2018 IEEE International Parallel and Distributed Processing Symposium

Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling

Michael Sutton
Department of Computer Science
The Hebrew University of Jerusalem
Email: msutton@cs.huji.ac.il

Tal Ben-Nun
Department of Computer Science
ETH Zurich
Email: talbn@inf.ethz.ch

Amnon Barak
Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel

***Abstract*—Connected component identification is a fundamental problem in graph analytics, serving as a basis for subsequent computations in a wide range of applications. To determine connectivity, several parallel algorithms, whose complexity is proportional to the number of edges or graph diameter, have been proposed. However, an optimal algorithm may extract graph components by working proportionally to the number of vertices, which can be orders of magnitude lower than the number of edges. We propose Afforest: an extension of the Shiloach-Vishkin connected components algorithm that approaches optimal work efficiency by processing subgraphs in each iteration. We prove the convergence of the algorithm, analyze its work efficiency characteristics, and provide further techniques to speed up processing graphs containing a huge component. Designed with modern parallel architectures in mind, we show that the algorithm exhibits higher memory locality than existing methods. Using both synthetic and real-world graphs, we demonstrate that Afforest achieves speedups of up to 67x over the state-of-the-art on multi-core CPUs (Broadwell, POWER8) and up to 23x on GPUs (Pascal).**

***Keywords*-Connected Components; Parallel Algorithms; Graph Algorithms;**

I. Introduction

Large-scale data processing in many fields frequently requires working with graph structures. One of the basic building blocks in graph processing is identifying the Connected Components (CC) of a given graph $G = (V, E)$, used as the entry point for many computations.

Several approaches for solving the CC problem have been proposed, including graph traversal [1], Min-Label Propagation (LP) [2], the Shiloach-Vishkin (SV) algorithm [3], and others [4]. While the approaches are inherently different from one another, they ultimately traverse the entire graph, processing all edges once, or even several times. However, in order to compute the CCs of the graph, it is not necessary to traverse all the edges, since one path is enough to conclude connectivity. Rather, the optimal work should be proportional to the number of vertices, which may be orders of magnitude lower than the number of edges.

In this paper, we attempt to increase the work efficiency of parallel CC by restructuring and extending the Shiloach-Vishkin algorithm. The proposed parallel algorithm, named **Afforest**[1], modifies the convergence logic so that it can be efficiently applied on subgraphs separately. This property is then used to approximate components using sampled subgraphs, thereby decreasing redundant edge processing while still obtaining the exact solution.

We perform a comprehensive analysis of Afforest, using theoretical models and empirical results. In particular, we show that the algorithm can perform most of the component identification with work proportional to $|V|$ alone, whereas the remainder can be obtained with $\mathcal{O}(|V|)$ to $\mathcal{O}(|E|)$ complexity, depending on graph topology. The empirical results depict the characteristics of the algorithm, showing that the memory access pattern is geared towards modern parallel architectures, and that the majority of the work completes after a small constant number of subgraph iterations.

To demonstrate the performance of Afforest, we measure its running time on various synthetic and real-world data, including random graphs, road maps, web graphs, and large-scale social networks. We show that the performance gain of Afforest is consistent between three different shared-memory multi-core architectures: Intel Broadwell CPUs, IBM POWER8 CPUs, and NVIDIA Pascal GPUs.

The contributions of this paper are as follows:

- We introduce Afforest: an extension of the Shiloach-Vishkin algorithm, optimized for modern parallel architectures.
- We prove the convergence of Afforest and provide in-depth analysis of its work-efficiency.
- We show how subgraph sampling dramatically decreases the number of edges processed in graphs containing a huge component.
- Results are shown on three architectures, demonstrating up to 67× speedup over current state-of-the-art.

II. Connected Components Algorithms

A. Statement of the Problem

A correct solution to the CC problem on an undirected graph G involves assigning each vertex a label ℓ s.t. if there exists a path between two vertices $u, v \in V$, then $\ell(u) = \ell(v)$. Notations used in this paper can be found in Table I.

The prominent algorithms for solving the CC problem can be roughly categorized into two approaches: (a) *tree-hooking*, where an auxiliary data structure representing in-

[1]Available online at https://www.github.com/michaelsutton/afforest

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00012

Table I: Definitions and Notations

Symbol	Description
D	Diameter of graph $G = (V, E)$.
C	Number of connected components in G.
c_i	Set of vertices in component i in V.
c_{max}	Largest component in G ($\arg\max_i \lvert c_i \rvert$).
$\ell(u)$	Final component label of vertex u.
$\mathcal{N}(u)$	Neighborhood of u, i.e., all vertices v s.t. $(u, v) \in E$.

termediate component trees (commonly named π) is iteratively updated as the graph is processed; and (b) *traversal*, where component labels are propagated by visiting vertex neighbors (either BFS or DFS). We note that the latter approach exhibits higher work efficiency, as each edge is visited exactly once during the algorithm.

