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

Laxman Dhulipala
MIT CSAIL
laxman@mit.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. The fastest existing multicore algorithms for solving graph 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 12.4x (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. To guide the user in selecting the best variants in ConnectIt for different situations, we provide a detailed analysis of the different strategies. Finally, we show how the techniques in ConnectIt can be used to speed up two important graph applications: approximate minimum spanning forest and SCAN clustering.

PVLDB Reference Format:
Laxman Dhulipala, Changwan Hong, and Julian Shun. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. PVLDB, 14(4): 653–667, 2021.
doi:10.14778/3436905.3436923

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 [28]. A recent paper by Sahu et al. [85] 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 [98], as well as a subroutine in popular clustering algorithms [38, 39, 41, 80, 102, 103].

In the sequential setting, connected components can be easily solved using breadth-first search, depth-first search, or union-find. However, it is important to have fast parallel algorithms for the problem in order to achieve high performance. Many parallel algorithms for connected components have been proposed in the literature (see, e.g., [4–8, 10, 15–18, 20, 22–27, 31, 40, 42, 44–48, 51, 54–58, 61–66, 68, 70, 73, 76, 77, 79, 81–84, 87, 88, 90, 93, 95–97, 100, 104, 105], among many others). Recent state-of-the-art parallel implementations are based on graph traversal [31, 88, 90, 93], label propagation [77, 88, 93], union-find [20, 57, 81], or the hook-compress paradigm [5, 7, 14, 18, 27, 44, 68, 73, 79, 87, 95, 101, 105]. Recent work by Sutton et al. [97] uses sampling to find the connected components on a subset of the edges, which can be used to reduce the number of edge inspections 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 from the literature. Furthermore, as many real-world graphs are frequently updated under insertion-heavy workloads (e.g., there are about 6,000 tweets per second on Twitter, 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 ConnectIt implementations 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, 34]. 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. We have recently extended our techniques to the GPU setting [53].

ConnectIt Overview

Algorithms. ConnectIt is designed for *min-based connectivity algorithms*, which are based on vertices propagating labels to other vertices that they are connected to, and updating labels based on the minimum label received. All of the algorithms conceptually view the label of a vertex v as a directed edge from v to the vertex

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436923

corresponding to v's label. Thus, these directed edges form a set of directed trees. All of the algorithms that we study maintain acyclicity in this forest (ignoring self-loops at the roots of trees). The min-based algorithms that we study include both *root-based algorithms*, which include a broad class of union-find algorithms and several other algorithms that only modify the labels of roots of trees in the forest, as well as *other min-based algorithms* which deviate from this rule and can modify the labels of non-root vertices.

Sampling and Two-phase Execution. Inspired by the Afforest algorithm [97], 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 [13], 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 a min-based algorithms on all vertices and edges.

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 min-based algorithms that we consider can be used for the finish phase in combination with any of the three sampling schemes: k-out, breadth-first search, and low-diameter decomposition sampling. For the implementations where the finish phase uses 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 (only our root-based algorithms support spanning forest).

Incremental Connectivity. Due to the frequency of updates to graphs, various parallel streaming algorithms for connected components have been developed [1, 32, 37, 72, 86, 92]. 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 the variants of Rem's algorithm combined with the splice compression scheme, support asynchronous updates and queries, and most of them are lock-free or wait-free.

Experimental Evaluation. We conduct a comprehensive experimental evaluation of all of the connectivity and spanning forest implementations in ConnectIt on a 72-core multicore machine, in both the static and incremental setting. Compared to existing work, the 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 high performance, ConnectIt is able to produce implementations that can process the Hyperlink2012 graph [74], which is the largest publicly-available real-world graph and has over 3.5 billion vertices and 225 billion directed edges,[1] 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 128 billion directed 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. Finally, we show that ConnectIt can be used to speed up two graph applications: approximate minimum spanning forest and index-based SCAN clustering.

System	Graph	Mem. (TB)	Threads	Nodes	Time (s)
Mosaic [69]	Hyperlink2014	0.768	1000	1	708
FlashGraph [106]	Hyperlink2012	.512	64	1	461
GBBS [31]	Hyperlink2012	1	144	1	25.8
GBBS (NVRAM) [34]	Hyperlink2012	0.376	96	1	36.2
Galois (NVRAM) [43]	Hyperlink2012	0.376	96	1	76.0
Slota et al. [94]	Hyperlink2012	16.3	8192	256	63
Stergiou et al. [96]	Hyperlink2012	128	24000	1000	341
Gluon [29]	Hyperlink2012	24	69632	256	75.3
Zhang et al. [105]	Hyperlink2012	≥ 256	262,000	4096	30
ConnectIt	Hyperlink2014	1	144	1	**2.83**
	Hyperlink2012	1	144	1	**8.20**

Table 1: System configurations, including memory (terabytes), num. hyper-threads and nodes, and running times (seconds) of connectivity results on the Hyperlink graphs. The last rows show the fastest ConnectIt times. The fastest time per graph is shown in green.

Contributions. The contributions of this paper are as follows.

(**1**) We introduce ConnectIt, which provides several hundred different multicore implementations of 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 (in terms of directed edges) of between 4.6 million insertions per second for very small batch sizes to 7.1 billion insertions 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**) We show that ConnectIt can be used to speed up approximate minimum spanning forest by 2.03–5.36x and index-based SCAN clustering by 42.5–50.5x.
(**7**) The ConnectIt source code is available at https://github.com/ParAlg/gbbs/tree/master/benchmarks/Connectivity/ConnectIt.

2 PRELIMINARIES

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

[1] We symmetrize the graphs to obtain an undirected graph for connectivity, and when reporting the number of edges we count each edge once per direction.

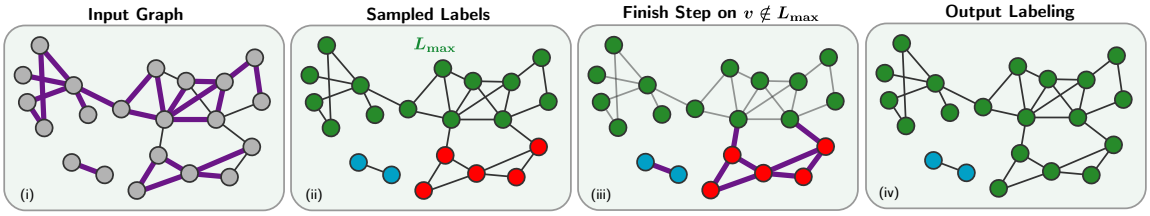


Figure 1: This figure illustrates the ConnectIt framework for connectivity using k-out Sampling on a small input graph. (i) illustrates the input graph. The bolded purple edges are those selected by the k-out sample with $k = 2$. (ii) shows the partial connectivity labeling after computing connectivity on the sampled edges (e.g., using 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.

and $m = |E|$ to refer to the number of directed 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. Unless otherwise mentioned, we store graphs in the CSR

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 by most modern processors.

Linearizability. Linearizability [49, 50] 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.

