling results in consistent high performance.

To summarize, we find that *k*-out Sampling is an effective and robust sampling scheme that consistently works across a variety of different graph types, and algorithm types (union-find variants, Liu-Tarjan variants, and Shiloach-Vishkin) to produce algorithms that are within 1.12x on average of the fastest running times obtained by any algorithm in our framework. Finally, for the Hyperlink2012 graph, the largest publicly-available real-world graph, the Union-Rem-CAS algorithm combined with *k*-out Sampling achieves the fastest time out of all algorithm combinations in our framework, running in *8.2 seconds* in total. This is a *3.14x improvement* over the fastest known result for this graph, previously achieved by the Graph Based Benchmark Suite [31] using the work-efficient connectivity algorithm by Shun et al. [94], and is competitive with two empirical lower-bounds on the performance of any connectivity algorithm (analyzed in Section C.4.1).

Performance Using BFS Sampling. The third group of rows in Table 3 show the performance of ConnectIt algorithms using the BFS Sampling scheme on our graph inputs. We can immediately see that the performance of this method depends significantly on the diameter of the input graph (listed in Table 2) as the number of rounds in parallel BFS is proportional to the diameter—for example, the performance of these methods on the road_usa graph is significantly worse than the unsampled running times (roughly 60–70x slower). One interesting exception is for Label-Propagation, where the algorithm is 5.1x faster than the unsampled version, since the Label-Propagation algorithm can avoid processing the high-diameter component identified by the BFS.

The performance on the remaining graphs is highly competitive: the fastest running time using BFS Sampling achieves between 1.2–4.9x speedup over the fastest running time without sampling, and 2.5x speedup on average. Like with *k*-out Sampling, the fastest running times using BFS Sampling are obtained using union-find variants, with Union-Rem-CAS running within 3% of the fastest variant using BFS Sampling. Union-Async and Union-Hooks are similarly competitive, running between 1.1–4.7x faster and 1.1–4.8x faster than the fastest algorithm without sampling (both achieve 2.4x speedup on average). Finally, we observe that for the three largest graphs, Union-Rem-CAS consistently achieves the fastest performance out of all variants using BFS Sampling, with BFS Sampling achieving the fastest overall running time for ConnectIt on the Hyperlink2014 graph.

Our Liu-Tarjan, Shiloach-Vishkin, and Label-Propagation algorithms compose efficiently with BFS Sampling, achieving significant speedups over the versions of these algorithms that do not use sampling. On graphs excluding road_usa (where as discussed, BFS Sampling performs poorly), the Liu-Tarjan algorithms achieve between 4.4–28x speedup over the unsampled versions, achieving 12.3x speedup over the unsampled versions on average. Our Shiloach-Vishkin algorithm also achieves strong speedups over the unsampled

versions, achieving between 3.87–12.4x speedup, and 7.58x speedup on average. Our Label-Propagation algorithm achieves between 3.41–13.1x speedup over the unsampled Label-Propagation algorithm, and achieves 7x speedup on average.

To summarize, we find that BFS Sampling is a robust and explainable sampling strategy that achieves very high performance when combined with our CONNECTIT algorithms on low-diameter graphs exhibiting a massive connected component, such as social networks and Web graphs. On high-diameter graphs such as road networks, the performance results suggest that a different sampling scheme, such as k-out Sampling would be a better choice, unless the Label-Propagation algorithm is used in which case BFS Sampling is a good choice (the main source of performance improvement for Label-Propagation is having the BFS fully cover the largest component, since the BFS can be accelerated using direction-optimization [10]).

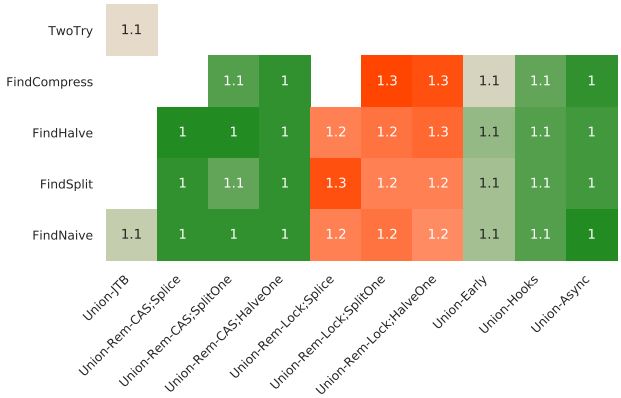


Figure 13: Relative performance of union-find implementations, averaged across all inputs with k-out Sampling.

Performance Using LDD Sampling. Finally, we consider the performance of CONNECTIT algorithms using the LDD Sampling scheme on our graph inputs (the fourth group of rows in Table 3). First, we consider the high-diameter road_usa graph. Compared to BFS Sampling, the performance numbers using LDD Sampling are significantly better (between 22–25x), except for Label-Propagation where the time using LDD Sampling is 5.5x slower since the massive high-diameter component is still slowly traversed by a large number of costly iterations, even after applying LDD Sampling. The reason for the improvement on other algorithms is since LDD Sampling only requires a few rounds to compute an LDD, whereas a BFS on this network requires thousands of rounds to traverse the high-diameter component in this graph.

Focusing now on low-diameter graphs, the performance of LDD Sampling achieves significant speedups over the CONNECTIT algorithms in the no-sampling setting. Our Union-Rem-CAS implementation achieves between 0.98–4.67x speedup over the fastest connectivity implementation in the no-sampling setting, and 2.3x speedup on average. Union-Async and Union-Hooks also achieve high performance, achieving between 0.96–4.6x and 0.95–4.5x speedup (2.34x and 2.28x speedup on average), respectively. The remaining union-find implementations also achieve high performance under this scheme. Overall, we see that Union-Rem-CAS is a robust choice in this setting, and is either the fastest implementation, or within 30% of the fastest across all low-diameter graphs. On the three largest graphs, Union-Rem-CAS is either the fastest algorithm, or within 22% of the fastest running time, and for the largest graph, this algorithm combined with LDD Sampling is within 1% of the fastest running time.

Our Liu-Tarjan, Shiloach-Vishkin, and Label-Propagation algorithms achieve good speedups over the unsampled versions on graphs other than road_usa. The Liu-Tarjan algorithms achieve between 3.5–21.4x speedup over these algorithms in the unsampled versions, and 10x speedup on average. The Shiloach-Vishkin implementation achieves between 3.1–27.3x speedup over the unsampled versions, and 11.6x speedup on average. Finally, the Label-Propagation implementation also achieves good speedup, ranging from between 2.1–9.7x speedup, and 5.7x speedup on average.

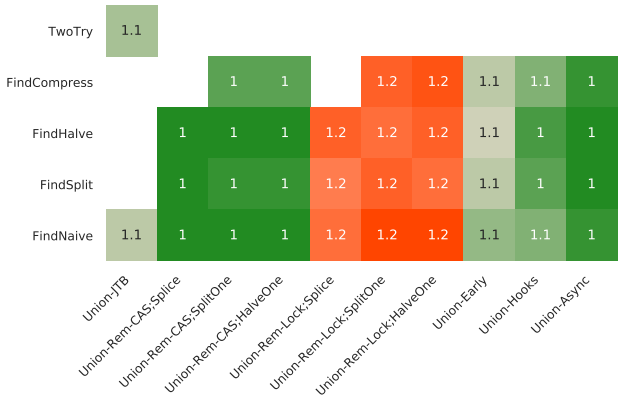


Figure 14: Relative performance of union-find implementations, averaged across all inputs with BFS Sampling.

To summarize, we find that LDD Sampling offers a more robust sampling mechanism compared with BFS Sampling with performance that achieves significant speedups over the CONNECTIT algorithms that do not use sampling. Furthermore, its performance is not overly degraded on high-diameter networks, and achieves excellent performance on low-diameter social networks and Web graphs. We perform a thorough analysis of the parameters to the LDD Sampling scheme in Section C.4.