B. Parallel CC Algorithms

Min-Label Propagation (LP) [2], [5] is a parallel modification to sequential graph traversal, where labels are propagated between all vertices in parallel. Each vertex is initialized with a unique label, and propagation begins in parallel following a minimum-label conflict resolution rule. The algorithm proceeds in iterations until no changes are made. The overall work done by LP is $\mathcal{O}(D \cdot |E|)$. Data-driven [6] approaches for LP reduce the amount of work performed in each iteration, at the cost of maintaining a frontier of active vertices. LP is known for its scalability in distributed-memory environments [2], as it only requires size-1 vertex halos to compute. On the other hand, the algorithm highly depends on the graph diameter, since each "winning" label must be propagated through all paths between connected vertices.

In **BFS-CC**, connected components are identified by propagating in parallel from a single root vertex until an entire component is visited, sequentially traversing components until no vertices remain unvisited. Unlike LP, BFS does not require conflict resolution, at the expense of serialization between components. In cases where the graph has a low diameter and a limited number of large components, the amount of potential parallelism in BFS-CC is high, and this approach proves to be highly efficient. Moreover, the direction-optimizing variant of BFS [1], [7] (**DOBFS-CC**) may avoid processing edges by performing "bottom-up" searches, reducing the work to be sub-linear in $|E|$.

Shiloach and Vishkin (SV) [3] introduced the original *tree-hooking* parallel algorithm for CC based on two operations: *hook* and *shortcut*. Rather than propagating values through the graph, SV transforms the input graph into trees, iteratively connecting those trees and reducing their depth. The algorithm converges when the graph is converted into a forest of depth-one trees, each representing a different component. The root of each such tree can then be viewed as

procedure Shiloach-Vishkin(V, E):

```
1:  for all v ∈ V : π(v) ← v
2:  hooking ← true
3:  while hooking do
4:      hooking ← false
5:      for all u ∈ V do in parallel
6:          for all v ∈ N(u) do in parallel
7:              if π(u) < π(v) and
    π(v) = π(π(v)) then
8:                  π(π(v)) ← π(u)          ▷ hook phase
9:                  hooking ← true
10:             end if
11:         end for
12:     end for
13:     for all v ∈ V do in parallel
14:         while π(π(v)) ≠ π(v) do
15:             π(v) ← π(π(v))              ▷ shortcut phase
16:         end while
17:     end for
18: end while
19: return π
```

Figure 1: Shiloach-Vishkin Algorithm

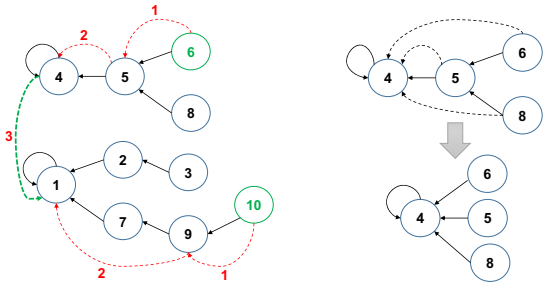
the resulting unique component label. Under PRAM modeling, SV admits a best-time complexity of $\mathcal{O}(\log(|V|)$ with $|V| + 2|E|$ processors and total work of $\mathcal{O}(\log(|V|) \cdot |E|)$.

Fig. 1 lists the SV algorithm, as implemented by [8], [9]. The vector π represents parent-pointing trees, and is initialized to $|V|$ self-pointing trees (line 1). During the *hook* phase (line 8), various neighbor edges of v may compete on the assignment $\pi(\pi(v)) \leftarrow \pi(u)$, each with a different u, but only one succeeds at each iteration. Since hooking is followed by tree compression (*shortcut*, line 15), eventually all competing edges hook, after which SV converges.

To summarize, as opposed to sequential algorithms, which perform work linear in $|E|$, both the SV and LP algorithms perform super-linear work in $|E|$, depending on $\log(|V|)$ or the diameter D, respectively. BFS-CC and DOBFS-CC, on the other hand, maintain linear work efficiency in their parallel counterparts with the disadvantage of limited parallelism.

III. Afforest: Core Algorithm

While SV is highly amenable to parallelism, and works well in theory independently of graph topology, the original formulation may not be well-suited for modern hardware, as its design heavily relies on the PRAM model and the existence of $\mathcal{O}(|E| + |V|)$ processors. In this section, we introduce the core of the Afforest algorithm, which extends and restructures the Shiloach-Vishkin (SV) algorithm for contemporary processors, and prove its convergence.



(a) Link Edge (6,10) (b) Compress

Figure 2: Link and Compress Illustrations

As in SV, Afforest connects trees and compresses them in successive order. However, in contrast to the original algorithm, the control flow and convergence logic are handled locally by the processor at each edge, so as to avoid iterating over the same edge multiple times.

The *link* procedure is portrayed in Fig. 2a and implemented in Fig. 3. Given an edge (u, v), *link* ensures that u and v are within the same component tree in π, or connects them otherwise. As opposed to SV, overriding concurrent work is avoided by using atomic compare-and-swap conditional writing operations, as proposed in [10].