Graph Connectivity and Related Problems. 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 ***connectivity labeling*** C for each vertex, such that $C(u) = C(v)$ if and only if vertices u and v are in the same connected component. We represent connectivity labelings using an array of n integers. A ***partial connectivity labeling*** is a labeling C such that $C(u) = C(v)$ implies that u and v are in the same component. A ***connectivity query*** takes as input two vertex identifiers and returns *true* if and only if their labels are the same. A ***spanning forest (SF)*** in $G = (V, E)$ contains one tree for each connected component in G containing 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 $V_1, \ldots, V_k$, 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].

3 CONNECTIT FRAMEWORK

In this section, we define the components of the ConnectIt framework and the specific ways in which 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 1 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 incremental setting. We present correctness proofs for our algorithms in all of the settings in the full version of our paper [33]. 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:     L_max ← IdentifyFrequent(labels)
7:     labels ← finish.FinishComponents(G, labels, L_max)
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***, which is represented as an array of n integers, 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 and stored in the *labels* array. Next, it identifies $L_{\max}$, 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 $L_{\max}$.

Properties of Sampling Methods. Before discussing the sampling and finish methods, we discuss the properties that we require in ConnectIt to obtain a correct parallel connectivity algorithm.

Definition 3.1. Consider a graph G. Let C be the labeling produced by a sampling method S and let the labeling $C' = \text{Connectivity}(G[C])$ where $G[C]$ is the graph induced by contracting G using C.[2] We say that a sampling method S is ***correct*** if

(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.

In other words, the connectivity labeling can be viewed as a tree where the parent pointer of a vertex is its connectivity label. Requirement **(1)** states that these directed trees have height one—each vertex either points to itself, or points to a root vertex, r, that points to itself. Requirement **(2)** states that the connectivity label

[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.

found by composing the sampled label ($C[v]$) with the connectivity label of the contracted vertex on the contracted graph, i.e., $C'[C[v]]$, yields a correct connectivity labeling.

Properties of Finish Methods. Next, we define the correctness properties for finish methods in ConnectIt. The definition uses the interpretation of labels as rooted trees, which we discussed above.

Definition 3.2. Let C be a connectivity labeling, which initially maps every vertex to its own node ID. We call a connectivity algorithm ***monotone*** if the algorithm updates the labels such that the updated labeling can be represented as the union of two trees in the previous labeling.

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.

The finish methods considered in ConnectIt are linearizably monotone, with only a few exceptions. The first is a subset of the variants of Rem's algorithms (all variants using the *splice* rule), which we analyze separately, and the family of other min-based connectivity algorithms, which includes a subset of Liu-Tarjan algorithms, Stergiou's algorithm, Shiloach-Vishkin, and label propagation. In the full version of our paper [33], we provide proofs showing that both of these exceptions, and the large class of linearizably monotone finish methods, are correct when composed with our sampling schemes. The key idea for the proof for linearizably monotone algorithms is to apply induction over the linearization order of the operations, and apply the fact that each operation is monotone and thus preserves the partial connectivity information. The proof for the other min-based algorithms is by contradiction: at the end of the algorithm, two vertices in the same component must have the same label, as otherwise the finish algorithm will not have terminated, even if we ignore the largest sampled component.

3.2 Sampling Algorithms

This section introduces the correct sampling methods used in ConnectIt. Due to space constraints, we provide the pseudocode for our sampling methods in the full version of our paper [33].

k-out Sampling. The k-out method takes a positive integer parameter k, selects k edges out of each vertex uniformly at random, and computes the connected components of this sampled graph. An important result shown in a recent paper by Holm et al. [52] 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. Holm et al. 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. [97] 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 of magnitude 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 the full version of our paper [33].

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 $c = \Theta(\log n)$ would ensure that we find the massive component with high probability. 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. [90] give a simple and practical work-efficient (linear-work) 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. [90] on low-diameter real-world graphs and observing 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 the full version of our paper [33].

In the full version of the paper, we prove that k-out Sampling, BFS Sampling, and LDD Sampling are all correct, i.e., they all produce connectivity labeling satisfying Definition 3.1.

The Potential Benefits of Sampling. Intuitively, the main advantage of our sampling schemes is that if the graph has a single large component containing a significant fraction of the edges, applying a sampling method can allow us to skip processing most of these edges. Specifically, suppose applying a correct sampling scheme uncovers a frequent component $L_{\max}$ containing $\mathcal{X}$ edges while processing only $\mathcal{Y}$ edges. Then, by skipping processing vertices in $L_{\max}$ in the finish phase, the total number of edges that are processed by the algorithm is $m - \mathcal{X} + \mathcal{Y}$.

Figure 2 illustrates the potential benefits from applying our sampling schemes on a suite of large real-world graphs (see Section 4

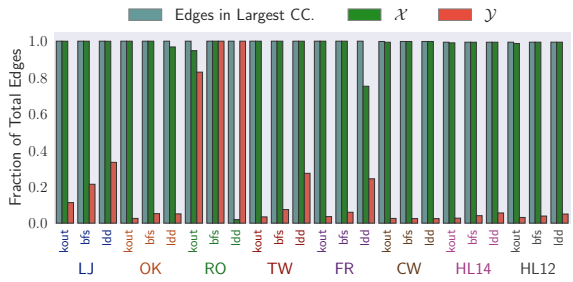


Figure 2: Bar plot showing the performance of different sampling strategies on eight large real-world graphs in terms of the number of edges in the largest connected component, the number of edges in the most frequent sampled component ($\mathcal{X}$) and the number of edges processed by the sampling strategy ($\mathcal{Y}$). All quantities are shown as a fraction of the total number of edges, m.

for graph details). We observe that $\mathcal{X}$ is usually a large fraction of both the number of edges in the largest component and of m, indicating that the sampling methods can help us skip nearly all of the edges in the graph in the finish phase. The one exception is the road_usa (RO) graph, where LDD and BFS Sampling both suffer due to the graph's large diameter, which we discuss in more detail in Section 4.1. However, both LDD and BFS Sampling perform well on the other low-diameter graphs since direction-optimization enables them to complete while only examining a small number of edges. Note that applying LDD on high-diameter graphs results in a large number of small clusters, and thus this scheme is better suited for low-diameter graphs [75]. We observe that k-out Sampling results in a large value of $\mathcal{X}$ in all cases. Finally, $\mathcal{Y}$ is typically significantly smaller than $\mathcal{X}$ indicating that our approach enables us to process $\mathcal{Y} + (m - \mathcal{X})$ edges in total, which is only a small fraction of m.

3.3 Finish Algorithms

We now describe different min-based 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 min-based algorithms that we consider as part of the framework are union-find (many different variants, described below), Shiloach-Vishkin [87], Liu-Tarjan [68], Stergiou [96], and label propagation. An important feature of ConnectIt is in modifying these finish methods to *avoid traversing* the vertices with the most frequently occurring ID as identified from sampling in Algorithm 1. We provide pseudocode for our implementations of all of our implementations described in this section in the full version of our paper [33].