Union-Find Performance using Sampling. Figures 13, 14, and 15 show heatmaps of the relative performance of different union-find variants when using k-out Sampling, BFS Sampling, and LDD Sampling, respectively. Like Figure 3, the running times are aggregated across all of our graph inputs. Unlike Figure 3, the relative running times of each algorithm are significantly closer together for all three sampling schemes, ranging from 1–1.3x slower than the fastest union-find variant on average for k-out Sampling, 1–1.2x for BFS Sampling, and 1–1.3x for BFS Sampling. As mentioned previously, the fastest union-find variants are just 1.12x slower on average than the fastest CONNECTIT algorithm across all sampling schemes.

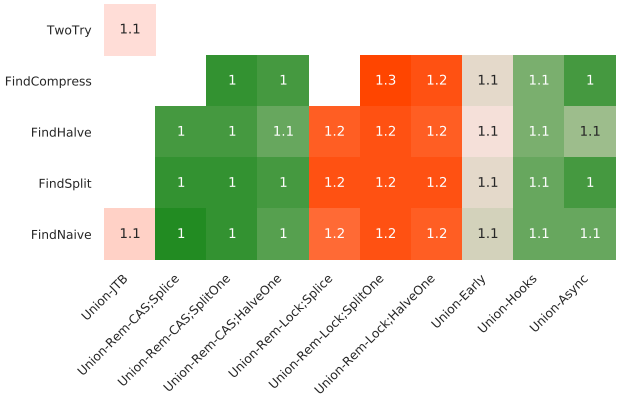


Figure 15: Relative performance of union-find implementations, averaged across all inputs with LDD Sampling.

Therefore, we conclude that applying any union-find variant in conjunction with our sampling schemes results in high performance across all of our graph inputs.

In summary, all three schemes result in meaningful performance improvements over the unsampled versions, and significantly outperform existing state-of-the-art connectivity results using ConnectIt algorithms. For graphs where it is not possible to explore the space of sampling options, we recommend using the k-out Sampling scheme, as it performs the most consistently across a wide range of graph types. In terms of ConnectIt algorithms, we recommend using the Union-Rem-CAS algorithm, which achieves consistently high performance across all of our graph inputs and sampling schemes.

C.2.1 ConnectIt *vs. State-Of-The-Art Connectivity Implementations.* In this section, we provide a detailed comparison between ConnectIt and a set of state-of-the-art publicly-available connectivity implementations.

Comparison with BFSCC [92]. First, we observe that the BFSCC algorithm has highly variable performance that depends on the input graph diameter. In the no-sampling setting for ConnectIt, it ranges from being 92x slower than the fastest ConnectIt algorithm for the road_usa graph, to up to 1.81x and 1.86x faster for the com-Orkut and Twitter graphs, respectively. Compared to ConnectIt algorithms using sampling on instances other than the road_usa graph, where it performs understandably poorly, our implementations are still always faster, ranging from between 1.22–21.7x faster, and 6.46x faster on average.

Comparison with Shun et al. [94]. We observe that the provably work-efficient LDD-based algorithm from Shun et al. [94] achieves more consistent performance across all graphs, including both low and high-diameter networks. Our implementations without sampling are between 1.45–19.4x faster, and 5.6x faster on average. Our implementations with sampling are between 3.1–27.5x faster, and 9.3x faster on average. We note that prior to this paper, this algorithm, implemented as part of the Graph Based Benchmark Suite was the fastest reported running time for the Hyperlink2012 graph, running in 25 seconds [31]. *The fastest* ConnectIt *algorithm,* Union-Rem-CAS *with* SplitAtomicOne *improves on this result, even without using any sampling, running in just under 14 seconds.*

Comparison with MultiStep [98]. The MultiStep method implements a hybrid scheme that performs the BFS Sampling scheme combined with Label-Propagation. The MultiStep algorithm performs $O(\text{diam}(G)m+n)$ work in the worst case, and requires $O(\text{diam}(G)\log n)$ depth, where $\text{diam}(G)$ is the graph diameter. The performance of the algorithm is highly variable, especially on high-diameter graphs like road_usa, where it is 2.2x slower than our Label-Propagation implementation. Compared with the fastest ConnectIt implementation without sampling, our implementations are significantly faster, even without sampling. Our codes are between 1.95–21.4x faster across all low-diameter graphs, and 10.1x faster on average. Compared with the fastest ConnectIt implementation with sampling, our codes are between 9.6–30.3x faster across all low-diameter graphs, and 18.6x faster on average. Their code performs significantly worse than ours on road_usa, running 1057x slower than our fastest code. We note that their algorithm was unable to successfully run on the Twitter graph. Compared to the Slota et al. implementation, our implementation of the MultiStep approach (BFS Sampling combined with Label-Propagation) is between 3.9–22x faster than their implementation, and 12.8x faster on average. The MultiStep code implements the direction-optimization, but we observed that their BFS performance is significantly slower than BFS performance in ConnectIt, which could be due to a cache-optimized version of the edge traversal from the Graph Based Benchmark Suite used in our implementation.

Comparison with Galois [77]. The connectivity algorithm implemented in Galois uses a Label-Propagation, in conjunction with a priority-based asynchronous scheduler [77]. The algorithm is not work-efficient, and we are not aware of any theoretical upper-bounds on its work when using Galois' priority-based scheduler. Excluding the road_usa graph where their code performs very poorly using Label-Propagation, our codes without sampling are between 1.78–3.69x faster than theirs (2.36x faster on average), and our codes using sampling are between 2.84–12.6x faster than theirs (7.27x faster on average). Our codes are 2.17x faster than their Async algorithm on the road_usa graph (we chose tried other implementations from Galois and report times for the fastest one, since their Label-Propagation implementation performed poorly on this graph). Finally, their slow performance on the Twitter graph is likely due to the fact that the Galois implementation requires multiple rounds where a large fraction of the vertices are active when performing Label-Propagation.

Comparison with Patwary et al. [84]. The connectivity algorithm by Patwary et al. is a concurrent version of Rem's algorithm. The algorithm is the same as the Union-Rem-Lock implementation in our framework. However, we find that our implementations are significantly faster than theirs, and that both of our implementations of Rem's algorithms are competitive with, and often outperform theirs. Our fastest implementations without sampling achieve between 1.27–2.87x speedup over their implementation (1.99x speedup on average), and our fastest implementations with sampling achieve between 2.43–6.28x speedup over their implementation (4.4x speedup on average).

We note that they also provide a lock-free version of Rem's algorithm, but we observed that this algorithm does not always produce correct answers, and thus do not report results for it.

Comparison with GAPBS [11]. Finally, we compare ConnectIt with the Shiloach-Vishkin and Afforest implementations from the GAP Benchmark Suite (GAPBS). The Shiloach-Vishkin implementation from GAPBS suffers from an implementation issue that can cause it to perform $O(mn)$ work and $O(n)$ depth in the worst case (under an adversarial scheduler), but we note that this behavior does not seem to arise in practice. The Afforest implementation is implemented jointly with the authors of the Afforest paper by Sutton et al. [104], and was always faster than the original Afforest implementation [104]. Our fastest implementations significantly outperform their Shiloach-Vishkin implementation. Our fastest implementations without sampling are between 3.67–17.9x faster (9.55x faster on average), and our fastest implementations with sampling are between 3.67–61.2x faster (27.1x faster on average). We observe that the Afforest algorithm is sometimes faster than our algorithms without sampling (note that Afforest does perform sampling). Our fastest implementations without sampling are between 0.33–4.17x faster than their Afforest implementation (2.08x faster on average). Our fastest implementations with sampling are between 1.53–8.55x faster than their Afforest implementation (3.9x faster on average). We perform an in-depth evaluation of the Afforest sampling scheme in Section C.4.