The procedure searches for the tree root with the higher index, and attempts to connect it to the other tree (if it is not already connected). In Fig. 2a, we see the process of *linking* the edge (6, 10) over trees in π. As with *hook* (Fig. 1, line 8), the process begins by walking up one parent from both u and v. However, while *hook* would defer the connection to next iteration (since vertex 9 is pointing to another parent), the *link* procedure will continue its search, "jumping" up until reaching the root (vertex 4).

Applying *link* on all edges $(u, v) \in E$ results in a single tree representing each component. From this point, the CC problem can be solved by running *compress* (Fig. 2b) on the resulting trees once to create C depth-one trees. Below, we prove these properties.

A. Proof of Convergence

For ease of notation, we define the ith ancestor in π as:

$$\pi^{(i)}(x) \equiv \underbrace{\pi(\dots\pi}_{i \text{ times}}(x)).$$

An N-cycle in π is a set of unique vertices $\{x_1, \dots, x_N\}$ s.t.

$$\pi(x_i) = \begin{cases} x_{i+1} & 1 \le i < N \\ x_1 & i = N \end{cases};$$

and therefore

$$\forall i : \pi^{(N)}(x_i) = x_i.$$

We proceed to define a condition that must always hold for the correctness of our algorithm:

procedure *link*$(edge\,(u, v), \pi)$:

```
1:  p1 ← π(u)
2:  p2 ← π(v)
3:  while p1 ≠ p2 do
4:      h ← max{p1, p2}
5:      l ← min{p1, p2}
6:      if compare_and_swap(π(h), h, l) then
7:          return
8:      end if
9:      p1 ← π(π(h))
10:     p2 ← π(l)
11: end while
```

procedure *compress*(v, π):

```
1: while π(π(v)) ≠ π(v) do
2:     π(v) ← π(π(v))
3: end while
```

Figure 3: Link and Compress Procedures

Invariant 1. $\pi(x) \le x$

Lemma 1. *If Invariant 1 holds, there are no N-cycles in π for $N \ge 2$.*

Proof: Assume there exists an N-cycle with the set $\{x_1, \dots, x_N\}$ and $N \ge 2$. According to the invariant, and since elements in the cycle are unique, for any $1 \le i < N$ it holds that $\pi(x_i) = x_{i+1} < x_i$, and so $x_N < x_1$. On the other hand, by the definition of a cycle we have $\pi(x_N) = x_1$, thus, we reach a contradiction. ■

Lemma 2. *If Invariant 1 is true, then it remains true after applying the **link** or **compress** procedures.*

Proof: Observe that the only modification to π in *link* occurs in line 6, which sets $\pi(h)$ to l if the compare-and-swap (CAS) operation succeeded. Since $l = min\{p_1, p_2\} \le max\{p_1, p_2\} = h$, the invariant holds. In the *compress* procedure, if $\forall x : \pi(x) \le x$, it follows that $\forall x : \pi(\pi(x)) \le x$, so the argument trivially holds for the assignment $\pi(v) \leftarrow \pi(\pi(v))$ in line 2. ■

Lemma 3. *At any stage of the execution, h and l in **link** represent ancestors of u and v. Namely, either h is an ancestor of u and l is an ancestor of v, or vice versa.*

Proof: p_1 and p_2 are initially assigned to direct parents of u and v respectively (lines 1–2). Then, l holds the lower index from $\{p_1, p_2\}$, h holds the higher index (lines 4–5), and p_1, p_2 are again assigned with ancestors of l and h (lines 9–10). This means that p_1 and p_2 each preserve an ancestor relation to either u or v. ■

Lemma 4. *In **link**(u, v, π), for any $u', v' \in V$, if u' and v'*

are within the same component tree, they will remain in it.

Proof: If u' and v' are in the same tree, they share a common ancestor, i.e., $\exists_{I,J} : \pi^{(I)}(u') = \pi^{(J)}(v')$. The only write to π occurs during the CAS operation in line 6, which only modifies roots (due to swap condition), so for every $i \leq I$ and $j \leq J$, $\pi^{(i)}(u')$ and $\pi^{(j)}(v')$ will not change throughout the invocation. Thus, the common ancestor, as well as the path to it, will remain unchanged as multiple processors modify π. ■

Lemma 5. *If u and v are not within the same component tree, the call **link**(u, v, π) will ensure both trees are merged.*

Proof: As long as u and v are not in the same tree, line 3 in *link* would always be evaluated to true (i.e., the loop will not exit). Given that in each iteration the parents of u and v are both traversed, and that there are no cycles (due to Lemmas 1, 2), a root h s.t. $h = \pi(h)$ and $h \geq l$ will be reached at some point.