3.3.1 Union-Find. We consider several different concurrent (asynchronous) union-find algorithms, which are all min-based. All of these algorithms are linearizably monotone for a set of concurrent union and find operations, with the exception of the concurrent Rem's algorithm variants using the splice rule, 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 can be combined with sampling by simply skipping traversing the vertices with label equal to $L_{\max}$ 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 [59]. 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 **UF-Async** since it is the classic union-find algorithm directly adapted for an asynchronous shared-memory setting. The algorithm links from lower-indexed to higher-indexed vertices to avoid cycles, and only performs links on roots (thus implying that the algorithm is monotone). 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 (they do not consider FindCompress, but it is relatively easy to show that it is linearizable). The fact that these algorithms are linearizably monotone 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 UF-Async algorithm: UF-Hooks and UF-Early. **UF-Hooks** is closely related to UF-Async, with the only difference between the algorithms being 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. **UF-Early** is also similar to UF-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 UF-Async by Jayanti and Tarjan [59] applies to UF-Hooks and UF-Early, and shows that both algorithms are linearizable for a set of concurrent find and union operations. Thus, these algorithms are also linearizably monotone 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 [60], and refer to this algorithm as **UF-JTB**. The algorithm either performs finds naively, without using any path compression (**FindNaive**), or uses a strategy called **FindTwoTrySplit**, which guarantees provably-efficient bounds for their algorithm, assuming a source of random bits. We refer to [60] for the pseudocode and proofs of correctness.

Concurrent Rem's Algorithm. We implement two concurrent versions of Rem's algorithm (Rem's algorithm was first published in Dijkstra's book [36]): a lock-based version by Patwary et al. [81] (**UF-Rem-Lock**) and a lock-free compare-and-swap based implementation (**UF-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 algorithm 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. The third rule performs the splicing operation described in Rem's algorithm, which atomically swaps the parent of the higher index vertex in the path to the lower index vertex. 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 algorithm that do not perform SpliceAtomic are linearizable for a set of concurrent union and find operations by simply following the proof of Jayanti and Tarjan [59]. The implementations of Rem's algorithm combined with SpliceAtomic do not satisfy linearizability of both concurrent find and union operations. In the full version of our paper, we show that the algorithm is correct in the *phase-concurrent* setting, where union and find operations are separated by a barrier.

We note that a recent paper by Alistarh et al. [2] performed a careful 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. Compared to their lock-free Rem's implementation, our implementation is more general, allowing the algorithm to be combined with path-halving and splicing in addition to path-splitting.

3.3.2 Other Min-Based Algorithms. Lastly, we overview the other min-based algorithms supported by ConnectIt.

Liu-Tarjan's Algorithms. Recently, Liu and Tarjan present a framework for simple concurrent connectivity algorithms based on several rules that manipulate an array of parent pointers using edges to transmit the connectivity information [68]. These algorithms are not true concurrent algorithms, but really parallel algorithms designed for the synchronous Massively Parallel Computation (MPC) setting. We implement the framework proposed in their paper as part of ConnectIt. 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. On 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, the shortcut phase performs path compression, and the optional alter phase updates the endpoints of an edge to be the current labels of its endpoints. We provide details of the different options for implementing each phase in the full version of our paper [33].

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. We evaluate all algorithm combinations that are expressible in the Liu-Tarjan framework, which we list in the full version of our paper [33]. Note that only the root-based algorithms in the Liu-Tarjan framework are linearizably monotone. The remaining algorithms are not monotone since a non-root vertex can be moved to a different subtree (one where the previous tree and new trees are disconnected). The non-monotone algorithms result in correct connectivity algorithms due to the fact that edges which were previously applied continue to be applied in subsequent rounds of the algorithm [68].

Stergiou et al.'s Algorithm. Stergiou et al. [96] recently proposed a min-based connectivity algorithm for the massively parallel computation setting, which is not monotone. 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 [68].

Shiloach-Vishkin and Label Propagation. ConnectIt also includes the classic Shiloach-Vishkin (SV) algorithm, which is linearizably monotone, and the folklore label propagation (Label-Prop) algorithm, which is not monotone, both of which we discuss in the full version of the paper [33].

Sampling for Other Min-Based Algorithms. Lastly, we describe how to combine the other min-based algorithms above which are *not monotone* with the sampling algorithms in ConnectIt. If the largest component after sampling, which has ID $L_{\max}$, 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 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 [68]. We provide a detailed proof in the full version of the paper [33].

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 are not root-based, and Stergiou's algorithm. We defer our description to the full version of our paper [33].

3.5 Streaming

We now discuss how ConnectIt supports streaming graph connectivity in the parallel batch-incremental and wait-free asynchronous settings. Formally, an algorithm in the ***parallel batch-incremental*** streaming setting receives a sequence of *batches* of operations, where each batch consists of ***Insert***(u, v) operations and ***IsConnected***(u, v) queries. The algorithm must process the batches one after the other, but operations within a batch are not ordered, and so the streaming algorithm can use parallelism to accelerate processing a batch. We also consider the stronger asynchronous setting. Formally, in the ***wait-free asynchronous*** streaming setting, operations are not presented to the algorithm as batches, but instead the Insert and IsConnected operations can be called concurrently by a set of asynchronous threads [49]. Note that any wait-free asynchronous algorithm can be easily extended to a parallel batch-incremental algorithm by simply invoking the concurrent implementation on every operation in parallel. ConnectIt supports the following types of algorithms in the streaming setting:

(1) The union-find algorithms in Section 3.3.1, excluding Rem's algorithms with the SpliceAtomic method. These algorithms are linearizably monotone in both the parallel batch-incremental and wait-free asynchronous settings.
(2) Shiloach-Vishkin (SV) and the root-based Liu-Tarjan algorithms. These algorithms are linearizably monotone in the parallel batch-incremental setting, although we show that IsConnected queries can be applied concurrently in the wait-free asynchronous setting. However, in these algorithms insertions cause edges to be processed multiple times until convergence, so they must be processed in batches.
(3) UF-Rem-CAS and UF-Rem-Lock using SpliceAtomic. For this class of algorithms, we consider the *phase-concurrent* setting which lies between the wait-free asynchronous and parallel batch-incremental settings. Essentially, in this setting, we have a synchronous barrier between insertion and query phases, but within a phase, operations can be called concurrently, like in the wait-free asynchronous setting.

Due to space constraints, we provide details about our streaming algorithm in ConnectIt in our full paper [33].

3.6 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. We note that our code can be easily extended to the wait-free asynchronous setting.

We use the compression techniques provided by Ligra+ [31, 91] 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. The compression scheme uses difference-encoding for each vertex's adjacency list, storing the differences using variable-length byte codes. This compression scheme is supported by the Graph Based Benchmark Suite (GBBS) [31, 35], and we used this code base as the basis for implementing ConnectIt.

4 EVALUATION

Overview. We show the following results in this section:
- Without sampling, the UF-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).