C.3 Streaming Parallel Graph Connectivity

In this appendix, we provide deferred experimental results on our ConnectIt streaming algorithms. We provide more experimental results showing tradeoffs between batch size and throughput, and addition latency plots.

Additional Observations about Throughput Experiments. We observe that the throughput of all algorithms, even on the smallest graph that we consider is at least hundreds of millions of updates per second for all graphs. In general, the throughput appears to be higher for larger graphs, although some graphs, such as com-Orkut have very high throughput despite being quite small, which could be due to cache effects, since the vertex data fits within LLC. Another source of high performance is that the updates within a batch are left unpermuted in this experiment, and are ordered lexicographically. This ordering provides good data locality when performing work, since the consecutive edges that are applied are likely to contain vertices whose IDs are close together in the vertex ordering.

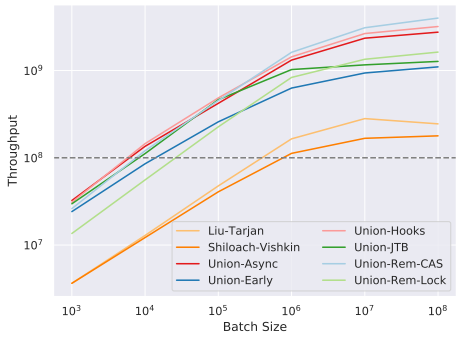
We ran an experiment to measure the effect of the edge ordering on throughput across different input graphs and found that the performance difference between the two methods scales almost linearly with the number of LLC cache misses incurred by the algorithm. For example, for the Twitter graph, applying the edges of the graph in a single batch for the Union-Rem-CAS algorithm with SplitAtomicOne with and without permuting the batch results in 3.85e9 billion and 2.23e9 billion edges per second, respectively, or a 1.72x higher throughput in the unpermuted setting. The number of LLC misses measured in these settings is 1.10B and 2.05B respectively, which is 1.8x fewer misses in the unpermuted setting. The trend is similar for other graphs, with the performance gap explained by the larger number of cache misses in the permuted setting. We feel that enforcing that batches are permuted is somewhat unrealistic, since our expectation is that real-world data streams possess a large degree of locality (intuitively many of the same vertices are updated, and updates to the same vertex are likely to occur in close succession).

Throughput vs. Batch Size. Figure 16 shows the throughput obtained by our algorithms as a function of the batch size for a subset of our input graphs (the trends for graphs not shown in the figure are very similar). We observe that all ConnectIt algorithms achieve consistently high throughput on very small to very large batch sizes. Our fastest union-find algorithms achieve a throughput of over 10M updates per second at a batch size of 100, and exceed 100M updates per second for batches of size 1000 on all graphs.

Mixed Inserts and Queries Throughput. In these experiments we study how the ratio of insertions to queries affects the overall throughput. We use the Union-Rem-CAS algorithm as a case study for this experiment, and evaluate the performance of different variants of this algorithm, varying the ratio of insertions to updates. To generate an update stream with an insert-to-query ratio of $x\%$, we generate $1/x$ many queries on random pairs of vertices, and create a batch permuting the original updates and queries together. Another alternate experiment would be to fix a particular batch size, and then vary the ratio of queries to updates. We tried this strategy but observed that this has the undesirable effect of varying the number of connected components, which affects both query and update performance and thus makes it difficult to interpret the results. Instead, we choose to fix the updates, and generate random pairs of vertices to use as queries. Unlike the experiments in the previous section, we randomly permute the batch so as to accurately reflect the cost of the Union-Rem-CAS variants which must reorder a batch for correctness.

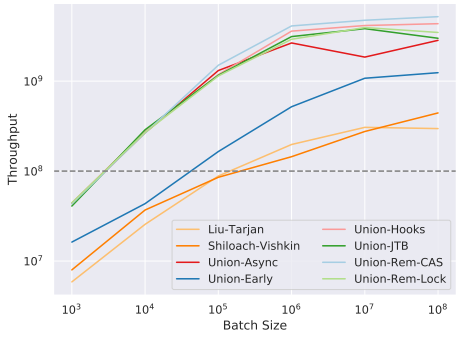
Figure 17 shows the result of this experiment on com-Orkut and LiveJournal graphs (the results for other graphs are similar). We observe that the throughput of the algorithms is higher when the ratio between the inserts-to-queries is small, and gradually drops as the ratio approaches 1. We also observe that the ordering of the algorithms (the relative throughputs) changes as the ratio of inserts-to-queries changes. When there are very few inserts, and mostly queries, the fastest algorithms are those that perform additional compression within the query, since queries can help subsequent queries save work by compressing the paths traversed in the union-find trees. On the other hand, when the ratio of inserts-to-queries is large, the algorithms that perform best are those which perform no additional compression during the finds. On both graphs, after a ratio of about 0.6–0.7, the fastest algorithm is the variant which performs no additional compression on a find, and uses the SplitAtomicOne option (like in the static setting).

Additional Latency Plots. Figure 18 shows the latency of different ConnectIt implementations while processing an update-only sequence containing a the 10% sample of the Hyperlink2012 graph. We observe that the latency for all of our algorithms is highly consistent, with the median time per batch for different batch sizes always being within 1–2% of the average batch time across all algorithms and graphs. The update time, which is plotted in log-scale, grows linearly with the batch size which is also linearly increasing in log-scale. Since the total time taken to process the graph at different

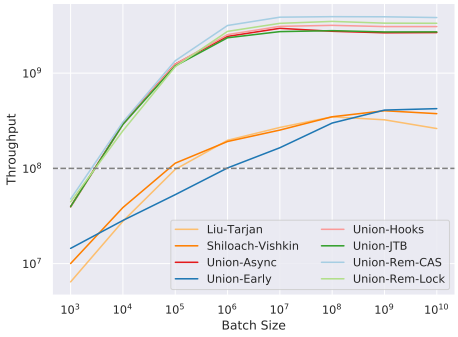


(a) Throughput vs. batch-size for road_usa

(b) Throughput vs. batch-size for Orkut



(c) Throughput vs. batch-size for LiveJournal



(d) Throughput vs. batch-size for Twitter

Figure 16: Throughput vs. batch size of different union-find implementations on different graphs. The experiment is run on a 72-core machine with 2-way hyper-threading enabled.

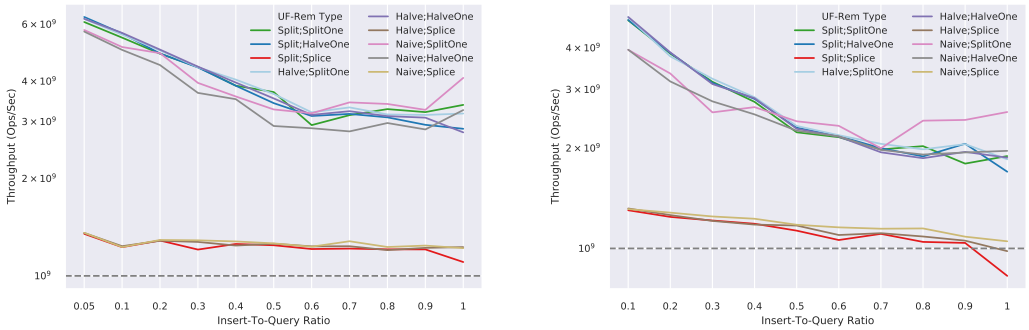


Figure 17: Throughput of different Union-Rem-CAS implementations on the com-Orkut and LiveJournal graphs, plotted against the ratio of insertions to queries in a batch. The experiment is run on a 72-core machine with 2-way hyper-threading enabled.

batch sizes is different, we rescale the x-axis for each experiment to lie in the range $[0, 1)$, and subsample the streams to plot a manageable number of operations. We plot the operations as occurring at their start time.

C.4 Sampling Evaluation

We now discuss the empirical performance of the sampling methods presented in this paper. We focus on analyzing (i) the running time of each sampling method; (ii) the fraction of the vertices covered by the largest component the sampling method identifies; and (iii) the number of inter-component edges remaining after running the sampling method.