In this stage, there are three cases to consider:

1) h remains a root and this processor succeeds in the CAS operation: h and l are now directly connected, and by Lemma 3 they represent u and v, which will now be connected.
2) Another processor connects h to the tree that l is located in, i.e., $\exists_{i,j} : \pi^{(i)}(h) = \pi^{(j)}(l)$. In this case, Lemma 4 holds for h and l, which are ancestors of u and v by Lemma 3.
3) Another processor connects h to vertex l', where l and l' do not reside in the same tree ($\forall_{i,j} : \pi^{(i)}(l) \neq \pi^{(j)}(l')$). In this case, the replacement induces $\pi(h) = l'$, and a root h' with $h' \geq l$ must exist. Thus, *link* will traverse up from h and l toward h', leading back to the initial conditions of Lemma 5. Since the number of component trees is finite and there are no cycles in π, there is a limit on the number of such unique l' vertices, and thus case (3) cannot repeat indefinitely.

■

Theorem 1. *Initializing π with $\pi(v) = v$, followed by applying **link** on each $(u, v) \in E$ in parallel, results in a single tree for each connected component in the graph.*

Proof: Invariant 1 initially holds since $\forall v : \pi(v) = v \leq v$. When applying *link* over an edge (u, v), Lemma 4 states that u and v will remain in the same tree if they are already in one, whereas in the other case, Lemma 5 states that *link* will merge the two component trees.

To complete the proof, we must ensure that for every $u', v' \in V$, if there is a path between u' and v', then they reside in the same tree, namely: $\exists_{i,j} : \pi^{(i)}(u') = \pi^{(j)}(v')$. Observe that by definition, if u' and v' are connected, then there exists a list of edges in E that constitutes this path. Thus, if all edges in this path are processed by *link*, the corresponding trees will merge, and the condition holds. ■

Theorem 2. *Applying **compress** on every $v \in V$ in parallel reduces all trees to single-level depth and does not affect tree connectivity.*

Proof: Let π' represent the state of π before the *compress* call. For a vertex v at tree depth d, its path to the root is defined by the sequence $\{\pi'^{(i)}(v)\}_{i=1}^{d}$. At each iteration i of the while loop (*compress*, line 2), $\pi(v)$ is modified from $\pi'^{(i)}(v)$ to $\pi'^{(i+1)}(v)$, until reaching the root $\pi'^{(d)}(v)$. At this point, the exit condition $\pi(\pi(v)) = \pi(v)$ for the self-pointing root holds, and the depth of v is reduced to one. Since for each vertex v the root is not modified, it follows that tree connectivity is preserved.

Observe that there are no write conflicts, since each processor writes exclusively to $\pi(v)$. However, another processor may own a node u in the path to the root of v ($u = \pi'^{(i)}(v)$ for some $1 \leq i \leq d$). In this case, $\pi(u)$ may be modified to $\pi'^{(j)}(u) = \pi'^{(i+j)}(v)$ (for iteration j of u) as it is read in line 2. Two cases exist for the value read by the processor of v: $\pi'^{(i+j)}(v)$, and $\pi'^{(i+j-1)}(v)$. In either case, the path to the root is unchanged, only shortened. ■

B. Subgraph Processing

Since *link* ensures that each processed edge corresponds to the same component tree, it is possible to iterate over E in any order, without having to reprocess edges. Thus, the graph can be partitioned to disjoint edge subsets (subgraphs) and processed independently.

Repeatedly applying *link* over subgraphs, however, may increase the depth of component trees with each tree merge, making subsequent *link* calls more costly. To overcome this issue, *compress* operations can be interleaved between *link* phases. By applying tree compression, the depth of each component tree is reduced to one, increasing the efficiency of subsequent *link*s. By the idempotent definition of *compress* (Lemma 2, Theorem 2), interleaving such rounds between *link* phases does not modify the result of the algorithm.

As we shall show, the order of the edges and their partitioning into subgraphs can adversely affect the behavior of Afforest. In the following sections, we analyze and explore various such partitioning strategies, and demonstrate how they can be used to increase the work efficiency of CC identification.

IV. Subgraph Sampling

In this section, we show that by first processing a certain subset of $\mathcal{O}(|V|)$ edges, it is possible to dramatically decrease the number of edges traversed, while still obtaining the exact solution to the CC problem.

A. Spanning Forests

We use the equivalent problem of finding a spanning forest (SF) of a graph to reason about the number of edges required for CC identification. We say that a subgraph of G *preserves connectivity*, if every two vertices $u, v \in V$

connected by a path in G remain connected in the subgraph. Similarly, a subgraph *partially preserves connectivity* if for every component c_i, $\Theta(|c_i|)$ vertices remain connected in the subgraph.

A spanning tree of a connected component $c_i \subseteq V$ is a subset of E that represents a tree subgraph of G, which includes all vertices $v \in c_i$. A SF of G is the union of such spanning trees for each connected component in the graph. The size of each spanning tree is therefore $|c_i| - 1$, and the overall number of edges of the SF is $|V| - C$, where C is the total number of components (see Table I). A SF subgraph preserves connectivity of G, since all vertices within each component c_i remain connected within the corresponding spanning tree.