Graph	n	m	Diam.	Num C.	Largest C.	LT-DC (s)	LT (s)
RO	23.9M	57.7M	6,809	1	23.9M	0.108	0.241
LJ	4.8M	85.7M	16	1,876	4.8M	0.101	0.226
CO	3.1M	234.4M	9	1	3.1M	0.094	0.520
TW	41.7M	2.4B	23*	1	41.7M	0.115	2.80
FR	65.6M	3.6B	32	1	65.6M	0.182	6.07
CW	978.4M	74.7B	132*	23.7M	950.5M	0.534	54.2
HL14	1.7B	124.1B	207*	129M	1.57B	1.02	101.3
HL12	3.6B	225.8B	331*	144M	3.35B	1.64	192.5

Table 2: Graph inputs, including vertices, directed edges, graph diameter, the number of components (Num C.), the number of vertices in the largest component (Largest C.), and the time required in seconds to load the graph input in the binary CSR format used in GBBS, assuming that the graph is already in the disk-cache (LT-DC). The loading time (LT) is the time to fully read the input graph from a 1TB *Samsung-SM961* SSD. (The focus of this paper is not on accelerating this hardware-dependent loading time, but on accelerating connected components in the in-memory setting where the data is already in the disk-cache.) 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.

- With sampling, the UF-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 directed 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 algorithm from STINGER, an existing state-of-the-art graph streaming system, by between 1,461–28,364x (Section 4.4).
- An evaluation of ConnectIt's fastest algorithms on synthetic networks, and guidelines for selecting sampling and finish methods based on graph propertries (Section 4.5).

We show the additional experimental results in our full paper [33]:
- 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.
- The fastest ConnectIt algorithms with sampling are as fast as or faster than basic graph routines that perform one indirect read per 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.
- The trends for spanning forest are similar to connectivity, and on average the additional overhead needed to obtain the spanning forest compared to connectivity is 23.7%.

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 [19]. 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 (RO)*** is an undirected road network from the DIMACS challenge [30]. ***LiveJournal (LJ)*** is a directed graph of the LiveJournal social network [21]. ***com-Orkut (CO)*** is an undirected graph of the Orkut social network. ***Twitter (TW)*** is a directed graph of the Twitter network [67]. ***Friendster (FR)*** is an undirected graph describing friendships from a gaming network. ***ClueWeb (CW)*** is a directed Web graph from the Lemur project at CMU [21]. ***Hyperlink2012 (HL12)*** and ***Hyperlink2014 (HL14)*** are directed hyperlink graphs obtained from the WebDataCommons dataset where vertices represent Web pages [74]. 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 [96], we symmetrize them before applying our algorithms and 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 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 UF-Rem-CAS is consistently the fastest finish algorithm. We find that the fastest variant of UF-Rem-CAS across all graphs uses the FindNaive option for find (i.e., it does not perform any extra compression after the union operation), and most frequently uses the SplitAtomicOne option to perform path compression during the union operation. We observe that the running times of the other splice options, SpliceAtomic and HalveAtomicOne, are almost identical to SplitAtomicOne.

The UF-Hooks algorithm also achieves high performance, and is between 1.20–1.49x slower than UF-Rem-CAS across all graphs (1.35x slower on average). The UF-Async algorithm also achieves consistent high performance, and is between 1.34–1.96x slower than UF-Rem-CAS across all graphs (1.62x slower on average). UF-JTB is consistently slower than UF-Rem-CAS, being between 2.46–5.56x slower across all graphs, and 4.09x slower on average. The fastest find option for UF-JTB in this setting was always the two-try splitting option, with the exception of the road_usa graph where FindNaive was slightly faster. We discuss the source of performance differences between these algorithms in Section 4.1.1.

Compared to the union-find algorithms, the Liu-Tarjan algorithms are much slower on our inputs. The fastest Liu-Tarjan variant is still 2.64–10.2x slower than UF-Rem-CAS (6.74x slower on average). Stergiou's algorithm was always slower than the fastest variant from the Liu-Tarjan framework. Finally, our implementation of SV is between 2.98–7.62x slower than UF-Rem-CAS (5.14x slower on average). The performance of Label-Prop is between 3.11–4.92x slower on all graphs except for road_usa. On road_usa, its performance is 478x worse than that of UF-Rem-CAS because it requires a large number of rounds where most vertices are active due to the high diameter of the graph.

Grp.	Algorithm	RO	LJ	CO	TW	FR	CW	HL14	HL12
No Sampling	UF-Early	3.61e-2	3.48e-2	8.63e-2	2.52	1.50	59.8	17.0	32.9
	UF-Hooks	3.37e-2	1.75e-2	2.69e-2	0.390	1.17	6.05	9.37	20.0
	UF-Async	4.02e-2	2.03e-2	3.12e-2	0.426	1.21	7.92	12.2	25.5
	UF-Rem-CAS	**2.80e-2**	1.27e-2	1.91e-2	0.316	0.902	4.04	6.64	13.9
	UF-Rem-Lock	5.07e-2	1.95e-2	2.84e-2	0.437	1.23	5.64	9.20	19.3
	UF-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
	SV	0.138	4.34e-2	5.70e-2	1.65	5.38	21.2	38.5	106
	Label-Prop	13.4	4.66e-2	6.37e-2	1.24	4.37	13.4	20.7	46.5
k-out Sampling	UF-Early	3.25e-2	9.00e-3	8.61e-3	0.117	0.227	2.28	4.77	8.94
	UF-Hooks	3.62e-2	9.18e-3	9.16e-3	0.121	0.230	2.22	3.63	8.51
	UF-Async	3.33e-2	8.97e-3	**8.56e-3**	0.117	0.228	2.21	3.60	8.49
	UF-Rem-CAS	3.43e-2	**8.96e-3**	8.62e-3	0.117	0.227	2.15	3.51	**8.20**
	UF-Rem-Lock	4.45e-2	1.13e-2	1.01e-2	0.138	0.344	2.63	4.33	9.91
	UF-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
	SV	5.72e-2	9.72e-3	8.78e-2	0.124	0.237	2.70	5.03	12.5
	Label-Prop	12.6	1.02e-2	9.63e-3	0.121	0.375	2.44	4.75	9.68
BFS Sampling	UF-Early	2.69	1.07e-2	9.26e-3	9.42e-2	0.186	2.27	4.02	9.33
	UF-Hooks	2.65	1.09e-2	9.71e-3	9.53e-2	0.186	2.29	2.94	9.40
	UF-Async	2.69	1.08e-2	9.12e-3	9.31e-2	0.189	2.23	2.87	9.23
	UF-Rem-CAS	2.66	1.06e-2	9.19e-3	**9.24e-2**	**0.183**	2.21	**2.83**	9.11
	UF-Rem-Lock	2.67	1.13e-2	1.07e-2	0.113	0.219	2.69	3.68	10.8
	UF-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
	SV	2.54	1.12e-2	9.72e-3	9.87e-2	0.196	2.59	4.13	12.2
	Label-Prop	2.58	1.19e-2	1.03e-2	9.47e-2	0.446	2.31	3.21	9.91
LDD Sampling	UF-Early	0.117	1.32e-2	8.63e-3	0.124	0.193	1.74	4.63	8.52
	UF-Hooks	0.112	1.33e-2	8.81e-3	0.127	0.197	1.75	3.58	8.46
	UF-Async	0.103	1.32e-2	8.49e-3	0.123	0.193	1.71	3.48	8.31
	UF-Rem-CAS	9.86e-2	1.29e-2	8.48e-3	0.122	0.193	**1.69**	3.46	8.28
	UF-Rem-Lock	0.126	1.54e-2	1.03e-2	0.144	0.226	2.16	4.31	9.97
	UF-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
	SV	0.250	1.36e-2	8.81e-3	0.131	0.197	2.07	4.70	11.2
	Label-Prop	14.3	1.41e-2	8.99e-3	0.127	2.03	1.76	3.79	9.06
Other Systems	BFSCC [89]	2.60	1.94e-2	1.05e-2	0.169	1.34	5.56	61.6	62.5
	WorkeffCC [90]	0.41	0.247	2.78e-2	0.587	2.18	5.97	11.4	25.8
	MultiStep [93]	29.6	0.272	0.138	—	1.76	—	—	—
	Galois [78]	6.10e-2	2.55e-2	3.40e-2	1.167	1.77	—	—	—
	PatwaryRM [81]	6.81e-2	3.65e-2	3.93e-2	0.428	1.15	—	—	—
	GAP-SV [14]	0.103	0.134	0.150	5.669	7.01	—	—	—
	GAP-AF [97]	4.29e-2	5.30e-2	7.32e-2	0.172	0.306	—	—	—

