A New Parallel Algorithm for Connected Components in Dynamic Graphs

Robert McColl, Oded Green, David A. Bader
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA

***Abstract*—Social networks, communication networks, business intelligence databases, and large scientific data sources now contain hundreds of millions elements with billions of relationships. The relationships in these massive datasets are changing at ever-faster rates. Through representing these datasets as dynamic and semantic graphs of vertices and edges, it is possible to characterize the structure of the relationships and to quickly respond to queries about how the elements in the set are connected. Statically computing analytics on snapshots of these dynamic graphs is frequently not fast enough to provide current and accurate information as the graph changes. This has led to the development of dynamic graph algorithms that can maintain analytic information without resorting to full static recomputation.**

In this work we present a novel parallel algorithm for tracking the connected components of a dynamic graph. Our approach has a low memory requirement of $O(V)$ and is appropriate for all graph densities. On a graph with 512 million edges, we show that our new dynamic algorithm is up to $128X$ faster than well-known static algorithms and that our algorithm achieves a $14X$ parallel speedup on a x86 64-core shared-memory system. To the best of the authors' knowledge, this is the first parallel implementation of dynamic connected components that does not eventually require static recomputation.

I. Introduction

In graph theory, given an undirected graph $G = (V, E)$, a connected component $C \subseteq V$ ensures that for each $s, t \in C$ there is a path between s and t. Finding the connected components of a graph is a well-studied problem. The component labeling of a graph can be used as building block within other calculations: betweenness centrality, community detection, image processing, and others [8]. Hopcroft and Tarjan [20] presented one of the first approaches for partitioning a graph into connected components using a series of Depth First Searches (DFS), one for each component. Breadth First Search (BFS) can also be used in place of DFS. Approaches relying only on DFS and BFS are aimed at static graphs which can be thought of as snapshots of a dynamic graph at an exact time.

In examining social networks such as Facebook, where vertices and edges may represent people and friendships or messages, the high rate of change makes computing many analytics on snapshots of these massive graphs impractical as the time between updates is much less than the time needed to compute these analytics. This has led to the development of algorithms for dynamic graphs in which edges can be inserted or deleted. With respect to connected components, edge insertions may join two different components, and edge deletions may split one component into two. Given the graph G and the components labels C, determining if an insertion has joined two components can be done in $O(1)$ time; however, determining if a deletion has broken a component is more expensive. Both of these scenarios must be detected and handled by any dynamic graph algorithm.

In order to keep up with rapid changes, the algorithm and its implementation must attain performance through full system utilization and workload balancing. A strategy used in this and other works is to aggregate updates into a batch over time or until a certain number are collected. This batch can then be applied in parallel. Batches increase the available amount of parallel work and provide opportunities to reduce redundant calculations between updates; however, they also present synchronization challenges and the potential for workload imbalance.

In this work, we show how to maintain an exact labeling of the connected components of a dynamic graph with millions of edges while applying batches of edge insertions and deletions in parallel. To accomplish this task, we employ a "parent-neighbor" sub-graph structure of up to a fixed size $O(V)$. In this sub-graph, parent and neighbor relationships represent paths to the root of a breadth-first traversal of each component. As long as each vertex has a path to the root, the component is unbroken. In practice, we show that the average case for maintaining this approach is much faster than performing the $O(V + E)$ work required to recompute from scratch. The storage complexity is $O(V)$.

The remainder of the paper will be structured as follows: this section presents related work on serial and parallel connected components algorithms for both static and dynamic graphs. In Section II, we present our new algorithm and the required data structures. In Section III, we will discuss experimental methodology. Section IV gives quantitative and performance results. Finally, in Section V, we will give conclusions and future work.

A. Related Work

Many authors have published a variety of parallel algorithms that compute solutions to the connected components problem on shared-memory computers. Hirschburg *et al.*

[19] presented the CONNECT algorithm, a classic parallel algorithm to find the connected components in an undirected graph. Two variations are presented, the first requires $|V|^2$ processors and the second requires $V\lceil V\log V\rceil$ processors. Both have a time complexity of $O(\log^2 V)$.

Shiloach and Vishkin [25] gave an $O(\log V)$ algorithm that used $|V| + 2|E|$ processors. Because of its simplicity, parallelism, and load balancing it has been implemented on several multi-core and many-core systems. In the average case, it completes in $\sim d/2$ iterations, where d is the diameter of the graph.

Shiloach and Even present an algorithm that tracks connected components in dynamic graphs as edges are removed [24]. They accomplish this by maintaining a structure representing the vertices in the levels of breadth first search tree for each component. For each deleted edge, if the vertices of the edge are on the same level, the data structure does not require updating. However, if they are on different levels and the lower vertex has no other neighbors above it, the lower vertex (and possibly a subtree) could potentially drop a level or fall out of the tree altogether. Both of which require updating the data structure. In more recent theoretical work [23], a sequence of graphs are maintained, one per each edge inserted, and reachability trees. The amortized update time is $O(E + V\log V)$ and a worst-case query time of $O(V)$.

In [12] a technique is shown that allows treating dense graphs as sparse graphs - this is known as sparsification. Sparsification is achieved by dividing the original graph into smaller subgraphs with V vertices and $O(V)$ edges. Certificates (aka graph properties) are computed for each subgraph. This is followed by merging of these certificates. Ferragina [15] uses the sparsification technique with the algorithm of [21] to give an additional algorithm for computing static and parallel connected components for a PRAM-like system.

Henzinger *et al.* [17] use a series of graphs in which the vertices are colored based on their degrees to detect new components in the face of deletions. Deleted edges are removed from all of the graphs that are represented and the colors are updated, then any $O(V + E)$ connected components algorithm is run on all graphs to discover components containing only vertices of a certain color which indicates the creation of a new component.

Henzinger and King [18] created another algorithm that maintains several spanning trees in each component for different levels of sparsity and performs updates on the trees only when deletions occur in both the graph and the tree.

The problem with many of these streaming algorithms is that they are too expensive to compute in practice, require too much storage - even up to the size of the graph $O(V + E)$, ignore real world graph properties, or do not consider practical multi-core and many-core systems.