BFS Sampling and LDD Sampling Performance. We start with the closely related BFS Sampling and LDD Sampling schemes. Table 6 reports the running times and sampling quality of these two schemes. For LDD Sampling, we report the results using the fixed parameters that are used in our experiments ($\beta = 0.2$, and setting permuting to False).

Perhaps as expected, the running time of BFS Sampling is high on the high-diameter road_usa graph, although it completely covers this graph achieving 100% coverage, with no inter-component edges remaining. For the remaining graphs, the running times are reasonable, and a BFS from a random source covers a large portion of the

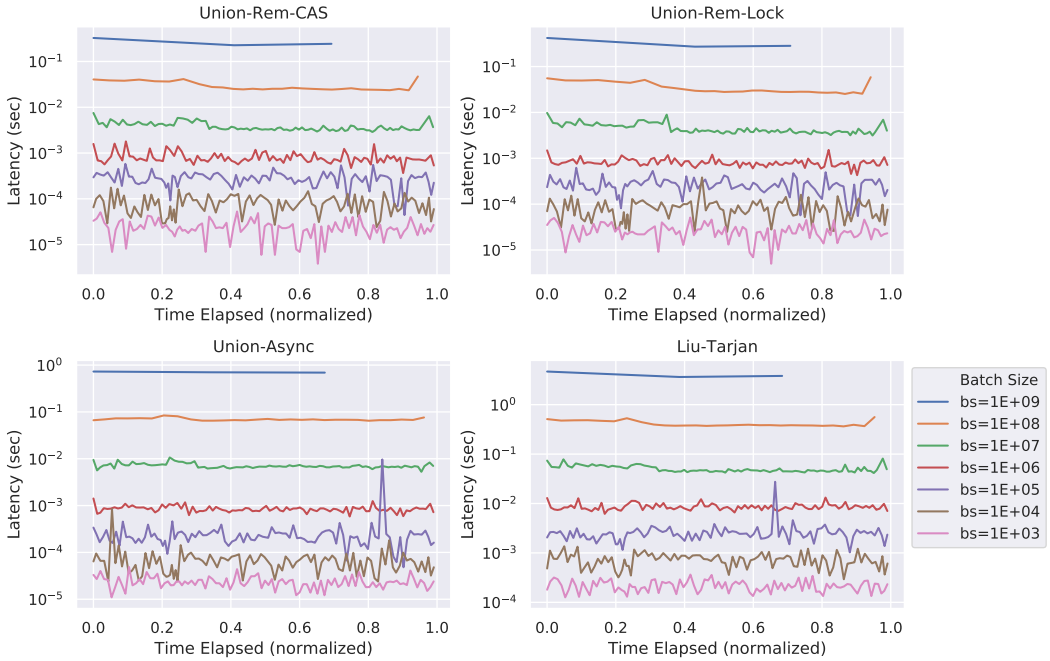


Figure 18: Latency of batch operations (seconds) plotted for varying batch sizes for the Union-Rem-CAS, Union-Rem-Lock, Union-Async, and Liu-Tarjan algorithms on the Hyperlink2012 graph. The running time of each operation sequence is normalized by the duration of the experiment at that batch size. The y-axis is in log-scale. All experiments are run on a 72-core machine with 2-way hyper-threading enabled.

Graph	BFS (s)	BFS Cov.	BFS IC.	LDD (s)	LDD Cov.	LDD IC.
road_usa	2.67	100%	0%	0.0743	100%	0%
LiveJournal	0.0109	99.9%	0.0129%	0.0136	99.9%	0.0129%,
com-Orkut	0.00909	100%	0%	0.00823	100%	0%
Twitter	0.0868	100%	0%	0.117	100%	0%
Friendster	0.330	100%	0%	0.3266	29.0%	43.9%
ClueWeb	2.04	97.6%	0.161%	1.523	97.6	0.161%
Hyperlink2014	2.49	91.4%	0.560%	3.101	91.4%,	0.530%,
Hyperlink2012	8.42	93.9%	0.538%	7.586	93.9%	0.483%,

Table 6: Performance of the BFS Sampling and LDD Sampling schemes on graphs used in our experiments. The **Cov** columns report the percentage of vertices captured in most frequent component captured by the sampling method. The **IC.** columns report the fraction of inter-component edges remaining after applying the sampling method. All times reported are measured on a 72-core machine (with 2-way hyper-threading enabled).

Graph	KOut(Hybrid) (s)	KOut(Hybrid) Cov.	KOut(Hybrid) IC.
road_usa	0.0267	94.4%	0.505%
LiveJournal	8.82e-2	99.9%	4.20e-4%
com-Orkut	8.574e-2	100%	0%
Twitter	0.112	99.9%	5.66e-3%
Friendster	0.274	52.7%	8.39e-4%
ClueWeb	2.11	94.8%	0.735%
Hyperlink2014	3.31	89.9%	0.0323%
Hyperlink2012	7.79	91.5%	0.0857%

Table 7: Performance of the k-out Sampling schemes on graphs used in our experiments. The **Cov** column reports the percentage of vertices captured in most frequent component captured by the sampling method. The **IC.** column reports the fraction of inter-component edges remaining after applying the sampling method. All times reported are measured on a 72-core machine (with 2-way hyper-threading enabled).

graph, leaving just a fraction of a percentage of inter-component edges on four of our graphs.

For LDD Sampling the performance is quite reasonable, running in close to the running time of a BFS on each of these graphs, ranging from 35.9x faster on road_usa, to at most 1.24x slower on Hyperlink2014. LDD Sampling achieves an average speedup of 5.35x over BFS Sampling, running between 1.39x slower to 1.34x faster, and is within 1% of the running time of BFS Sampling on average on graphs excluding road_usa. Thus, for graphs with low to moderate diameter, the performance of the two schemes is roughly equivalent. Compared to BFS Sampling, LDD Sampling achieves similar coverage of vertices in the most frequent connected component, and also cuts a similar number of inter-component edges, actually cutting 5–11% fewer edges on both the Hyperlink2014 and Hyperlink2012 graphs. However, both schemes already cut fewer than 1% of the edges, and so the difference in quality is very modest.

LDD Sampling Parameters. Next, we study how adjusting the value of β and applying a random permutation on the vertex ordering in the LDD algorithm affects both the coverage and number of inter-component edges. In theory, the β parameter in LDD Sampling controls both the strong diameter of the components, as well as the number of edges cut in the graph in expectation. In particular, for some $0 < \beta < 1$, LDD Sampling produces clusters with diameter $O(\log n/\beta)$, and cuts $O(\beta m)$ edges in expectation.

Figure 19 plots the tradeoff between different values of β and the overall running time of the algorithm on four of our input graphs, including the high-diameter road_usa graph, the Friendster social network, and the ClueWeb and Hyperlink2012 web graphs. We see that for road_usa, the running time decreases as a function of β, which is expected since larger values of β produce lower diameter

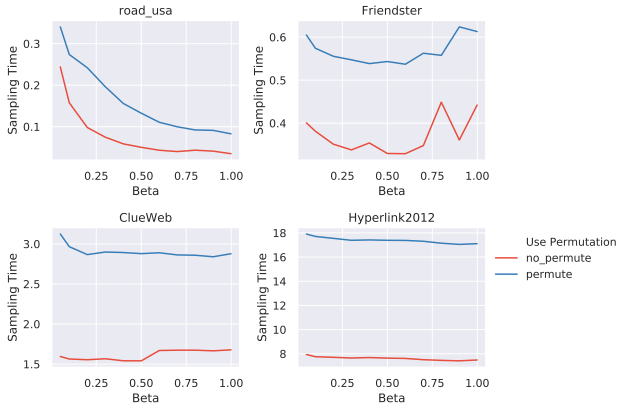


Figure 19: Plot of β vs. running time (seconds) when permuting is both enabled and disabled in LDD sampling. Experiments are run on a 72-core machine with 2-way hyper-threading.