Table 3: Running times (seconds) of ConnectIt algorithms and state-of-the-art static connectivity algorithms 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 —.

Takeaways. Without sampling, the UF-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*

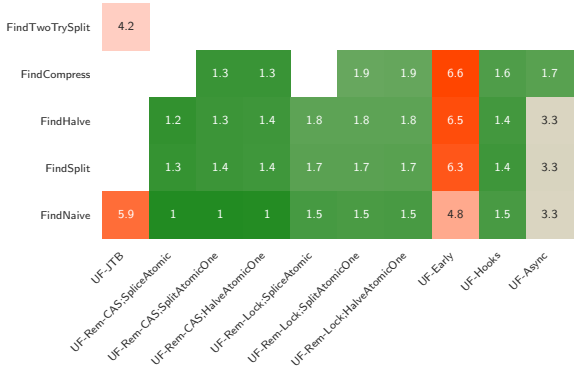


Figure 3: Relative performance of different union-find implementations on graphs used in our evaluation of ConnectIt in the *No Sampling* setting. The numbers are slowdowns relative to the fastest implementation.

Sampling setting, averaged across all graphs. UF-Rem-CAS implementations achieve consistently high performance in this setting using either SplitAtomicOne, HalveAtomicOne, or SpliceAtomic to perform compression. The UF-Rem-Lock implementation is 1.5–1.9x slower than UF-Rem-CAS for all 6 variants of this algorithm. UF-Early, UF-Hooks, and UF-Async are all slower on average: between 4.8–6.6x, 1.4–1.6x, and 1.7–3.3x on average, respectively. Finally, UF-JTB is much slower, although the FindTwoTrySplit option only incurs a more modest slowdown of 4.2x on average.

Performance Analysis. We annotated our union-find algorithms to measure the ***Max Path Length (MPL)***, or the longest path length experienced by the union-find algorithm during the execution of any Union operation, and the ***Total Path Length (TPL)***, which is 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 report the detailed results of the analysis in the full paper [33]. We find that the TPL is 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, 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 the full paper [33], 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, then 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 the 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, the fastest ConnectIt algorithm for these graphs is obtained by combining one of these sampling schemes with the UF-Rem-CAS algorithm (each scheme is fastest on one graph).

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 WorkeffCC (defined below), 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 memory. We list the main takeaways of the experimental results here, and present detailed results of the comparison in the full paper [33].

- **BFSCC**: The BFS-based connectivity implementation available from Ligra [89]. This algorithm computes each connected component by running a parallel BFS from the vertex with the lowest ID within it. We find that using sampling, ConnectIt is between 1.22–80.0x faster than BFSCC, and 15.6x faster on average.
- **WorkeffCC**: The work-efficient connectivity implementation by Shun et al. [90], which recursively computes LDD (publicly available as part of GBBS [31]). Our implementations without sampling are between 1.45–19.4x faster (5.6x faster on average). Our implementations with sampling are between 3.1–27.5x faster (9.03x faster on average). This algorithm previously held the record time for connectivity on the Hyperlink2012 graph on any system, running in 25.8 seconds on a 72-core machine. The fastest ConnectIt algorithm, UF-Rem-CAS with SplitAtomicOne, is 3.14x faster on the same machine, and thus breaks this record.
- **Multistep**: The hybrid BFS/label propagation method by Slota et al. [93]. 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 [77]. We found that their label propagation algorithm is consistently the fastest implementation in their code base, and report running times for this implementation for all but one graph (road_usa, discussed below). Our codes without sampling are between 1.78–3.69x faster than theirs (2.32x faster on average), and our codes using sampling are between 2.17–12.3x faster than theirs (6.21x faster on average). Our codes are 2.43x faster than their default (EdgetiledAsync) algorithm on the road_usa graph, which was their fastest algorithm for this graph.
- **PatwaryRM**: We compare with the multicore Rem's algorithm by Patwary et al. [81]. Our fastest implementations without sampling achieve between 1.27–2.87x speedup over their implementation (1.99x speedup on average), and our fastest implementations

Algorithm	RO	LJ	CO	TW	FR	RM	BA	CW	HL14	HL12
UF-Early	1.48e9	9.23e8	1.38e9	4.31e8	1.05e9	3.49e8	5.16e8	4.00e8	3.15e9	2.80e9
UF-Hooks	3.12e9	4.21e9	5.94e9	2.79e9	1.49e9	7.27e8	1.18e9	4.69e9	5.17e9	4.48e9
UF-Async	3.49e9	3.36e9	5.29e9	2.73e9	1.41e9	8.05e8	1.13e9	4.86e9	5.92e9	4.69e9
UF-Rem-CAS	**3.98e9**	**5.28e9**	**7.16e9**	**3.85e9**	**2.01e9**	**8.78e8**	**1.46e9**	**5.73e9**	**6.64e9**	**5.64e9**
UF-Rem-Lock	1.56e9	3.68e9	5.95e9	3.36e9	1.74e9	7.67e8	1.42e9	3.56e9	2.99e9	3.21e9
UF-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
SV	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 (directed edge insertions per second) achieved by ConnectIt streaming algorithms on our graph inputs, and two synthetic graph inputs from the RMAT (RM) 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 10% of the edges to use in the batch. Otherwise, the entire graph is used as part of a single batch, which is not permuted. Note that there are no queries in the batch. For each graph we display the algorithm with the highest throughput in bold.

with sampling achieve between 2.09–6.28x speedup over their implementation (4.31x speedup on average).

- **GAPBS**: We compared with the GAP Benchmark Suite, a state-of-the-art shared-memory graph processing benchmark [14], which implements the Afforest algorithm [97]. Our fastest implementations without sampling are between 0.33–4.17x faster than their Afforest [97] implementation (2.08x faster on average). Our fastest implementations with sampling are between 1.32–8.55x faster than their Afforest implementation (3.9x faster on average).