B. Real World Graph Properties

In Albert *et al.* [1] the authors present the small-world phenomena, which states that in many networks the distance between two vertices is relatively small. It has been shown that social networks have low diameters (maximal length of the shortest path connecting any two vertices) [1]. Leskovec *et al.* [22] define the effective diameter as the 90th percentile distance of all the vertices. Using this definition limits the effect of any tail-like structures in the graph.

Barabasi *et al.* [3] show that the distribution of the edges over the vertices in the graph follows a power law. They also discuss the concept of preferential attachment in social networks, which states that highly-connected vertices are more likely to be incident to insertions.

In Broder *et al.* [6] the authors show that World Wide Web (WWW) has one connected component that contains 90% of the vertices in the graph. The work of Leskovec *et al.* [22] confirmed that many real world networks have these properties.

These properties help to motivate our work. Considering them together, it is intuitive that a single edge deletion to a high degree vertex is not likely to break apart a component. Based on this, we posit that it should be possible to track only a small portion of the edges around high degree vertices or even a fixed number per vertex in order to maintain a correct component labeling in these graphs.

C. STINGER - A Dynamic Graph Data Structure

STINGER [2] is an high-performance data structure designed for dynamic graph problems. STINGER is a compromise between the massive storage and fast updates of adjacency matrices and the minimal storage and static nature of Compressed Sparse Row (CSR) representation. STINGER has faster insertions and better locality than traditional dynamic sparse structures like adjacency lists. Further, STINGER is designed for parallelism such that multiple threads can read and update the graph concurrently [10].

STINGER has been used to implement a variety of dynamic graph algorithms including clustering coefficients [9], community detection, and betweenness centrality [16]. The reader is referred to [2] for additional details. We use STINGER in this work to maintain the dynamic graph and to provide a software platform for our algorithm. STINGER is free and open source software co-developed by our group[1].

II. Tracking Connected Components in a Dynamic Graph

In the following section we present our new algorithm for maintaining connected components for dynamic graphs. We briefly discuss the concept of tracking connected components in a dynamic graph. We follow this with an introduction to our new algorithm and data structure. Finally,

[1]Available from http://www.stingergraph.com

Table I
THE DATA STRUCTURES MAINTAINED WHILE TRACKING DYNAMIC CONNECTED COMPONENTS.

Name	Description	Type	Size (Elements)
C	Component labels	array	$O(V)$
$Size$	Component sizes	array	$O(V)$
$Level$	Approximate distance from the root	array	$O(V)$
PN	Parents and neighbors of each vertex	array of arrays	$O(V \cdot thresh_{PN}) = O(V)$
$Count$	Counts of parents and neighbors	array	$O(V)$
$thresh_{PN}$	Maximum count of parents and neighbors for a given vertex	value	$O(1)$
$\tilde{E}_I$	Batch of edges to be inserted into graph	array	$O(batch\ size)$
$\tilde{E}_R$	Batch of edges to be deleted from graph	array	$O(batch\ size)$

the insertion and deletion approaches are discussed. For simplicity, the pseudo code in this section does not explicitly indicate atomic instructions.

Dynamic graph algorithms present several challenges which include: 1) Correctness: for exact algorithms, the results should be correct and consistent at fixed points in time for the graph at that same point. 2) Parallelism: to achieve the performance necessary to keep up with high-speed data streams, an algorithm must be able to use all of the resources available in the system. Additionally, synchronization and communications need to be minimized. 3) Time complexity: the complexity of the dynamic algorithm should be better than that of the static; however, as long as the real-world performance of the dynamic algorithm is better on average on the data of interest, it may be tolerable for the complexities to be equivalent. 4) Storage complexity: this too should generally be comparable to or better than the static case. Computing the connected components of a static graph requires $O(V + E)$ memory including the component labels $O(V)$ and the graph itself $O(V+E)$. A dynamic graph algorithm should not increase this bound. Given that the size of the graph can be on the same scale as the total memory in the machine, it is preferred that a dynamic graph algorithm limit the amount of extras storage required to $O(V)$.

Updating the connected components following an edge insertion only requires a comparison of the component labels of the vertices belonging to the edge. If the vertices have the same component label, then the insertion operation is complete. If the vertices have different component labels, then the two independent components need to be relabeled as the same component.

When maintaining a component labeling, edge deletion is considerably more challenging to handle than edge insertion. Deletions require determining if the deleted edge was the single path connecting two otherwise independent components. This is easy when the deleted edge is the only edge incident to one (or both) of the vertices; however, if both vertices have additional adjacencies, alternative path(s) between the vertices may exist. Obviously it is possible to use a SPSP (Single Pair Shortest Path) algorithm such as BFS to verify that an alternative path exists. Unfortunately, the worst case complexity for this is $O(V + E)$ which is the same as the complexity bound for computing connected components. For the cases that the edge deletion did split the connected component into two parts, it is necessary to search and relabel the new components.

Algorithm 1 A parallel breadth-first traversal that extracts the parent-neighbor subgraph.

```
Input: $G(V, E)$
Output: $C_{id}$, $Size$, $Level$, $PN$, $Count$
1: for $v \in V$ do
2:     $Level[v] \leftarrow \infty$, $Count[v] \leftarrow 0$
3: for $v \in V$ do
4:     if $Level[v] = \infty$ then
5:         $Q[0] \leftarrow v$, $Q_{start} \leftarrow 0$, $Q_{end} \leftarrow 1$
6:         $Level[v] \leftarrow 0$, $C_{id}[v] \leftarrow v$
7:         while $Q_{start} \neq Q_{end}$ do
8:             $Q_{stop} \leftarrow Q_{end}$
9:             for $i \leftarrow Q_{start}$ to $Q_{stop}$ in parallel do
10:                for each neighbor $d$ of $Q[i]$ do
11:                    if $Level[d] = \infty$ then
12:                        $Q[Q_{end}] \leftarrow d$
13:                        $Q_{end} \leftarrow Q_{end} + 1$
14:                        $Level[d] \leftarrow Level[Q[i]] + 1$
15:                        $C_{id}[d] \leftarrow C_{id}[Q[i]]$
16:                    if $Count[d] < thresh_{PN}$ then
17:                        if $Level[Q[i]] < Level[d]$ then
18:                            $PN_d[Count[d]] \leftarrow Q[i]$
19:                            $Count[d] \leftarrow Count[d] + 1$
20:                        else if $Level[Q[i]] = Level[d]$ then
21:                            $PN_d[Count[d]] \leftarrow -Q[i]$
22:                            $Count[d] \leftarrow Count[d] + 1$
23:            $Q_{start} \leftarrow Q_{stop}$
24:        $Size[v] \leftarrow Q_{end}$
```

It becomes apparent that it is desirable to find a mechanism that can state if the deletion is "safe", meaning that it is possible to state in $O(1)$ time whether or not the deletion could have broken a component. We require that this mechanism have a 100% true positive rate – all deletions marked as safe are truly safe, but allow for some false negatives as the search and relabel process will appropriately handle these cases.