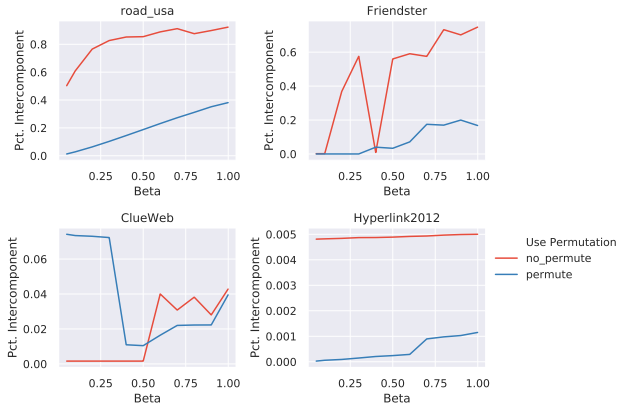


Figure 20: Plot of β vs. fraction of inter-component edges remaining after sampling when permuting is both enabled and disabled in LDD sampling.

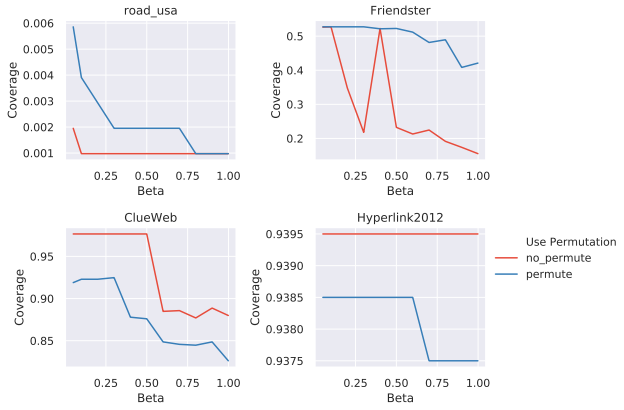


Figure 21: Plot of β vs. percentage of vertices covered in the largest component when permuting is both enabled and disabled in LDD sampling.

clusterings which in turn require fewer synchronous rounds to compute. However, higher values of β also indicate a larger number of distinct components, since more vertices wake up in early rounds and start their own clusters. This explains why the running time actually increases for large values of β on the Friendster graph.

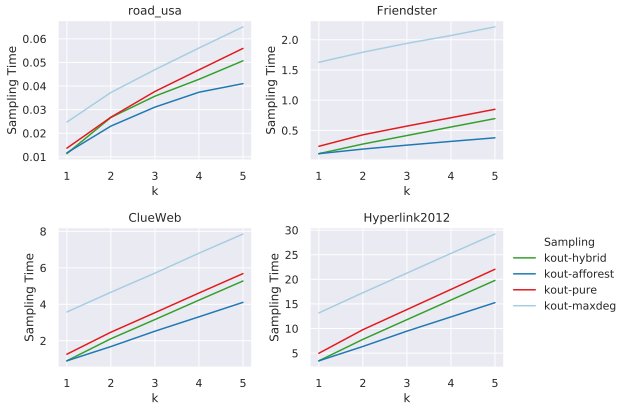


Figure 22: Plot of k vs. running time (seconds) for different variants of k-out Sampling. Experiments are run on a 72-core machine with 2-way hyper-threading.

Figure 20 plots the fraction of *inter-component edges*; letting I be the fraction of inter-component edges, the theory predicts that $I \approx \beta$. This fraction is of interest, since I is a lower-bound on the number of edges that must be processed by our connectivity algorithms, since all edges coming out of the most frequently identified component are skipped. We can see that theory is very useful in predicting the number of inter-component edges—for the road_usa graph the number of inter-component edges grows linearly with β (almost exactly following the line $.4\beta$). We also observe that the number of cut edges is not always monotonically increasing with β, which can be observed on the ClueWeb graph, although the fraction of cut edges does seem to grow as β is increased as a rule of thumb.

Lastly, Figure 21 shows the fraction of vertices covered in the most frequent component identified by LDD Sampling, for different values of β. We see that the size of this component is extremely small for the road_usa graph (less than 1%), which is consistent with what theory would predict for a high diameter graph. For lower diameter graphs, the fraction covered in the most frequent component is much higher, ranging from 80%–98% for the ClueWeb graph and 93%-94% for the Hyperlink2012 graph. Both using a random permutation and using the original vertex ordering perform well, although on the large real-world web graphs, the original ordering consistently produces a massive component that is 1–5% larger than the one produced by a random permutation of vertices.

k-out Sampling. We now turn to k-out Sampling, which is particularly interesting due to both its connection with recent fundamental theoretical results [51]. As described in Section 3.2, given an input $k \geq 1$, the algorithm implemented in this paper selects the first neighbor incident to each vertex, and selects $k-1$ other edges uniformly at random. The sampled components are computed by contracting all selected edges. We use a union-find algorithm to perform the contraction.

However, there are several other design possibilities and options in this scheme. We explore the following k-out algorithm variants in this paper:

- Afforest (kout-afforest): This scheme proposed by Sutton et al. [104] selects the first k edges incident to each vertex to use in Algorithm 4.

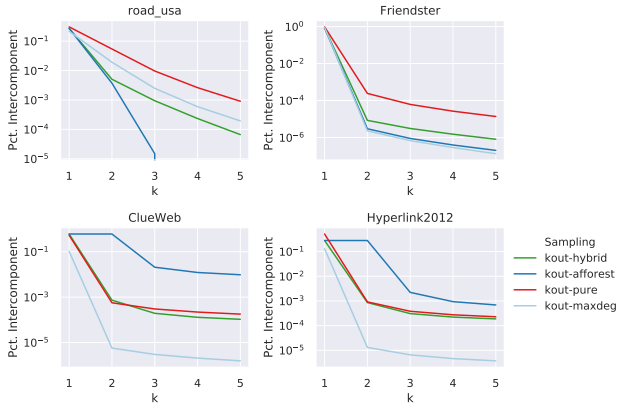


Figure 23: Plot of k vs. fraction of inter-component edges remaining after sampling for different variants of k-out Sampling.

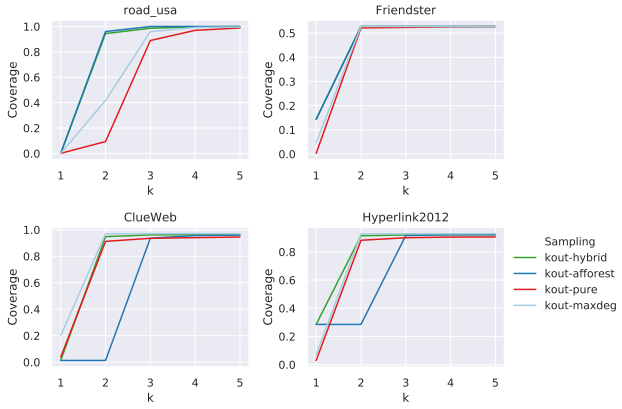


Figure 24: Plot of k vs. percentage of vertices covered in the largest component for different variants of k-out Sampling.

- Pure Sampling (kout-pure): This scheme proposed by Holm et al. [51] selects k edges incident to each vertex uniformly at random to use in Algorithm 4.
- Hybrid Sampling (kout-hybrid): This scheme selects the first edge incident to each vertex, and $k-1$ edges incident to the vertex uniformly at random to use in Algorithm 4.
- Max-Degree Sampling (kout-maxdeg): This scheme selects the edge corresponding to the largest degree neighbor incident to each vertex, and $k-1$ edges other incident to the vertex uniformly at random to use in Algorithm 4.