There is a dual relation between finding the CCs of a graph and determining a spanning forest. For instance, tree-hooking CC algorithms can be used to find a SF by tracking the edges contributing a tree merge during the execution, effectively omitting all edges that form cycles. In the other direction, for finding the CCs of the graph, it is sufficient to only process a SF of the graph to achieve correct CC labeling, since the SF preserves connectivity.

B. Uniform Edge Sampling

We now show how for certain families of input graphs, a random subgraph with $\mathcal{O}(|V|)$ edges can be sampled, such that connectivity is partially preserved.

Let G' be a connected (single-component) d-regular graph with n vertices and m edges. Let G'_p be a random subgraph obtained from G' by independently sampling edges with a probability of p. Frieze et al. [11] proved that under mild conditions for G', if $p \geq \frac{1+\epsilon}{d}$ for some $\epsilon > 0$, then G'_p contains a connected component of size $\Theta(n)$ almost surely as n increases.

Claim 1. *For $p = \frac{1+\epsilon}{d}$, the expected number of edges in G'_p is $\mathcal{O}(n)$.*

Proof: By definition $m = \frac{d}{2}n$. Since each edge in G'_p is sampled with probability $p = \frac{1+\epsilon}{d}$, the expected number of edges in G'_p is $p \cdot m = \frac{1+\epsilon}{2}n = \mathcal{O}(n)$. ■

Generalizing to a d-regular graph G with multiple components, Claim 1 can be applied to obtain a random subgraph with partially preserved connectivity and $\mathcal{O}(|V|)$ edges.

Unlike regular graphs, in general graphs with arbitrary degree distributions, uniformly sampling random edges from E creates a bias towards vertices with higher degree. This is undesirable, as the only edge of a degree-one vertex is surely included in any SF.

C. Vertex Neighbor Sampling

In this paper we present a sampling method for general graphs. Rather than sampling each edge with equal probability, our method prioritizes edges that connect vertices with lower degrees, up to a point where an edge connecting a degree-one vertex is always selected.

Specifically, we use a random sampling method based on vertex neighborhood, in which a fixed number of random edges are selected for each vertex. This way, the $\mathcal{O}(|V|)$ random edges are equally distributed across vertices and components of the graph. In Section V-B we show that the resulting subgraph covers larger portions of each component. We also show that in the context of CC, processing this subgraph first can speed up the convergence rate of the Afforest algorithm, as well as reduce the overall number of processed edges.

D. Large Component Skipping

Afforest traverses graph edges via vertex neighborhoods, i.e., for each $u \in V$, the set $\{(u, v) : v \in \mathcal{N}(u)\}$ is processed. As a result, each unordered edge is accessed twice, once from each direction. Since *link* only needs to be applied on each edge once (Theorem 1), this redundancy can be utilized for the following theorem:

Theorem 3. *At a certain point after applying **link** on a subset of E, fix a single tree c in π representing an intermediate component. Then, for each $u \in c$, the set $\{(u, v) : v \in \mathcal{N}(u)\}$ can be skipped, that is, not processed by **link**, and Theorem 1 will still hold.*

Proof: Let (u, v) be an unprocessed edge. If $u, v \in c$, the edge is redundant and would not modify π, and can thus be skipped. If w.l.o.g. $u \in c, v \notin c$, then the edge (v, u) will be accessed from v's neighborhood since $u \in \mathcal{N}(v)$. Otherwise, both $u, v \notin c$ and thus remain unaffected. ■

Using the above theorem, a natural choice for an intermediate component to skip is the largest identified component. Typical large-scale real-world graphs [12], [13], [14] comprise one large component (c_{max}), covering >90% of the vertices, and a multitude of small components. By skipping the largest intermediate component, many edges can be omitted. This process is illustrated in Fig. 4.

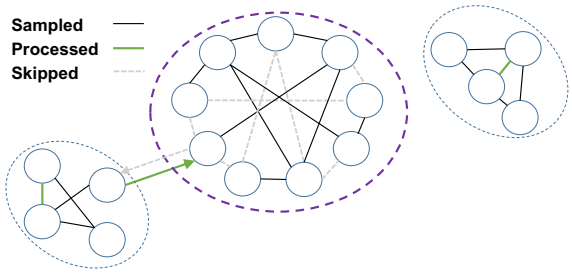


Figure 4: Subgraph Sampling for Connectivity Computation

We note that identifying the largest intermediate component in π is an $\mathcal{O}(|V|)$ task. However, since any component can be chosen to be skipped, it is sufficient to *estimate* the largest one, e.g., using probabilistic methods, rather than obtaining it exactly.

procedure Afforest($V, E, neighbor\_rounds$):