The key challenge is minimizing the false negatives - the cases where the mechanism suggests that the deletion is unsafe when it actually is safe. This goes back to reducing the need to search for an additional path between the vertices to avoid doing the same work as static recomputation.

A. The Parents-Neighbors Sub-graph

In this subsection we present our algorithm and its respective data structure that has a low memory requirement of $O(V)$. Some approaches have higher memory requirements

Algorithm 2 The algorithm for updating the parent-neighbor subgraph for inserted edges.

```
Input: $G(V, E)$, $\tilde{E}_I$, $C_{id}$, $Size$, $Level$, $PN$, $Count$
Output: $C_{id}$, $Size$, $Level$, $PN$, $Count$
 1: for all$\langle s, d\rangle \in \tilde{E}_I$ in parallel do $E \leftarrow E \cup \langle s, d\rangle$
 2:     insert$(E, \langle s, d\rangle)$
 3:     if $C_{id}[s] = C_{id}[d]$ then
 4:         if $Level[s] > 0$ then
 5:             if $Level[d] < 0$ then
 6:                 // d is not "safe"
 7:                 if $Level[s] < -Level[d]$ then
 8:                     if $Count[d] < thresh_{PN}$ then
 9:                         $PN_d[Count[d]] \leftarrow s$
10:                         $Count[d] \leftarrow Count[d] + 1$
11:                     else
12:                         $PN_d[0] \leftarrow s$
13:                     $Level[d] \leftarrow -Level[d]$
14:             else
15:                 if $Count[d] < thresh_{PN}$ then
16:                     if $Level[s] < Level[d]$ then
17:                         $PN_d[Count[d]] \leftarrow s$
18:                         $Count[d] \leftarrow Count[d] + 1$
19:                     else if $Level[s] = Level[d]$ then
20:                         $PN_d[Count[d]] \leftarrow -s$
21:                         $Count[d] \leftarrow Count[d] + 1$
22:                 else if $Level[s] < Level[d]$ then
23:                     for $i \leftarrow 0$ to $thresh_{PN}$ do
24:                         if $PN_d[i] < 0$ then
25:                             $PNV_d[i] \leftarrow s$,
26:                             Break for-loop
27:         $\tilde{E}_I \leftarrow \tilde{E}_I \backslash \langle s, d\rangle$
28: for all$\langle s, d\rangle \in \tilde{E}_I$ do
29:     if $C_{id}[s] \neq C_{id}[d]$ then
30:         if $Size[s] = 1$ then
31:             $Size[s] \leftarrow 0$
32:             $Size[d] \leftarrow Size[d] + 1$
33:             $C_{id}[s] \leftarrow C_{id}[d], PN_s[0] \leftarrow d$
34:             $Level[s] \leftarrow$ abs$(Level[d]) + 1, Count[s] \leftarrow 1$
35:         else
36:             connectComponent(Input, $s$, $d$)
```

Algorithm 3 The algorithm for updating the parent-neighbor subgraph for deleted edges.

```
Input: $G(V, E)$, $\tilde{E}_R$, $C_{id}$, $Size$, $Level$, $PN$, $Count$
Output: $C_{id}$, $Size$, $Level$, $PN$, $Count$
 1: for all$\langle s, d\rangle \in \tilde{E}_R$ in parallel do
 2:     $E \leftarrow E \backslash \langle s, d\rangle$
 3:     $hasParents \leftarrow$ false
 4:     for $p \leftarrow 0$ to $Count[d]$ do
 5:         if $PN_d[p] = s$ or $PN_d[p] = -s$ then
 6:             $Count[d] \leftarrow Count[d] - 1$
 7:             $PN_d[p] \leftarrow PN_d[Count[d]]$
 8:         if $PN_d[p] > 0$ then
 9:             $hasParents \leftarrow$ true
10:     if (not $hasParents$) and $Level[d] > 0$ then
11:         $Level[d] \leftarrow -Level[d]$
12: for all $\langle s, d\rangle \in \tilde{E}_R$ in parallel do
13:     for all $p \in PN_d$ do
14:         if $p \geq 0$ or $Level[$abs(p)$] > 0$ then
15:             $\tilde{E}_R \leftarrow \tilde{E}_R \backslash \langle s, d\rangle$
16: $PREV \leftarrow C_{id}$
17: for all $\langle s, d\rangle \in \tilde{E}_R$ do
18:     $unsafe \leftarrow (C_{id}[s] = C_{id}[d] = PREV_s)$
19:     for all $p \in PN_d$ do
20:         if $p \geq 0$ or $Level[$abs(p)$] > 0$ then
21:             $unsafe \leftarrow$ false
22:     if $unsafe$ then
23:         if $\{\langle u, v\rangle \in G(E, V) : u = s\} = \emptyset$ then
24:             $Level[s] \leftarrow 0, C_{id}[s] \leftarrow s$
25:             $Size[s] \leftarrow 1, Count[s] \leftarrow 0$
26:         else
27:             Algorithm 4
28:             repairComponent(Input, $s$, $d$)
```

of $O(V + E)$. This limits the size of graph that can be analyzed. Further, a smaller memory footprint can allow for better usage of the cache and reduced dependence on memory bandwidth.