Both the kout-hybrid and kout-maxdeg schemes are new strategies proposed in this paper. We are not aware of any experimental evaluation of the kout-pure scheme proposed by Holm et al. [51] prior to our work. The result Holm et al. provide for kout-pure is as follows:

THEOREM 8. *Let G be an undirected n-vertex graph and let $k \geq c \log n$, where c is a sufficiently large constant. Let G' be a random k-out subgraph of G. Then the expected number of edges in G that connect different connected components of G' is $O(n/k)$.*

k-out Sampling Running Time. Figure 22 shows the running times of the different strategies on four of our graph inputs. We observe that the kout-maxdeg strategy consistently requires the largest running times, since it requires reducing over all edges incident to the vertex, and performing an indirect read on the neighboring endpoint to find the largest degree neighbor. The other schemes all have a similar cost, with kout-afforest being the fastest across all graphs, and kout-pure being the second slowest across all graphs. As expected, kout-hybrid's performance interpolates that of kout-hybrid and kout-pure, and is equal to the performance of kout-afforest for $k = 1$. This is expected, since the hybrid algorithm is the same as kout-afforest for $k = 1$, and selects $k-1$ random edges, thus approximating the performance of kout-pure for larger k.

k-out Sampling Inter-Component Edges. Figure 23 plots the fraction of *inter-component edges* generated by different variants of the k-out Sampling scheme as a function of k. We plot the fraction of inter-component edges in log-scale to help interpret the plot. We see that using $k = 1$ performs very poorly for all schemes, with the exception of the kout-maxdeg scheme, where it achieves reasonable performance for the ClueWeb and Hyperlink2012 graphs (both of these graphs are power-law degree distributed, and have several very high degree vertices). The kout-afforest scheme performs very well on the road_usa graph, probably due to the fact that most vertices in this graph have just a few neighbors—for $k \geq 4$ the kout-afforest sampling scheme finds all components of this graph and thus the inter-cluster edge percentage is 0 for $k \geq 4$. For the other graphs, the kout-maxdeg scheme consistently performs best.

The kout-pure and kout-hybrid schemes perform similarly, with the kout-hybrid scheme usually performing slightly better when the kout-afforest algorithm performs well due to the graph's vertex ordering. On the other hand, for the ClueWeb and Hyperlink2012 graphs, the kout-afforest scheme performs quite poorly—these appear to be instances where greedily taking the first k vertices based on the input vertex order results in the graph clustering into local clusters, with many inter-cluster edges between them. The vertex orders of these graphs are based on the lexicographical ordering of pages on the Web, which orders web pages within the same domain consecutively. Therefore, one explanation for this behavior is that the first k edges are all "inter-domain" edges which result in clusters based on domains being formed by the sampling algorithm.

k-out Sampling Coverage. Next, Figure 24 plots the fraction of vertices contained in the most frequent component identified by sampling as a function of k. The results are similar to those previously analyzed from Figure 23: on some graphs with good vertex orderings (road_usa, Friendster), kout-afforest achieves excellent performance. However, on other graphs (ClueWeb, Hyperlink2012) kout-afforest achieves very poor performance due to the ordering containing significant local structure that prevents large components from forming. The kout-pure algorithm addresses this issue, and achieves consistent performance on all graphs. The kout-hybrid algorithm further improves the performance of kout-pure, significantly outperforming kout-afforest on the ClueWeb and Hyperlink2012 graphs for $k = 2$, and achieving performance close to kout-afforest on the other graphs. Table 7 shows the running times, percent coverage, and percent of inter-component edges remaining for the k-out Sampling scheme used in our paper ($k = 2$ using the kout-hybrid algorithm).

k-out Sampling Summary. Finally, we consider how the performance of these schemes compares with the predictions of the theory. First, we note that the theory rules out the possibility of Theorem 8

holding for $k = 1$ (however, it is conjectured to hold for $k \geq 2$). We see that in practice for $k \geq 2$ a huge fraction of the edges become intracluster, with very few remaining inter-cluster edges, and far fewer edges remain than predicted by theory. For example, for the Hyperlink2012 graph, the kout-pure scheme results in 0.0009149% inter-component edges, or 206M inter-component edges, which is about 1/20th the number of vertices in this graph. Our results suggest that the empirical performance of these schemes is far better than suggested by theory. It would be interesting to explore applying these ideas to speed up connectivity algorithms in other settings, such as practical distributed connectivity implementations, or possibly even in practical dynamic connectivity algorithms.

C.4.1 Comparison with MapEdges and GatherEdges. Next, we compare the cost of our connectivity algorithms with two basic graph processing primitives, MAPEDGES and GATHEREDGES. These are primitives that map over all vertices in parallel, and perform a parallel reduction over their incident edges, storing the result in an array proportional to the number of vertices. MAPEDGES returns a value of 1 per edge, which corresponds to computing the degree of each vertex after reading every edge. GATHEREDGES performs an indirect read into an array at the location corresponding to the neighbor id, and so the performance of this primitive depends on the structure and ordering of the input graph.

Both primitives are useful baselines for understanding the performance of more complex graph algorithms. In particular, MAPEDGES corresponds to the cost of reading and processing a graph input, and storing some output value for each vertex. GATHEREDGES serves as a useful practical lower-bound on the performance of graph algorithms that perform indirect accesses over every edge. This is useful for connectivity algorithms, since every connectivity algorithm that always outputs a correct answer (Las Vegas algorithm) must check some connectivity information for both endpoints of an edge, which seems impossible to do without performing an indirect read[3]. All connectivity algorithms in our framework perform an indirect read across every edge to the parent array of the neighboring vertex.

Table 8 shows the running time of the MAPEDGES and GATHEREDGES primitives, and the time taken by the fastest connectivity implementation in CONNECTIT for that graph. First, we observe that there is an order of magnitude difference in running times between the MAPEDGES and GATHEREDGES: GATHEREDGES is 13.5x slower on average than MAPEDGES across all inputs. The difference between the two is more noticeable for small graphs, since the sequential reads used in MAPEDGES are likely to be served by cache. On the other hand, for GATHEREDGES, even our smallest graph has 23.9M vertices, whose vertex data is already larger than our multicore machine's LLC, and thus many of the indirect reads across edges will be cache misses.

C.5 Algorithms for Massive Web Graphs

Table 1 in the introduction presents the results of comparing CONNECTIT algorithms with these existing state-of-the-art results from different computational paradigms. We can see that in all cases, CONNECTIT achieves significant speedup, sometimes ranging on orders of magnitude in terms of raw running time alone. We now provide a detailed breakdown of how CONNECTIT algorithms compare with the fastest systems from different categories.

[3]One exception is for n vertex graphs with more than $\binom{n-1}{2}$ edges, in which case it is clear the graph is guaranteed to be connected.

Graph	MAPEDGES	GATHEREDGES	(No Sample)	CONNECTIT (Sample)
road_usa	5.54e-3	1.33e-2	2.80e-2	3.77e-2
LiveJournal	1.31e-3	8.59e-3	1.27e-2	8.96e-3
com-Orkut	9.77e-4	1.65e-2	1.91e-2	8.56e-3
Twitter	2.63e-2	0.488	0.316	9.24e-2
Friendster	2.77e-2	1.50	0.902	0.183
ClueWeb	0.790	2.77	4.04	1.69
Hyperlink2014	1.36	4.14	6.64	2.83
Hyperlink2012	2.96	10.4	13.9	8.20

Table 8: 72-core with hyper-threading running times (in seconds) comparing the performance of GATHEREDGES to the performance of the fastest connectivity implementation in CONNECTIT without sampling (No Sample), and with sampling used (Sample).