4.4 Streaming Parallel Graph Connectivity

Experiment Design. We run two types of streaming experiments. The first type generates a stream of edge insertions by sampling a fraction of the edges (f_u) from a static input graph to use as insertions. Unless otherwise mentioned, we use all edges as insertions ($f_u = 1$). The second type uses synthetic graph generators to sample edge insertions. We consider the RMAT and Barabasi-Albert (BA) graph generators [9, 11] in these experiments. We generate RMAT graphs using the parameters $(a, b, c) = (0.5, 0.1, 0.1)$. In both the RMAT and BA graphs, the number of vertices is $n = 2^{30}$ and the number of edges is $10n$. For both graphs, 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 COO, and sample 10% of the edges to use as insertions.

4.4.1 Streaming Throughput. We first consider the throughput achieved by each algorithm family in the setting where only insertions are applied. Table 4 reports the streaming throughput achieved by the fastest variant of each algorithm on each graph. Note that no sampling is applied in these streaming experiments. The insertions are streamed as part of a single, large batch, which is applied by the algorithm in parallel for Type (**1**) and (**2**), and separately applied for insertions first, then queries second, for Type (**3**) algorithms (see the discussion on these types in Section 3.5).

The performance of most of our union-find algorithms is consistently high on these graphs, in particular the performance of UF-Hooks, UF-Async, UF-Rem-CAS, and UF-Rem-Lock. As in the static setting, the UF-Rem-CAS algorithm consistently performs the best across all input graphs, achieving a maximum throughput of over 7 billion edge insertions per second on the com-Orkut graph. The other two union-find algorithms, UF-Early and UF-JTB, achieve somewhat inconsistent performance, which is consistent with our findings from Section 4.1. We discuss the performance of the remaining algorithms, which are even slower than UF-Early and UF-JTB, in the full version of our paper [33].

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 (seconds) and edge insertion rates (directed edges/second) for STINGER's dynamic connected components algorithm and ConnectIt's UF-Rem-CAS **algorithm (with** SplitAtomicOne**) when performing batch edge insertions on an empty graph with varying batch sizes. Inserted edges are sampled from the RMAT graph generator.**

4.4.2 Streaming Comparison with STINGER. STINGER is based on a dynamic data structure for representing dynamic graphs in the streaming setting [37]. STINGER supports a dynamic connected components algorithm by McColl et al. [72]. Their algorithm also supports edge deletions, which makes it more general than our algorithms. As far as we know, the only existing parallel algorithm designed for incremental connectivity is by Simsiri et al. [92], but unfortunately we were unable to obtain the code from the authors.

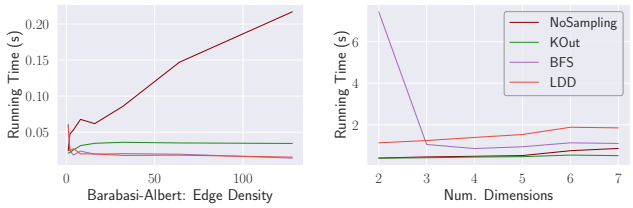
Comparison. The STINGER code takes a parameter which trades off space usage with the amount of re-computation that has to be done upon an update (edge insertion or deletion). We set it to the lowest possible value, which gave the best performance. We were unable to initialize the STINGER dynamic connectivity 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 this, we opted to generate batches from an RMAT graph generator using 2^{20} vertices. Our experiment inserts batches of varying sizes and measures the time required by the dynamic connectivity algorithm to process the batch. The times we report ignore the time taken by STINGER to update adjacency information.

Table 5 reports the results of the experiments for STINGER and for ConnectIt's UF-Rem-CAS algorithm with the SplitAtomicOne option. Both implementations are run on the same machine using all cores with hyper-threading. 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 batch-incremental connectivity implementations currently publicly available, it is designed for both edge insertions and deletions, and must perform extra work in anticipation of edge deletions, which our algorithms do not handle.

4.5 Algorithm Selection in ConnectIt

Given a graph, which combination of ConnectIt's sample and finish methods should one apply to obtain the best performance?

Evaluation on Synthetic Networks. To provide guidance, we evaluated UF-Rem-CAS with SplitAtomicOne and FindNaive, which are consistently ConnectIt's fastest connectivity methods, on two



(a) (b)

Figure 4: Running times (seconds) of the UF-Rem-CAS algorithm with SplitAtomicOne and FindNaive using different sampling methods on two synthetic graph families. Subfigure 4a displays results for graphs drawn from the Barabasi-Albert generator where the x-axis specifies the edge density, or the number of edges drawn for each newly added vertex. Subfigure 4b shows results for d-dimensional torii where the x-axis shows the dimension.

synthetic graph families with significantly different properties. Figure 4a displays results for graphs drawn from the Barabasi-Albert (BA) generator on 10^7 vertices, varying the edge density of the graphs in powers of two from 1 to 128. Figure 4b displays results for d-dimensional torii on 10^9 vertices where we vary the dimension of the grid (each vertex is connected to its $2d$ adjacent neighbors).

The diameter of the BA graphs decreases with increasing density, and as a result both BFS and LDD Sampling achieve good performance as the density increases. Both schemes use the direction-optimization technique used when traversing frontiers [13, 88], which is beneficial in high-density graphs. For the sparse (high-diameter) case where $n = m$, we found that k-out Sampling and not using sampling were the fastest.

For d-dimensional torii, k-out Sampling consistently performs the best across the range of d that we evaluate. For small d, there is little difference between k-out Sampling and no sampling, but for larger d, and thus higher average degree, k-out Sampling begins to achieve significant speedups over no sampling. The performance of BFS Sampling is poor on this graph family since the diameter of the d-dimensional torus is $O(n^{1/d})$, causing BFS to perform many rounds. The performance of LDD Sampling is also poor, since the induced clustering consists of many small clusters and thus most of the vertices must be processed in the finish phase.

Picking an Algorithm. Based on both our evaluation on the synthetic graph families above, and the results for a broad collection of large real-world graphs shown in Table 3, we devised a decision tree that can help users select an appropriate algorithm, which we show in Figure 5. We recommend using the UF-Rem-CAS algorithm, with any of the three possible splice strategies in combination with the FindNaive strategy. Based on our evaluation of union-find algorithms in Section 4.1.1, this family of algorithms is consistently the fastest. Regarding sampling, for extremely sparse networks with low average degree such as road_usa ($m/n < 3$), not using sampling can be beneficial. The reason is that 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 Sampling. On the other hand, if the graph is reasonably dense, and also has low diameter, either BFS or LDD Sampling can obtain the best results. Finally, if the graph diameter is unknown, or known to be high, then we recommend using k-out Sampling.

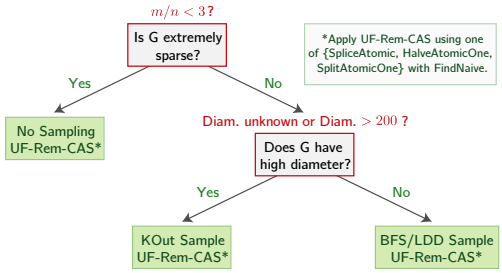


Figure 5: Decision tree for selecting a ConnectIt algorithm based on the input graph properties. The text shown in red are suggested (heuristic) decision rules derived from our evaluation.