We call our data structure the parent-neighbor sub-graph. This sub-graph is extracted using breadth-first traversals of the original graph (one for each connected component). The result is a directed sub-graph of the original undirected graph. Each vertex will maintain a list of vertices that are in the level above ("parents") and/or in the same level ("neighbor") of the traversal where a level is a single frontier in the search and parents / neighbors of a vertex must be adjacent to that vertex in the original graph. Note that if all the parents and neighbors were maintained then the memory requirement of this would be $O(V+E)$. Instead we place a threshold ($thresh_{PN}$) on the total number of parent-neighbors for each vertex. Given that each vertex can have at most $thresh_{PN}$ parent-neighbors, the storage requirement of the parent-neighbor subgraph is $O(V \cdot thresh_{PN})$. Since $thresh_{PN}$ is a constant $O(1)$, the storage requirement can be reduced to $O(V)$. In Section IV we discuss the impact of selecting $thresh_{PN}$.

This sub-graph is similar to the parent lists maintained in Brandes's betweenness centrality algorithm [5]. In that algorithm, every vertex has a list of the vertices in the level above that are adjacent to it. Since each vertex stores a list of up to the full size of its adjacency list, the memory required for Brandes's algorithm is $O(V + E)$. The key differences between our parent-neighbor sub-graph and the parent lists of [5] are that we have placed a bound on the maximum number of adjacencies in the list and that our list also stores adjacent vertices that are on the same level.

B. Data Structure and Algorithm Details

Table I denotes the variables used by the algorithm and data structure. We give a brief justification for the memory requirements. As each vertex knows the component it belongs to and the size of its component, a total of $O(V)$ memory is required.

To store the partial list of parents and neighbors of a vertex we have created an array called PN. To distinguish between the parents and neighbors, the parents use positive numbers and the neighbors use negative numbers. The benefit of such an implementation is that there is a single array and single counter for each vertex. Additional benefits include spatial locality, reduced memory footprint vs. separate arrays, and ease of programming. For this reason, vertices will be indexed $1, 2, 3, .., |V|$.

In the next subsection we further elaborate on the $Level$ array, yet, we want to note ahead of time that this array does not maintain the actual distance from the root but only an approximate distance as will become apparent. In addition to this, we use "negative" distances to mark vertices that potentially have lost all their parents yet still have neighbors. These vertices should still have a path to the root through their neighbors. The negative simply indicates to other vertices that this vertex should not be depended on when searching for a connection to the root.

As PN is an undirected sub-graph of a directed graph, each inserted or deleted edge, (u, v), is taken care of twice – once from v's perspective and once u's perspective.

C. Data Structure Initialization

Each vertex is initially unlabeled, its $Level$ is set to ∞, and its $Counter$ is set to zero. Component sizes are set to zero. In addition to the structures listed in Table I, a temporary workspace of two $|V|$ queues is used during the initialization.

We use a series of parallel BFS traversals, one for each connected component, to find and label the members of each component and initialize the component sizes, parents, and neighbors. Our BFS can be found in Algorithm 1. The first unlabeled vertex is selected and enqueued. While the queue is not empty, the edges of the vertices in the current frontier are explored concurrently. Newly discovered vertices are marked, enqueued to the next frontier, and a new parent is inserted. For previously discovered vertices, two scenarios of interest can arise: a new parent has been found or a new neighbor has been found. These will be added to the PN array.

Insertion of the current vertex as a parent or neighbor only occurs if the total number of parents and neighbors for the adjacent vertex is less than $thresh_{PN}$. Since the BFS is level-synchronous, all parents of a vertex will be found before even a single neighbor is found. Neighbors will only be added to the parent-neighbor list if the vertex did not have at least $thresh_{PN}$ parents. Synchronization is handled through atomic **compare-and-swap** operations on the $Level$ array and atomic **fetch-and-add** to enqueue newly found vertices and to update the PN array and counter. For simplicity these have not been marked in the pseudo code.

As a slight optimization, all edges connecting singleton connected components are skipped in the first pass. Once the larger components have been labeled, a parallel pass over all vertices is used to initialize the singleton components.

D. Insertions and the Subgraph

For an edge insertion, the vertices on each edge are first checked to see if they belong to the same component. The pseudo-code for edge insertion can be found in Algorithm 2. We differentiate two key scenarios for the insertion of edge $\langle s, d\rangle$: 1) the edge is within a connected component (intra-connecting) and 2) the edge joins two components (inter-connecting).

For the first, the levels of s and d are checked to see if a new parent or neighbor relationship can be created. Assume that $Level_s \leq Level_d$. If $Counter_d < thresh_{PN}$ then s is added to PN_d as a parent if $(Level_s < Level_d)$ or as a neighbor otherwise, and $Counter_d$ is incremented. If $Counter_d = thresh_{PN}$ and $Level_s < Level_d$ (i.e. s can be a parent), d's parents and neighbors are searched for neighbors that could be replaced by the parent s.

The intra-connecting edges are handled in parallel. The inter-connecting edges are handed consecutively upon completion of the intra-connecting edges.

When components are connected, a parallel BFS starts at the joining vertex for the smaller of the two components to relabel the smaller component's members and add them to the larger component's tree in PN. For performance purposes, singleton components are set aside during this step. In the following step, all singletons handled in parallel. Also, since two or more components could be connected through multiple edge insertions within a single batch, inter-connecting insertions are checked to see if the components have already been rebuilt and relabeled by another insertion before the parallel BFS rebuild is performed.

E. Deletions and the Subgraph

Once the data structure has been initialized, edge deletions within the graph can be checked against the parents and neighbors of the involved vertices to determine if the deletion is safe. The pseudo-code for edge deletion can be found in Algorithm 3.

Here we will focus on the deleted edge $\langle s, d\rangle$ from d's perspective again assuming $Level_s \leq Level_d$. Since the graph is undirected, the same process is repeated for s. This is crucial, as a deletion marked safe in one direction may still be considered unsafe from the other.

To determine if the deletion was safe, if s is in PN_d it is removed, and PN_d is searched for a remaining parent. If a parent remains whose level is non-negative, a connection to the root of the component must exist, the deletion is safe, and nothing else needs to be done.