1: **for all** $v \in V : \pi(v) \leftarrow v$
2: **for** $i \leftarrow 1$ **to** $neighbor\_rounds$ **do**
3: **for all** $\{v \in V : i \leq |\mathcal{N}(v)|\}$ **do in parallel**
4: $link(v, \mathcal{N}(v)_i, \pi)$
5: **end for**
6: **for all** $v \in V$ **do in parallel**
7: $compress(v, \pi)$
8: **end for**
9: **end for**
10: $c \leftarrow most\_frequent\_element(\pi)$
11: **for all** $\{v \in V : \pi(v) \neq c\}$ **do in parallel**
12: **for** $i \leftarrow neighbor\_rounds + 1$ **to** $|\mathcal{N}(v)|$ **do in parallel**
13: $link(v, \mathcal{N}(v)_i, \pi)$
14: **end for**
15: **end for**
16: **for all** $v \in V$ **do in parallel**
17: $compress(v, \pi)$
18: **end for**
19: **return** π

Figure 5: Afforest Algorithm with Subgraph Sampling

E. Afforest with Subgraph Sampling

In order to maximize work-efficiency in the average case, neighbor sampling and large component skipping can be combined. Neighbor sampling can link most of the trees in the first iterations, after which a large intermediate component can be identified and skipped.

Based on the two above optimizations, we present the final Afforest algorithm in Fig. 5. As in SV, the algorithm begins by initializing all vertices to self-pointing, single-node trees (line 1). The algorithm proceeds by applying *link* neighbor rounds to each vertex (line 4), each of which followed by a *compress* phase (line 7) for speeding up the following *link* rounds. Upon identifying intermediate components, the algorithm performs a probabilistic search for determining the largest identified component (line 10). The search is performed by randomly sampling π a constant number of times and finding the most referenced value. This relies on the fact that all trees are depth-1 (owing to *compress* in line 7). The algorithm then executes a full *link* over the remaining edges, skipping the largest component (line 11), followed by applying *compress* to finalize convergence (lines 16-18).

V. Modeling and Analysis

In this section, we break down and analyze Afforest using both theoretical complexity bounds and empirical results. In particular, we show that while it is possible to construct worst-case (however unlikely) scenarios, Afforest exhibits several properties that allow it to perform competitively in the average case.

Table II: Average-Case Work of SV and Afforest

Graph	Shiloach-Vishkin		Afforest	
	Iterations	Max. Depth	Avg. Iter.	Max. Depth
kron	4	17	1.02	16
urand	4	29	1.19	30
web	6	24	1.01	25
twitter	4	11	1.00	14
road	9	77	1.15	126
osm-eur	9	638	1.13	754

A. Core Algorithm Complexity

As described in Section III, Afforest's core is comprised of two procedures: *link* and *compress*. To analyze their upper bound, we consider worst-case hypothetical scenarios, which coincide both in graph structure and adversarial edge order. It is worth noting that unlike BFS-based algorithms, whose worst cases are determined by D, the constructed cases do not commonly occur in real-world graphs.

For the *link* procedure, one can construct a worst case of linear $\mathcal{O}(|V|)$ work performed during linking a single edge. For instance, consider an input graph containing a depth-one tree, where the tree's root has the highest index. In an adversarial edge order, all leaf vertices will succeed connecting to the root in descending indices, so the lowest index vertex will have to walk up a linear $|V|$ depth tree before becoming the new root.

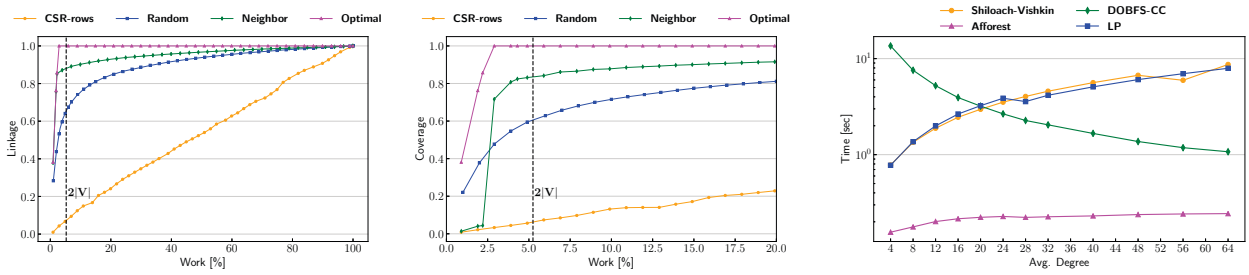
As for *compress*, it is possible to construct a similar worst-case scenario in which every processor traverses and compresses a tree with linear depth, bounding the complexity of the first invocation by $\mathcal{O}(|V|^2)$.

In the original SV algorithm, an additional step was added at each iteration to avoid such scenarios. However, more recent formulations [8] and implementations [9], [15] of SV omit this step because of its implementation complexity and its high unlikeliness. Regardless, in the average case, the expected number of iterations in SV is $\sim D/2$ [16].

Table II compares the number of iterations and maximal tree depth of SV with the average iterations and maximal tree depth in Afforest (without large component skipping), using real-world and synthetic graphs (listed in Table III). The table shows that in practice, the component tree depth in *link* is close to the depth of the trees in SV, even though *link* performs unbounded traversal on the component tree. Additionally, the average number of local (per-edge) iterations in Afforest is close to one, whereas in SV it varies. This suggests that most of the edge-processing work is performed on trees that have already converged, running a single local iteration of *link* for validation.