Distributed Memory Systems. There have been various distributed-memory connected components algorithms that report running times for Hyperlink2012, which we compare with here. Slota et al. [99] present a distributed-memory connectivity algorithm, and report a running time of *63s* for finding the largest connected component on the Hyperlink2012 graph using 8192 cores (256 nodes each with two 16-core processors) on the Blue Waters supercomputer. Stergiou et al. [101] describe a distributed-memory connected components algorithm. Using 12000 cores (1000 nodes each with 6 cores) and 128TB of RAM on a proprietary cluster at Yahoo!, they report a running time of *341s* for this graph. Dathathri et al. [28] report a running time of *75.3s* for connected components in the D-Galois graph processing system for Hyperlink2012 using 256 nodes (each with a KNL processor with 68 cores and 96GB RAM) on the Stampede KNL Cluster. Very recently, Zhang et al. [113] present a variant of Shiloach-Vishkin for distributed-memory and report a running time of *30s* on 262,000 cores of an XC40 supercomputer. Compared with these results, our fastest times are significantly faster.

- Compared to Slota et al. [99] our results are 7.68 times faster, using a machine with 16x less memory, and 56x fewer hyper-threads. Additionally, our algorithm finds all connected components, whereas their algorithm can only find the largest connected component (this is trivial to do on this graph by running a breadth-first search from a random vertex).
- Compared to Stergiou et al. [101], our results achieve 41.5x speedup over their result, using 128x less memory, and 166x fewer hyper-threads.
- Compared to the recent results of Dathathri et al. [28], our results are 9.18x faster. Furthermore, we use 24x less memory, and 483x fewer hyper-threads.
- Compared to the very recent results of Zhang et al., which is the fastest current distributed-memory running time to solve connectivity on this graph, our results are 3.65x faster, using at least 256x less memory (each node on their system has at least 64GB of memory, to up to 256GB of memory, but unfortunately the authors did not report these statistics), and 1819x fewer hyper-threads.

External Memory Systems. Mosaic [68] report in-memory running times on the Hyperlink2014 graph. Their system is optimized for external memory execution. They solve connected components in 700s on a machine with 24 hyper-threads and 4 Xeon-Phis (244

cores with 4 threads each) for a total of 1000 hyper-threads, 768GB of RAM, and 6 NVMes. FlashGraph [26] reports disk-based running times for the Hyperlink2012 graph on a 4-socket, 32-core machine with 512GB of memory and 15 SSDs. On 64 hyper-threads, they solve connected components in 461s.

- Compared with Mosaic, CONNECTIT is 247x faster, using 1.3x more memory, but 6.94x fewer hyper-threads. Although the Xeon-Phi chips Mosaic uses are no longer available, from a cost perspective, machines like the one we use in this paper are widely available from existing cloud service providers for a few dollars on the hour, but to the best of our knowledge, one cannot easily rent a machine like the one used in Mosaic.
- Compared with FlashGraph's result for the Hyperlink2012 graph, CONNECTIT is 56x faster, but uses 1.95x more memory, and 2.25x more hyper-threads. Therefore, we achieve orders of magnitude more performance, using comparatively little extra resources, although the comparison is not entirely fair, since FlashGraph must read this graph from an array of SSDs. It would be interesting for future work to evaluate whether our union-find algorithms can be used as part of a "semi-external" memory connectivity algorithm that streams the edges of the graph from SSD, and whether such an approach would improve on existing external and semi-external memory results.

Existing Shared-Memory (DRAM and NVRAM) Systems. Finally, we compare with several recent results showing that the Hyperlink2012 graph can be processed in the main memory of a single machine. To the best of our knowledge, the first such result is from Dhulipala et al. [31], who showed a running time of 38.3 seconds for this graph on a single-node system with 1TB of memory and 144 hyper-threads. The authors later presented an improved result, showing that the algorithm runs in just over 25 seconds on the same machine. We were able to reproduce this result, and obtained a running time of 25 seconds, on a similar machine for their algorithm. Subsequently, two simultaneous results achieved very high performance for processing this graph on a system with relatively little DRAM, but a significant amount of NVRAM [33, 41]. The result by Gill et al. [41] extends the Galois system to support graph processing when the graph is stored on non-volatile memory. They achieve a running time of 76 seconds on a machine with 376GB of memory, 3TB of NVRAM, and 96 hyper-threads. Dhulipala et al. [33] also describe a system for graph processing on NVRAMs based on a semi-asymmetric approach. Their system runs in 36.2 seconds on the same machine (376GB of memory, 3TB of NVRAM and 96 hyper-threads).

- Compared with GBBS, our results are 3.2x faster on an identical system. Given that this result was the fastest previous result under any computational model that we are aware of, we improve on the state-of-the-art time for processing this graph by 3.2x, while using exactly the same resources as those used by the authors of GBBS [31].
- Compared with the non-volatile memory results of Gill et al. [41] and Dhulipala et al. [33], our results for Hyperlink2012 are 9.26x faster, using a system with 2.65x more memory, and 1.5x more hyper-threads. It is an interesting question how quickly our new algorithms solve the connectivity problem when running in the system used by these authors, and also theoretically in the model proposed by Dhulipala et al. [33].

D CONNECTIT PSEUDOCODE

In this section, we provide pseudocode for methods in the CONNECTIT framework.

D.1 Sampling Schemes

In Algorithms 4, 5, and 6, we provide the deferred pseudocode for our sampling methods in CONNECTIT, and refer to the text in Section 3.2 for the descriptions.

D.2 Union Find Algorithms

In this section, we provide the pseudocode for all of our union-find implementations for the sake of completeness. Algorithm 7 shows our generic union-find template algorithm, which takes a custom union operator, find operator, and splice operator, and runs the resulting algorithm combination. Note that the splice operator is only valid for Rem's algorithms. Algorithm 8 provides pseudocode for the different find implementations. Algorithm 9 provides pseudocode for different splice options.

Next, we provide pseudocode for each of the union-find implementations in this paper, other than Union-JTB, whose pseudocode is presented in [59].

- Union-Async (Algorithm 10)
- Union-Hooks (Algorithm 11)
- Union-Early (Algorithm 12)
- Union-Rem-Lock (Algorithm 13)
- Union-Rem-CAS (Algorithm 14)

D.3 Shiloach-Vishkin

The pseudocode for our adaptation of the Shiloach-Vishkin algorithm is presented in Algorithm 15.

D.4 Liu-Tarjan Algorithms

Finally, we list the variants of the Liu-Tarjan framework implemented in CONNECTIT. We note that only five of these variants were explored in their original paper. We refer the reader to Section 3.3 for details about the different algorithm options in this framework, as well as the description of the overall framework.

(1) CUSA: {Connect, Shortcut, Alter}
(2) CRSA: {Connect, RootUp, Shortcut, Alter}
(3) PUSA: {ParentConnect, Shortcut, Alter}
(4) PRSA: {ParentConnect, RootUp, Shortcut, Alter}
(5) PUS: {ParentConnect, Shortcut}
(6) PRS: {ParentConnect, RootUp, Shortcut}
(7) EUSA: {ExtendedConnect, Shortcut, Alter}
(8) EUS: {ExtendedConnect, Shortcut}
(9) CUFA: {Connect, FullShortcut, Alter}
(10) CRFA: {Connect, RootUp, FullShortcut, Alter}
(11) PUFA: {ParentConnect, FullShortcut, Alter}
(12) PRFA: {ParentConnect, RootUp, FullShortcut, Alter}
(13) PUF: {ParentConnect, FullShortcut}
(14) PRF: {ParentConnect, RootUp, FullShortcut}
(15) EUFA: {ExtendedConnect, FullShortcut, Alter}

Algorithm 4 k-out Sampling

procedure kOUTSAMPLE($G(V, E)$, *labels*, k)
  edges ← { First edge from each vertex } ∪ {Sample $k - 1$ edges uniformly at random from each vertex}
  UNIONFIND(*edges*, *labels*)
  Fully compress the components array, in parallel.
  return *labels*