5 APPLYING CONNECTIT

In this section, we examine how ConnectIt can be used to accelerate graph processing in two important graph applications.

5.1 Approximate Minimum Spanning Forest

Computing a minimum spanning forest (MSF) of a weighted undirected graph is an important problem, and some popular graph clustering algorithms including single linkage, and affinity clustering can be viewed as post-processing of a graph's minimum spanning forest [12, 71]. We consider the closely related problem of computing an *approximate* MSF, and show how a folklore approach can be accelerated using ConnectIt.

Definition. Consider a weighted graph $G(V, E, w)$. Let $\mathcal{F}_{\text{OPT}}$ represent any spanning forest of G of minimum weight. The approximate minimum spanning forest (AMSF) problem is to compute a spanning forest $\mathcal{F}_{\text{APX}}$ of G, where $W(\mathcal{F}_{\text{OPT}}) \leq W(\mathcal{F}_{\text{APX}}) \leq (1+\epsilon)W(\mathcal{F}_{\text{OPT}})$. We make the standard assumption that the weights are polynomially-bounded, i.e., $\exists c$ s.t. $\forall e \in E, w(e) = O(n^c)$ for some constant c.

Algorithm. The following is a folklore algorithm for the AMSF problem. Let $W_{\min} = \min_{e \in E} w(e)$. Bucket the edges of G, such that the i'th bucket contains edges with weight in the range $[W_{\min}(1+\epsilon)^i, W_{\min}(1+\epsilon)^{i+1})$. As the weights in G are polynomially bounded, there are $O(\log_{1+\epsilon} n)$ buckets. The algorithm maintains a connectivity labeling C, which is updated as it processes the buckets from smallest to largest weight. For the i'th bucket, it first removes all edges that are self-loops. It then computes a spanning forest $\mathcal{F}_i$ on the remaining edges in the bucket, and updates the connectivity labeling so that C represents the connected components of $\cup_{j=0}^{i} \mathcal{F}_j$. The final minimum spanning forest is simply the union of all of the computed spanning forests.

Variants. We consider several variants of the algorithm above:

(1) AMSF-COO is a direct implementation of the algorithm described above, which works by writing the edges in the graph into the COO format, which is then sorted by weight. The buckets are represented using $O(\log n)$ pointers into this array.

(2) AMSF-F avoids extracting *all* of the edges into COO, and instead extracts the *buckets* in COO format one at a time from input graph (in CSR representation). The extracted edges are removed (filtered out) from the CSR representation, and at the end of the algorithm, the CSR graph has no remaining edges.

(3) AMSF-NF works similarly to AMSF-F, with the exception that the graph is not mutated when extracting edges (and thus all edges in the graph are inspected in every round).

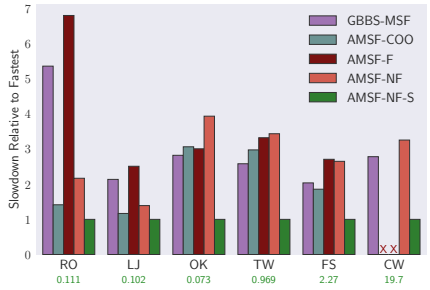


Figure 6: Relative performance of the AMSF algorithms with $\epsilon = 0.25$. The values on the y-axis are normalized to the running time of the fastest algorithm; this time in seconds is shown below the name of each graph in green. Methods resulting in failure are marked with a red x.

For the AMSF-NF variant, we consider applying a *sampling* optimization similar to the one used in ConnectIt, which we refer to as AMSF-NF-S. The idea is to compute, in each round, the largest connected component in the connectivity labeling, $L_{\max}$, and to skip processing vertices in the $L_{\max}$ component and their incident edges when searching for the spanning forest over edges in the current round. This optimization is correct, since any edge emanating out of the $L_{\max}$ component that is skipped will be considered from the other endpoint, which will not be skipped.

Experimental Results. We evaluated all of the AMSF variants above over weighted versions of our unweighted graph inputs by adding random weights drawn from an exponential distribution with a constant mean. We set $\epsilon = 0.25$. We did not evaluate weighted versions of the Hyperlink graphs due to storage constraints in our experimental environment. All AMSF variants use the UF-Rem-CAS method with SplitAtomicOne and FindNaive to concurrently update the connectivity labeling when processing edges within a bucket. We compare these variants to GBBS-MSF, an *exact* minimum spanning forest (MSF) algorithm from GBBS [31], which is a CSR-based implementation of Borůvka's algorithm. To the best of our knowledge, GBBS-MSF is the fastest existing multicore MSF algorithm.

Figure 6 shows our results. Other than the two smallest graphs, no AMSF variant without sampling outperforms the GBBS-MSF algorithm by a significant margin. For AMSF-COO, the primary reason for its poor performance on large graphs is the large overhead of sorting and processing edges stored in COO. AMSF-F suffers for similar reasons, since a large fraction of the edges fall into the first few buckets, which are explicitly stored in COO. Both AMSF-COO and AMSF-F fail due to memory allocation errors on the ClueWeb graph for this reason. AMSF-NF-S consistently attains the best performance across all graphs, obtaining between 2.03–5.36x speedup over the exact algorithm (2.95x on average), since it quickly finds a partial spanning forest spanning most of the largest connected component after processing the first few buckets, and can skip processing vertices in this component in subsequent rounds.

5.2 Index-Based SCAN

The Structural Clustering Algorithm for Networks (SCAN) clustering algorithm [103] clusters graphs using the idea that 'similar' vertices have similar neighbor sets, e.g., using a similarity measure such as cosine similarity. SCAN is defined using two parameters: a similarity threshold $\epsilon \in [0, 1]$ and $\mu \geq 2$. Two vertices are said to be *ϵ-similar* if their similarity is at least ϵ. A vertex is a *core* vertex if it has at least μ neighbors that it is ϵ-similar to. The objective is to find a maximal clustering where all vertices within a cluster are connected over a path of ϵ-similar edges.

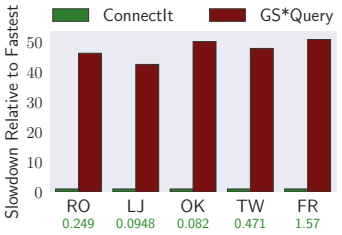


Figure 7: Relative performance of our parallel implementation of GS*-Query using ConnectIt, and GS*-Query when searching for a clustering with $\epsilon = 0.1$ and $\mu = 3$. The fastest running time in seconds for each graph is shown in green.

GS*-Index and GS*-Query [102]. Motivated by the fact that often one is interested in finding multiple clusterings with varying ϵ and μ, the GS*-Index [102] algorithm builds an index structure so that cluster retrieval can be quickly performed using the index. Once the index has been computed, for a given ϵ and μ the query algorithm, GS*-Query (Algorithm 4 in [102]), performs a sequential search from the core vertices, considering only ϵ-similar edges.