If d no longer has parents, a marker in the form of $Level_d \leftarrow -Level_d$,is placed to indicate this fact to the neighbors of d. This marker will be removed only when an inserted edge creates a new parent for d or PN_d is recreated during a component merger or split. If d still has neighbors, they will be checked to see that they are still valid (i.e. $Level > 0$), meaning that they have a path to the root. If such a path exists, then d has a path to the root. If so, from the perspective of d, the deletion was safe.

The first parallel for loop updates the parents and neighbors of the vertices involved in the edge deletion. Note, that the safety of the deleted edges is not confirmed by the end of this loop. Due to parallel race conditions that may cause two neighboring deleted edges to assume that they have paths to the root through each other, the data structure must be updated in the first parallel loop and safety must be checked

Algorithm 4 The algorithm for repairing the parent-neighbor subgraph when an unsafe deletion is reported.

```
Input: $G(V,E)$, $\tilde{E}_R$, $C_{id}$, $Size$, $Level$, $PN$, $Count$, $s$, $d$
Output: $C_{id}$, $Size$, $Level$, $PN$, $Count$
 1: $Q[0] \leftarrow d$, $Q_{start} \leftarrow 0$, $Q_{end} \leftarrow 1$
 2: $SLQ \leftarrow \emptyset$, $SLQ_{start} \leftarrow 0$, $SLQ_{end} \leftarrow 0$
 3: $Level[d] \leftarrow 0$, $C_{id}[d] \leftarrow d$
 4: $disconnected \leftarrow$ true
 5: while $Q_{start} \neq Q_{end}$ do
 6:     $Q_{stop} \leftarrow Q_{end}$
 7:     for $i \leftarrow Q_{start}$ to $Q_{stop}$ in parallel do
 8:         $u \leftarrow Q[i]$
 9:         for each neighbor $v$ of $u$ do
10:             if $C_{id}[v] = C_{id}[s]$ then
11:                 if $Level[v] \leq$ abs($Level[d]$) then
12:                     $C_{id}[v] \leftarrow C_{id}[d]$
13:                     $disconnected \leftarrow$ false
14:                     $SLQ[SLQ_{end}] \leftarrow v$
15:                     $SLQ_{end} \leftarrow SLQ_{end} + 1$
16:                 else
17:                     $C_{id}[v] \leftarrow C_{id}[d]$
18:                     $Count[v] \leftarrow 0$
19:                     $Level[v] \leftarrow Level[u] + 1$
20:                     $Q[Q_{end}] \leftarrow v$
21:                     $Q_{end} \leftarrow Q_{end} + 1$
22:             if $Count[v] < thresh_{PN}$ then
23:                 if $Level[u] < Level[v]$ then
24:                     $PN_v[Count[v]] \leftarrow u$
25:                     $Count[v] \leftarrow Count[v] + 1$
26:                 else if $Level[v] = Level[v]$ then
27:                     $PN_v[Count[v]] \leftarrow -u$
28:                     $Count[v] \leftarrow Count[v] + 1$
29:     $Q_{start} \leftarrow Q_{stop}$
30: if $disconnected$ then
31:     $Size[d] \leftarrow Q_{end}$
32: else
33:     for $i \leftarrow SLQ_{start}$ to $SLQ_{end}$ in parallel do
34:         $C_{id}[i] \leftarrow C_{id}[s]$
35:     while $SLQ_{start} \neq SLQ_{end}$ do
36:         $SLQ_{stop} \leftarrow SLQ_{end}$
37:         for $i \leftarrow SLQ_{start}$ to $SLQ_{stop}$ in parallel do
38:             $u \leftarrow SLQ[i]$
39:             for each neighbor $v$ of $u$ do
40:                 if $C_{id}[v] = C_{id}[d]$ then
41:                     $C_{id}[v] \leftarrow C_{id}[u]$
42:                     $Count[v] \leftarrow 0$
43:                     $Level[v] \leftarrow Level[u] + 1$
44:                     $SLQ[SLQ_{end}] \leftarrow v$
45:                     $SLQ_{end} \leftarrow SLQ_{end} + 1$
46:                 if $Count[v] < thresh_{PN}$ then
47:                     if $Level[u] < Level[v]$ then
48:                         $PN_v[Count[v]] \leftarrow u$
49:                         $Count[v] \leftarrow Count[v] + 1$
50:                     else if $Level[v] = Level[v]$ then
51:                         $PN_v[Count[v]] \leftarrow -u$
52:                         $Count[v] \leftarrow Count[v] + 1$
53:         $Q_{start} \leftarrow Q_{stop}$
```

in a secondary parallel loop by verifying that each vertex has it least one parent or valid neighbor.

For each unsafe deletion, the parent-neighbor graph needs to be corrected. This is done using a partial parallel breadth first traversal for which the pseudo-code can be found in Algorithm 4. In an early version of the algorithm, the approach was instead to perform a full search and simply rebuild the component, but the performance of this approach was found to be inferior to the presented approach.

The goal of this search is to find connections from the starting vertex d back to the root of the component by searching for other vertices in the same level that still have parents or vertices in the level closer to the root.

Initially, d is marked as the root of a new component and a BFS is begun to update PN data structure and component labels. If a connection to the original component is not found, the component is split into two and a new component rooted at d is created by this search process. If the search finds a connection back to the original component, the first search ends and a second traversal is started to relabel and rebuild part of the original component. The second traversal begins from the set of vertices found in that last frontier of the first and proceeds backward toward d. The resulting sizes of the breadth first searches are used to reconcile the component sizes. No vertices closer to the root than the level of d will ever be added to the search or relabeled. This limits the work of the search in the average case. Since unsafe deletions are processed consecutively, the parents and component labels are quickly checked before processing an unsafe delete to determine if the unsafe condition has already been repaired. As a result, in the worst case, the combined number of edges traversed by all searches in this step is limited to the number of edges in the graph; thus, the worst-case performance is equal to that of a static re-computation. As a slight optimization, vertices are checked to see if they are of degree zero and are directly initialized to being their own components.

Table II
GRAPH SIZES USED IN OUR EXPERIMENTS FOR TESTING THE ALGORITHM. MULTIPLE GRAPHS OF EACH SIZE WERE USED.