B. Convergence Analysis

The convergence of tree-hooking CC algorithms (a-la SV) can be characterized by the number of tree connections



(a) Linkage (b) Coverage (c) Performance w.r.t. Average Degree

Figure 6: Afforest Convergence Breakdown w.r.t. Partitioning Strategy and Graph Properties

achieved in each iteration. Below, we define two measures to evaluate the rate of convergence of such algorithms: *Linkage*, and *Coverage*.

Let T_t denote the number of trees in π after iteration t. Since the algorithm begins with self-pointing trees, $T_0 = |V|$. We also note that T_t is monotonically decreasing, with $T_\infty = C$ (after convergence). We can therefore define the ratio of trees connected following iteration t as:

$$\textit{Linkage}\,(t) \coloneqq \frac{|V| - T_t}{|V| - C}.$$

Second, given the ubiquity of giant components in graphs, we define the *coverage* measure as the largest fraction of c_{max} already belonging to a single tree at step t:

$$\textit{Coverage}\,(t) \coloneqq \frac{\tau_{max}^{(t)}}{|c_{max}|},$$

where $\tau_i^{(t)}$ is the number of vertices in tree i at iteration t and τ_{max} corresponds to the largest identified portion of c_{max}. This measure is important for understanding the earliest iteration in which *component skip* should be applied.

Using these two measures, we empirically evaluate the convergence properties of Afforest w.r.t. various subgraph partitioning strategies, as discussed in Section IV. Figures 6a and 6b plot the linkage and coverage measures during the runtime of the `web` graph (see Table III for details), which exhibits the slowest convergence rate in our measured dataset. The figures compare between four different partitioning strategies: row sampling, which partitions the graph's adjacency matrix by rows; random edge sampling with an increasing probability p; neighbor sampling, proposed in Section IV-C; and optimal subgraphs, as given by sampling a spanning forest. In the figures, the Y axis depicts the measure, whereas the X axis denotes the percentage of processed edges. Since Afforest processes each edge once, convergence is ensured after an X value of 100.

Both figures clearly illustrate the effectiveness of the neighbor sampling approach over the other strategies, attaining close-to-optimal convergence rate. After only two neighbor rounds, this strategy achieves $\sim$83% *linkage* and $\sim$80% *coverage*, vastly outperforming the other methods. Additionally, observe that adjacency matrix row sampling attains the slowest rate of convergence. This behavior is consistent with the other tested graphs.

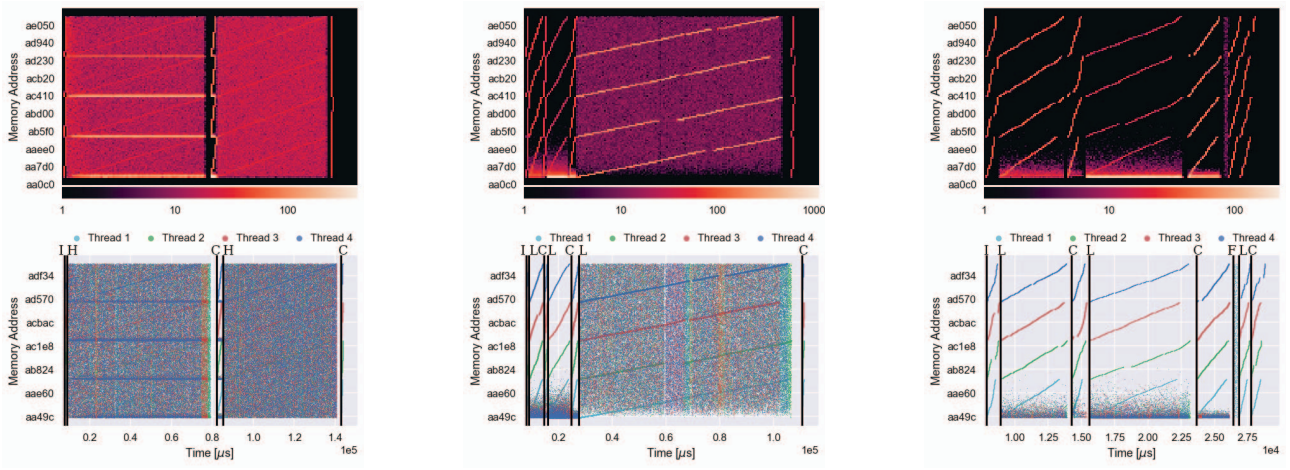
To illustrate that the effectiveness of neighbor sampling is agnostic to vertex degree, we generate Kronecker graphs with varying average degrees and plot their runtime in Fig. 6c using SV, Label Propagation (LP), DOBFS and Afforest. The figure shows that while the runtime of SV and LP correlates with the average degree, DOBFS exhibits an inverse relation, confirming previous experiments [7]; and Afforest remains largely unaffected. The correlation (in SV and LP) and lack thereof (in Afforest) is a direct result of the existence of edges that do not contribute to component identification, promoting the use of sophisticated sampling for CC identification.