Experimental Results. We parallelized the GS*-Query algorithm with ConnectIt using the UF-Rem-CAS algorithm with FindNaive and SplitAtomicOne, and compared the parallel query algorithm to the sequential GS*-Query algorithm. Figure 7 shows the results of our evaluation for $\epsilon = 0.1$ and $\mu = 3$ on a subset of our graph inputs. The index requires $O(m)$ space (with a significant constant), and so we were unable to evaluate it on our larger graph inputs. Our results show that by using ConnectIt, we can obtain between 42.5–50.5x speedup (47.4x on average) over the original sequential GS*-Query, potentially enabling an order of magnitude more clusterings to be evaluated by users. We have recently also parallelized the index construction algorithm (GS*-Index) [99].

6 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 connectivity implementations to date.

ACKNOWLEDGEMENT

Thanks to Edward Fan for initial implementations of Rem's algorithm. Thanks to Guy Blelloch, Siddhartha Jayanti, Jakub Lacki, and Yuanhao Wei for for helpful discussions and suggestions. The name of our framework is inspired by Saman Amarasinghe and the Commit group. 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. In *International Conference on Principles of Distributed Systems, (OPODIS)*. 15:1–15:16.

[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 *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 *IEEE Symposium on Foundations of Computer Science (FOCS)*. 674–685.

[5] Baruch Awerbuch and Yossi 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). 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] David A. Bader and Kamesh Madduri. 2006. GTgraph: A Synthetic Graph Generator Suite. *Atlanta, GA, February* 38 (2006).

[10] 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.

[11] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512.

[12] MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Raimondas Kiveris, Silvio Lattanzi, and Vahab S. Mirrokni. 2017. Affinity Clustering: Hierarchical Clustering at Scale. In *Advances in Neural Information Processing Systems (NeurIPS)*. 6867–6877.

[13] 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)*. 12:1–12:10.

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

[15] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab Mirrokni. 2019. Near-Optimal Massively Parallel Graph Connectivity. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 1615–1636.

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

[17] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, Vahab Mirrokni, and Warren Schudy. 2020. Parallel Graph Algorithms in Constant Adaptive Rounds: Theory Meets Practice. *PVLDB* 13, 13 (2020), 3588–3602.

[18] 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.

[19] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[20] 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.

[21] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *International World Wide Web Conference (WWW)*. 595–601.

[22] 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.

[23] E.Norberto Caceres, Frank Dehne, Henrique Mongelli, Siang W. Song, and Jayme L. Szwarcfiter. 2004. A Coarse-Grained Parallel Algorithm for Spanning Tree and Connected Components. In *European Conference on Parallel Processing (Euro-Par)*. 828–831.

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

[25] Ka W. Chong and Tak W. Lam. 1995. Finding Connected Components in $O(\log n \log\log n)$ Time on the EREW PRAM. *J. Algorithms* 18, 3 (1995), 378–402.

[26] 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.

[27] Guojing Cong and Paul Muzio. 2014. Fast Parallel Connected Components Algorithms on GPUs. In *European Conference on Parallel Processing (Euro-Par)*. 153–164.

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

[29] 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.

[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 *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1300–1319.

[33] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: A Framework for Static and Incremental Parallel Graph Connectivity Algorithms. (2020). https://arxiv.org/abs/2008.03909

[34] 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.

[35] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA)*. 11:1–11:8.

[36] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall.

[37] David Ediger, Rob 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.

[38] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. 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.

[39] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *PVLDB* 9, 12 (2016), 1233–1244.

[40] 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 (2018), 555–592.

[41] 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.

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

[43] 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.

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

[45] Shay Halperin and Uri Zwick. 1996. An Optimal Randomized Logarithmic Time Connectivity Algorithm for the EREW PRAM. *J. Comput. Syst. Sci.* 53, 3 (1996), 395–416.

[46] Susanne E Hambrusch and Lynn 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 (1990), 626–642.

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

[49] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[50] 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.

[51] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. 1979. Computing Connected Components on Parallel Computers. *Commun. ACM* 22, 8 (1979), 461–464.

[52] Jacob Holm, Valerie King, Mikkel Thorup, Or Zamir, and Uri Zwick. 2019. Random k-Out Subgraph Leaves Only $O(n/k)$ Inter-component Edges. In *2019*

IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS). IEEE, 896–909.

[53] Changwan Hong, Laxman Dhulipala, and Julian Shun. 2020. Exploring the Design Space of Static and Incremental Graph Connectivity Algorithms on GPUs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 55–69.

[54] 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.

[55] Jeremy Iverson, Chandrika Kamath, and George Karypis. 2015. Evaluation of Connected-Component Labeling Algorithms for Distributed-Memory Systems. *Parallel Comput.* 44 (2015), 53–68.

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

[57] 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.

[58] Chirag Jain, Patrick Flick, Tony Pan, Oded Green, and Srinivas Aluru. 2017. An Adaptive Parallel Algorithm for Computing Connected Components. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2428–2439.

[59] 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.

[60] 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.

[61] 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.

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

[63] 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)*. 18:1–18:13.

[64] Vaclav Koubek and Jana Krsnakova. 1985. Parallel Algorithms for Connected Components in a Graph. In *Fundamentals of Computation Theory*. 208–217.

[65] 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.

[66] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. 1990. Efficient Parallel Algorithms for Graph Problems. *Algorithmica* 5, 1-4 (1990), 43–64.

[67] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *International World Wide Web Conference (WWW)*. 591–600.

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

[69] 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.

[70] 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.

[71] Christopher D. Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. *Introduction to Information Retrieval*. Cambridge University Press.

[72] Robert McColl, Oded Green, and David A. Bader. 2013. A New Parallel Algorithm for Connected Components in Dynamic Graphs. In *IEEE International Conference on High Performance Computing (HiPC)*. 246–255.

[73] 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.

[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 C. Xu. 2013. Parallel Graph Decomposition using Random Shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 196–203.

[76] Dhruva Nath and Shachindra 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. 2014. Deterministic Galois: On-Demand, Portable and Parameterless. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 499–512.

[79] 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.

[80] M. M. Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok 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.

[81] M. M. 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.

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

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

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

[85] 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.

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

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

[88] 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.

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

[90] 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.

[91] 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.

[92] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2017. Work-Efficient Parallel Union-Find. *Concurrency and Computation: Practice and Experience* 30, 4 (2017).

[93] 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.

[94] George M. Slota, Sivasankaran Rajamanickam, and Kamesh 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.

[95] Jyothish Soman, Kothapalli Kishore, and P.J. Narayanan. 2010. A Fast GPU Algorithm for Graph Connectivity. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1–8.

[96] 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.

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

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

[99] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2021. Parallel Index-Based Structural Graph Clustering and Its Approximation. *To appear in ACM International Conference on Management of Data (SIGMOD)* (2021).

[100] Uzi Vishkin. 1984. An Optimal Parallel Connectivity Algorithm. *Discrete Applied Mathematics* 9, 2 (1984), 197–207.

[101] 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 (2017), 3:1–3:49.

[102] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2017. Efficient Structural Graph Clustering: An Index-based Approach. *PVLDB* 11, 3 (2017), 243–255.

[103] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. 2007. SCAN: A Structural Clustering Algorithm for Networks. In *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*. 824–833.

[104] 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 (2014), 1821–1832.

[105] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. 2020. FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing (PP)*. SIAM, 46–57.

[106] Da Zheng, Disa Mhembere, 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.