	Totals edge per average degree			
Vertices	8	16	32	64
2M	16M	32M	64M	128M
4M	32M	64M	128M	256M
16M	128M	256M	512M	—

III. EXPERIMENTAL METHODOLOGY

A. Synthetic Graphs and Experiments

Due to proprietary constraints, researchers do not always have access to real social networks for investigation and many of the datasets that are available are static. Furthermore, use of a single data set can limit the applicability of experiments. Instead, synthetic networks are used. Generating synthetic networks gives experimenters control over the size and properties of the network. Many works have been written using the Erdös-Rènyi (ER) [13], [14] model which uses a uniform random distribution for generating edges; however, this tends to create one well-connected component in which it is unlikely that an insertion or deletion would ever connect or disconnect any components. As such, we do not use this type of random graph.

In this work, we use an implementation of the Recursive Matrix (R-MAT) [7] synthetic random graph generator. This generator recursively divides the adjacency matrix into quadrants, randomly selects one of these quadrants with probabilities a, b, c, and d, and continues this process recursively until the selected quadrant is of size one. For our experimentation , we have used $a = 0.55, b = c = 0.1$, and $d = 0.25$. R-MAT graphs mimic the structure of real social networks in that they have a skewed degree distribution that follows a power law and tend toward one large component and many smaller components and singletons.

In this paper, we vary the size of the graph in terms of its scale S and edge factor E, where the number of vertices is 2^S and the number of edges is $E \cdot 2^S$. E thus corresponds to the average degree. We used scales 21, 22, and 24 with edge factors 8, 16, and 32 (qualitative results also include edge factor 64). We refer to these graphs as R-MAT-21, R-MAT-22, and R-MAT-24 with the average adjacencies. The sizes of these graphs are listed in Table II. We generated three graphs and an update stream for each scale and edge factor combination using different random seeds. For example for an R-MAT-21 graph with $E = 8$, three graphs and three streams were generated.

An update stream consists of a series of edges to be inserted or deleted. A fixed probability p_{delete} is used to determine whether or not an update will be a deletion. Deletions are selected from previous insertions.

In [10], [11] batches of $100K$ updates are used with $p_{delete} = 6.25\%$. For consistency, we use these parameters as well. For each graph, 10 batches of $100K$ are used. RMAT can potentially duplicate existing edges. We ignore these as they already are in the graph. A single deletion removes an edge regardless of the number of times that it has been inserted.

The system used for our tests is a quad-socket system with four 16-core AMD Opteron 6282 SE processors for a total of 64 cores running at 2.6GHz. Each core has a private 1MB L2 cache, and each processor has a shared 16MB L3 cache. The system has 256GB of DDR3 RAM running at 1600MHz.

B. First Attempts

During the creation of our new algorithm, we attempted several other approaches that were rejected due to being too computationally demanding, requiring a full static recomputation for each batch of $100K$, or having limited parallel scalability. These are presented here with a focus on how deletions are handled:

1) Adjacency list intersection in the hope of finding two-hop connecting paths. A similar approach with reasonable performance was shown in [11]; however, at batches of $100k$ it produces 750 unsafe deletes on average, thus requiring a full static recompute.

2) Maintain a spanning tree for each component. If a deleted edge is not in a tree, then it is considered safe. If the edge is in the tree, then the tree and affected components must be recomputed. In our experiments this approach is able to mark 90% of all deletions as safe.

3) Maintain two independent spanning trees for each connect component. Simply, find a spanning tree T, remove T from G to create G', and find a second spanning tree T' in G'. Deletions are safe until a vertex has no parent in either tree. When this occurs, the trees are recomputed from scratch. This approach is able to mark 99.7% of the deletes as safe, but this is not enough. This approach is also computationally demanding relative to others.

4) Attempting a BFS from one or both vertices to find a path between them. Given the low diameter, power law distribution, and large single component tendency, this can quickly encompass the entire component and most of the graph.

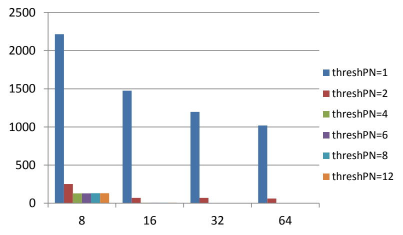


Figure 2. Average number of unsafe deletes in PN data structure for batches of $100K$ updates as a function of the average degree (x-axis) and $thresh_{PN}$ (bars).

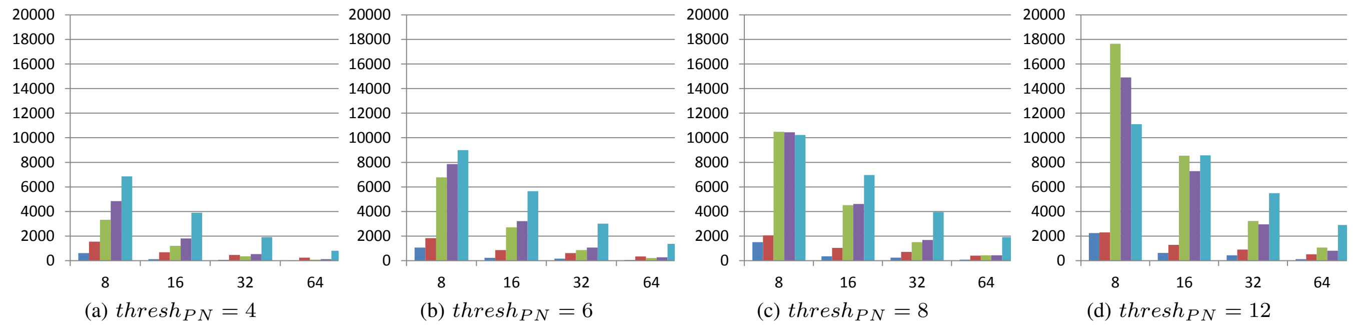
IV. Results

We present both quantitative and performance results. In the quantitative results, we count how many deletions removed relationships from the PN sub-graph, how many insertions resulted in new relationships being added to the PN sub-graph, and how many insertions resulted in a new parent replacing an existing neighbor. We also track the number of deletions reported as unsafe. In the performance results, we show speedups over static re-computation, strong scaling, and overhead given as a fraction of the total update time spent maintaining the metric.