Algorithm 5 Breadth-First Search Sampling

procedure BFS SAMPLING($G(V, E)$, *labels*, $c = 5$)
  for $i \in [c]$ **do**
    $s \leftarrow$ RANDINT() mod $|V|$
    labels ← LABELSPREADINGBFS(G, s) ▷ Runs direction-optimizing BFS from s. *labels*$[u] = s$ if u is reachable from s, and *labels*$[u] = u$ otherwise.
    freq ← IDENTIFYFREQUENT(*labels*)
    if *freq* makes up more than 10% of labels **then**
      return *labels*
  return $\{i \to i \mid i \in [1, |V|]\}$ ▷ Return the identity labeling

Algorithm 6 Low-Diameter Decomposition Sampling

procedure LDD SAMPLING($G(V, E)$, *labels*)
  clusters ← LDD(G, $\beta = 0.2$)
  return *labels*

(16) EUF: {ExtendedConnect, FullShortcut}

Algorithm 7 Connectivity(Union-Find)

```
FindOption = {                                          ▷ Shared by UnionFind Algorithms
    FindNaive,                                          ▷ No path compression
    FindAtomicSplit,                                    ▷ Path splitting
    FindAtomicHalve,                                    ▷ Path halving
    FindCompress,                                       ▷ Full path compression
}
SpliceOption = {                                        ▷ Used by Rem's algorithms when path has not yet reached a root
    SplitAtomicOne,                                     ▷ Performs one path split
    HalveAtomicOne,                                     ▷ Performs one path halve
    SpliceAtomic,                                       ▷ Performs a splice operation
}
UnionOption = {
    Union-Async,                                        ▷ Asynchronous union-find
    Union-Hooks,                                        ▷ Asynchronous hook-based union-find
    Union-Early,                                        ▷ Union with early-optimization
    Union-Rem-Lock,                                     ▷ Lock-based Rem's algorithm*
    Union-Rem-CAS,                                      ▷ CAS-based Rem's algorithm*
}
procedure CONNECTIVITY(G, labels, frequentid, UnionOption, FindOption, SpliceOption)
    U = Union-Find(UnionOption, FindOption, FindOption)
    parfor {u | labels[u] ≠ frequentid} do
        parfor {(u, v) ∈ d+(u)} do
            U.Union(u, v, labels)
```

Algorithm 8 Find Algorithms

```
procedure FINDNAIVE(u, P)
    v ← u
    while v ≠ P[v]* do
        v ← P[v]
    return v
procedure FINDCOMPRESS(u, P)
    r ← u
    if P[r] = r then return r
    while r ≠ P[r] do
        r ← P[r]
    while j ← P[u] > r do
        P[u] ← r, u ← j
    return r
procedure FindAtomicSplit(u, P)
    v ← P[u], w ← P[v]
    while v ≠ w* do
        CAS(&P[u], v, w)
        u ← v
    return v
procedure FindAtomicHalve(u, P)
    v ← P[u], w ← P[v]
    while v ≠ w* do
        CAS(&P[u], v, w)
        u ← P[u]
    return v
```

Algorithm 9 Splice Algorithms

```
1: procedure SplitAtomicOne(u, x, P)
2:     v ← P[r_u], w ← P[v]*
3:     if v ≠ w then
4:         CAS(&P[u], v, w)
5:     return v
6: procedure HalveAtomicOne(u, x, P)
7:     v ← P[r_u], w ← P[v]*
8:     if v ≠ w then
9:         CAS(&P[u], v, w)
10:    return w
11: procedure SpliceAtomic(u, v, P)
12:    p_u = P[u]*
13:    CAS(&P[u], p_u, P[v])
14:    return p_u
```

Algorithm 10 Union-Async

```
1: FIND = one of {FindNaive, FindAtomicSplit,
2:                FindAtomicHalve, FindCompress}
3: procedure UNION(u, v, P)
4:     p_u ← FIND(u, P), p_v ← FIND(v, P)
5:     while p_u ≠ p_v do                                   ▷ WLOG, let p_u > p_v
6:         if p_u = P[p_u] and CAS(&P[u], p_u, p_v) then
7:             return
8:         p_u ← FIND(u, P), p_v ← FIND(v, P)
```

Algorithm 11 Union-Hooks

```
1: FIND = one of {FindNaive, FindAtomicSplit,
2:                FindAtomicHalve, FindCompress}
3: H ← {i → ∞ | i ∈ |V|}                                   ▷ Array of hooks
4: procedure UNION(u, v, P)
5:     p_u ← FIND(u, P), p_v ← FIND(v, P)
6:     while p_u ≠ p_v do                                   ▷ WLOG, let p_u > p_v
7:         if p_u = P[p_u] and CAS(&H[u], ∞, p_v) then
8:             P[p_u] ← p_v
9:             return
10:        p_u ← FIND(u, P), p_v ← FIND(v, P)               ▷ WLOG, let p_u < p_v
```

Algorithm 12 Union-Early

```
1: FIND = one of {FindNaive, FindAtomicSplit,
2:                FindAtomicHalve, FindCompress}
3: procedure UNION(u, v, P)
4:     p_u ← u, p_v ← v
5:     while p_u ≠ p_v do                                   ▷ WLOG, let p_u > p_v
6:         if p_u = P[p_u] and CAS(&P[u], p_u, p_v) then
7:             BREAK
8:         z ← P[p_u], w ← P[z]
9:         CAS(&P[p_u], z, w)
10:        p_u ← w
11:    FIND(u, P), FIND(v, P)                               ▷ Elided if the find-option is FindNaive
```

Algorithm 13 Union-Rem-Lock

```
1: COMPRESS = one of {FindNaive, FindAtomicSplit, FindAtomicHalve}
2: SpliceAtomic = one of {SplitAtomicOne, HalveAtomicOne, SpliceAtomic}
3: L ← mutex[|V|]                                                    ▷ Array of locks
4: procedure UNION(u, v, P)
5:     r_u ← u, r_v ← v
6:     while P[r_u] ≠ P[r_v]* do                                     ▷ WLOG, let P[r_u] > P[r_v]
7:         if r_u = P[r_u] then
8:             L[r_u].LOCK()
9:             p_v ← P[r_v]
10:            if r_u = P[r_u] and r_u > p_v* then
11:                P[r_u] ← p_v
12:            L[r_u].UNLOCK()
13:            return True
14:        else
15:            r_u ← SpliceAtomic(r_u, r_v, P)
16:    if COMPRESS ≠ FindNaive then
17:        COMPRESS(u, P), COMPRESS(v, P)
18:    return False
```

Algorithm 14 Union-Rem-CAS

```
1: COMPRESS = one of {FindNaive, FindAtomicSplit, FindAtomicHalve}
2: SpliceAtomic = one of {SplitAtomicOne, HalveAtomicOne, SpliceAtomic}
3: procedure UNION((u, v, P))
4:     r_u ← u, r_v ← v
5:     while P[r_u] ≠ P[r_v]* do                                     ▷ WLOG, let P[r_u] > P[r_v]
6:         if r_u = P[r_u] and CAS(&P[r_u], r_u, P[r_v])* then
7:             if COMPRESS ≠ FindNaive then
8:                 COMPRESS(u, P)
9:                 COMPRESS(v, P)
10:            return True
11:        else
12:            r_u ← SpliceAtomic(r_u, r_v, P)
13:    return False
```

Algorithm 15 Shiloach-Vishkin

```
1: procedure CONNECTIVITY(G, labels, frequentid)
2:     changed ← true
3:     candidates ← {v ∈ V | labels[v] ≠ frequentid}
4:     prev_labels ← parents                                         ▷ A copy
5:     while changed = true do
6:         changed = false
7:         parfor {v ∈ candidates} do
8:             parfor {(u, v) ∈ d+(u)} do
9:                 p_u = labels[u], p_v = labels[v]
10:                l = min(p_u, p_v), h = max(p_u, p_v)
11:                if l ≠ h and h = prev_labels[h] then
12:                    WRITEMIN(&labels[h], l)
13:                    changed ← true
14:        parfor {v ∈ V} do
15:            FULLSHORTCUT(v, labels)
16:            prev_labels[v] ← parents[v]
```