A. Quantitative

Fig. 1 depicts quantitative results for $thresh_{PN}$ of 4, 6, 8, and 12 at different graph sizes. For a specific $thresh_{PN}$, different edge factors were tested from E = 8 to 64; these are the abscissa. Due to the similarity of the results for R-MAT-21 and R-MAT-22, we present charts only for the R-MAT-22 graphs.

We observe a trend that a decreasing number of deletions and insertions affecting the PN sub-graph as the graph



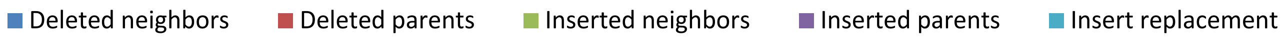


Figure 1. Average number of inserts and deletes in PN array for batches of $100K$ updates for RMAT-22 graphs. The subfigures are for different values of $thresh_{PN}$. Note that the ordinate is dependent on the specific bar chart. The charts for RMAT-21 graphs had very similar structure and have been removed for the sake of brevity.

becomes denser. This is due to the fact that the fixed-size sub-graph covers a smaller fraction of the total edges. The number of neighbor replacements that occurs steadily becomes greater than the number of inserted neighbors and inserted parents. Based on Leskovec *et al.* [22], graph densification causes a shrinkage in the graph diameter. As such, in the initial data structure creation, the number of parents that a vertex has goes up (on average) leading to fewer replacements as edges are added. It can be further inferred that as the graphs become denser $E > 64$, there will be even fewer updates made to PN.

Looking across all of the subfigures, we see that the number of insertions of parents (purple bars), neighbors (green bars), and parents replacing neighbors (light blue) increases with $thresh_{PN}$ for a fixed average degree.

As the graphs become denser, it is more likely that a deleted edge is in the PN sub-graph. This can reduce performance due to the extra work in checking for disconnections. Moreover, as the $thresh_{PN}$ increases, we see more insertions and deletions into the PN sub-graph as expected. However, the number of unsafe deletes is significantly small, meaning that the data structure does not require significant repairs.

Fig. 2 shows the number of unsafe deletions marked as a function of the average degree for different $thresh_{PN}$ for the R-MAT-22 graph. The figure for the R-MAT-21 graph is similar to R-MAT-22 and has been omitted. For any given density, there are fewer unsafe deletes as the value of $thresh_{PN}$ increases. It is clear that using $thresh_{PN} = 1$ or $thresh_{PN} = 2$ is simply ineffective. For $thresh_{PN} = 1$, each deletion from the data structure becomes an unsafe delete. Increasing $thresh_{PN}$ beyond 4 gives significantly diminished returns in terms of he number of deletes marked unsafe. For this reason, we chose $thresh_{PN} = 4$ for our performance results.

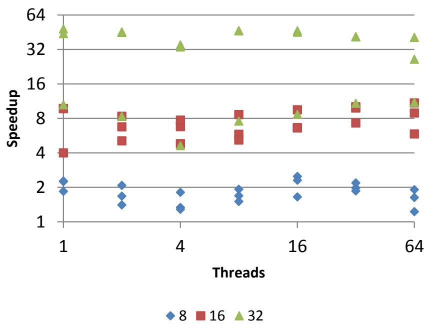


Figure 3. Strong scaling results on RMAT-22 graphs with different average degree as a function of the number of threads. Results include three graphs at each average degree.

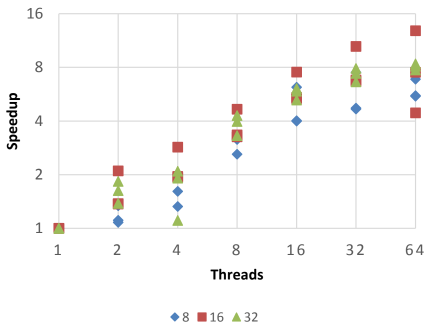


Figure 4. Speed up of the new algorithm over performing parallel static recomputation after each batch on three different RMAT-22 graphs with each average degree as a function of the number of threads.

B. Performance

In this section we present performance analysis of the parent-neighbor sub-graph approach. We demonstrate the scalability of our approach, compare performance versus a parallel static recomputation after each batch, and examine the effect of including updates to the parent-neighbor sub-graph in the update cycle.

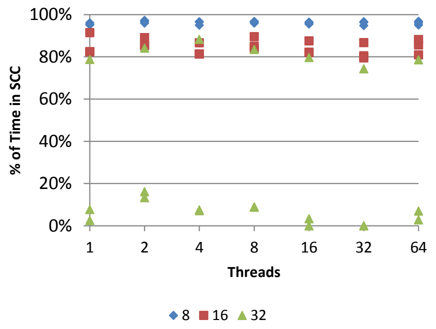


Figure 5. Fraction of the update time spent updating connected components over time spent updating the graph structure and connected components.

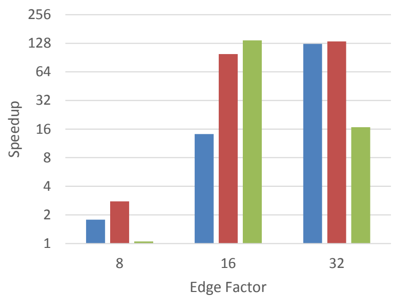


Figure 6. Speed up over performing static recomputation after each batch on scale 24 graphs for three graphs at each edge factor using 64 threads.

We compare our new algorithm to a parallel implementation based on the Shiloach-Vishkin [25] algorithm that has been experimentally determined to perform better than traditional BFS on R-MAT graphs stored the STINGER data structure. Although this implementation is not the most work-efficient, it is scalable and highly parallel with good workload balance, has low synchronization costs, and performs well for graphs with low diameter. As a reminder from the previous subsection, our results use $thresh_{PN} = 4$.

Fig. 3 gives strong scaling results (holding the amount of work constant while increasing the number of threads/cores) for our algorithm on nine different R-MAT-22 graphs, three different graphs for each edge factor (8, 16, and 32). The threads are increased in multiples of 2 from 1 to 64. The plot shows nearly linear but not optimal scaling up to 32 threads in comparison with a single thread. The speedup is $10.5x$ at 32 threads and $12.8x$ at 64 threads.

Looking across the average adjacencies, the trend is similar. At higher thread counts, increasing density results in slight improvements in the average speedup. This is due to a combination of increases in the amount of work that can be performed in parallel and fewer joined and broken components at higher densities.

Fig. 4 shows the performance improvement of using the parent-neighbor sub-graph approach over recalculating the connected components using the parallel static Shiloach-Vishkin implementation after each batch. This is shown for multiple graphs with different average degrees and is also shown at each thread count. We show that the speedup ranges from an average of 1.8x for an average degree of 8 up to 30.8x for an average degree of 32 with a maximum of 48.3x. A key insight in this graph is that the implementation of our algorithm on STINGER and our implementation of static connected components on STINGER have the same scalability. This can be inferred from that fact that the ratio between the time of the PN update and the static recomputation remains constant as the thread count is increased.

In Fig. 5, a similarly equal scalability is shown. This graph shows the percentage of the time taken to perform the PN updates in each update cycle (where the full time for the cycle also includes updating the STINGER graph structure itself). The fraction of the time taken by the PN updates at a given edge factor remains constant as the thread count is increased. As the density increases, the updates cost less time due to components splitting and merging less frequently. This is evident in the increased speedup in Fig. 4. The static cost remains constant regardless of how often components merge or split, but our approach becomes faster. At the same time, updating the data structure has increasing cost with increasing density due to the greater number of edges per vertex that must be traversed to insert or remove and edge. More information on the implementation, performance and scalability of updates in STINGER can be found in [10].

Fig. 6 shows speedup over performing static recomputation after each batch for scale 24 graphs at edge factors up to 32 using 64 threads. We see a similar speedup trend to scale 22. The variance across the graphs of the same size shows that our algorithm is more sensitive to the structure of the graph and which edges are inserted and deleted while the static algorithm is extremely consistent and load balanced. The graph also shows that denser graphs give much better results, with the third scale 24 edge factor 16 graph performing 1.26 million updates per second while tracking connected components - $137x$ faster than static recomputation.

V. Conclusions

In this work we presented a novel parallel low-memory algorithm and data structure for maintaining a labeling of the connected components in a dynamic graph. We have shown that the algorithm performs well on sparse graphs and that by tracking only a few edges per vertex ($thresh_{PN}$) the number of unsafe deletes is reduced resulting in high performance. We have shown that the new dynamic graph algorithm outperforms a well-known static algorithm and that it has the same parallel scalability. Further, we have shown good strong scaling results despite our algorithm containing some sections with only fine-grain parallelism.

Beamer *et al.* [4] have shown a BFS algorithm that searches from the undiscovered vertices once half of all vertices have been found. This outperforms traditional BFS due to a large number of edge traversals in the traditional BFS that do not find new vertices. Given that our algorithm uses a BFS in both the initial stage and the streaming stage, an efficient implementation of the Beamer algorithm for STINGER data structure should be investigated.

Acknowledgements

Funding was provided by the U.S. Army Research Office (ARO) and Defense Advanced Research Projects Agency (DARPA) under Contract Number W911NF-11-C-0088. The content of the information in this document does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] R. Albert, H. Jeong, and A. Barabási, "Internet: Diameter of the world-wide web," *Nature*, vol. 401, no. 6749, pp. 130–131, Sep 09 1999.

[2] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos, "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation," Georgia Institute of Technology, Tech. Rep., 2009.

[3] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[4] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–10.

[5] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[6] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Computer Networks*, vol. 33, pp. 309 – 320, 2000.

[7] D. Chakrabarti, Y. Zhany, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SIAM Proceedings Series*, 2004, pp. 442–446.

[8] M. B. Dillencourt, H. Samet, and M. Tamminen, "A general approach to connected-component labeling for arbitrary image representations," *Journal of the ACM (JACM)*, vol. 39, no. 2, pp. 253–280, 1992.

[9] D. Ediger, K. Jiang, J. Riedy, and D. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–8.

[10] D. Ediger, R. McColl, J. Riedy, and D. Bader, "Stinger: High performance data structure for streaming graphs," in *Proc. High Performace Embedded Computing Workshop (HPEC 2012)*, Waltham, MA, Sep. 2012.

[11] D. Ediger, J. Riedy, D. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, may 2011, pp. 1691 –1699.

[12] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, "Sparsification-a technique for speeding up dynamic graph algorithms," *Journal of the ACM (JACM)*, vol. 44, no. 5, pp. 669–696, 1997.

[13] P. Erdös and A. Rényi, "On random graphs I," *Publicationes Mathematicae*, pp. 290–297, June 1959.

[14] ——, "The evolution of random graphs," *Magyar Tud. Akad. Mat.*, pp. 17–61, 1960.

[15] P. Ferragina, "Static and dynamic parallel computation of connected components," *Information processing letters*, vol. 50, no. 2, pp. 63–68, 1994.

[16] O. Green, R. McColl, and D. A. Bader, "A fast algorithm for incremental betweenness centrality," in *Proceedings of the 4th ASE/IEEE International Conference on Social Computing,*, ser. SocialCom '12, 2012.

[17] M. R. Henzinger, V. King, and T. Warnow, "Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology," *Algorithmica*, vol. 24, pp. 1–13, 1999.

[18] M. R. Henzinger and V. King, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *J. ACM*, vol. 46, no. 4, pp. 502–516, Jul. 1999.

[19] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Commun. ACM*, vol. 22, no. 8, pp. 461–464, Aug. 1979.

[20] J. Hopcroft and R. Tarjan, "Algorithm 447: efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973.

[21] C. P. Kruskal, L. Rudolph, and M. Snir, "Efficient parallel algorithms for graph problems," *Algorithmica*, vol. 5, no. 1-4, pp. 43–64, 1990.

[22] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1.

[23] L. Roditty and U. Zwick, "A fully dynamic reachability algorithm for directed graphs with an almost linear update time," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, ser. STOC '04. New York, NY, USA: ACM, 2004, pp. 184–191.

[24] Y. Shiloach and S. Even, "An on-line edge-deletion problem," *J. ACM*, vol. 28, pp. 1–4, January 1981.

[25] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982.