C. Memory Access Pattern

Another motivating reason behind the definition of Afforest is improving memory locality and reducing contention in CC identification. In order to visualize these characteristics, we run SV and Afforest on a `urand` graph, plotting the memory access density and per-thread distribution in Fig. 7. Note that while the generated graph is small ($|V| = 2^{12}$, $|E| = 2^{16}$) to accommodate for large log-file sizes, accesses behave similarly for larger graphs with the same structure.

Specifically, Figures 7a, 7b and 7c present the memory access pattern of the parent component array π on SV, Afforest without component skipping, and Afforest, respectively. The top part of the figures shows heat-maps depicting the number of times a certain address has been accessed. In the bottom part of the figures, scatter-plots depict which thread has accessed the memory. This part also shows the stages of the algorithms (I=Initialization, L=Link, C=Compress, F=Find Largest Component, H=Hook).

From the figures it can be seen that the neighbor rounds in Afforest (first two *link*s) generate sequential memory accesses evenly distributed across threads, and accesses with high locality near the beginning of π (corresponding to tree roots). We also see that finding the largest component to skip incurs a small overhead for random sampling of π, however, the accesses that result from this process are structured.



(a) Shiloach-Vishkin (b) Afforest (without skipping) (c) Afforest

Figure 7: Memory Access Pattern of π on `urand` (best viewed in color)

In contrast to the structured access of Afforest, SV exhibits seemingly random access in *hook*, evenly distributed in π. This is a result of all edges being processed at each iteration of SV, so the total accesses to π in each iteration is higher. Additionally, in SV edges compete on tree hooks, thereby requiring multiple accesses to the same address.

VI. Performance Evaluation

This section evaluates the performance of Afforest on several graph datasets and hardware platforms.

The CPU version of Afforest is compared with the GAP Benchmark Suite [9], which contains state-of-the-art optimized CPU implementations of the BFS, DOBFS, and SV algorithms. The GPU implementation of Afforest is compared with the highly tuned, edge list-based SV code by Soman et al. [15]. We also compare the results with our own implementations of CPU Label Propagation and GPU CSR-based SV.

Our experimental setup consists of two different nodes. The first is a 2×10-core Intel Xeon E5-2630 v4 server (Broadwell architecture, SMT disabled) with 64 GB RAM and an NVIDIA Tesla P100 SXM2 16 GB GPU (Pascal architecture); and the second is a 2×10-core IBM POWER8 server (16-way SMT) with 256 GB RAM. All results report the median running time using the default configuration parameters over 16 measurements if the runtime is below 20 minutes, and the median of 3 measurements otherwise. Unless specified otherwise, the full number of cores is exclusively used for the algorithms.

Table III lists the evaluated graphs and their statistics. The datasets represent different types of graphs, including low-degree high-diameter road maps, large-scale social networks, locally-connected web graphs, and high-degree synthetic random and Kronecker graphs. For the synthetic graphs, we use the sizes defined by the GAP benchmark. Since these sizes are too large for the GPU RAM, the two datasets (`kron-gpu`, `urand-gpu`) are used there instead.

Table III: Graph Properties

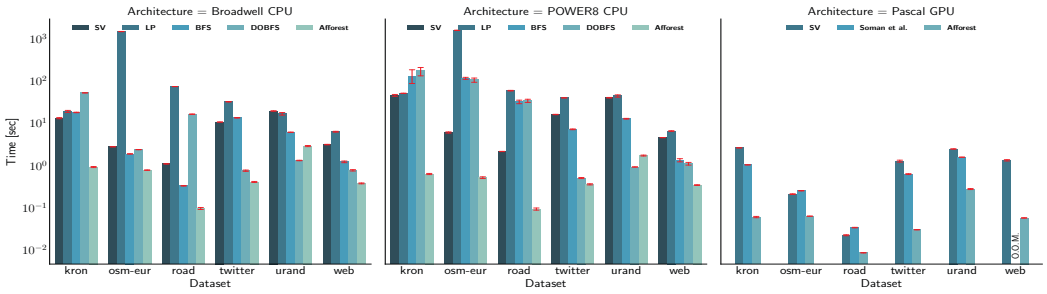
Name	Vertices	Edges	Avg. Degree	Connected Components	Size (GB)
Road Maps					
road [17]	24M	58M	2.41	2	806MB
osm-eur [18]	174M	348M	2.00	2	2.7GB
Social Networks					
twitter [19]	61.6M	1,468.4M	23.8	19.9M	12GB
Web					
web [20]	50.6M	1,949.4M	38.1	123	16GB
Synthetic Graphs					
kron	134.2M	2,111.6M	15.73	71.1M	17GB
urand	134.2M	2,147.4M	16	1	17GB
kron-gpu	67.1M	1,567.7M	23.36	30.9M	13GB
urand-gpu	67.1M	1,610.6M	24	1	13GB

A. Implementation Details

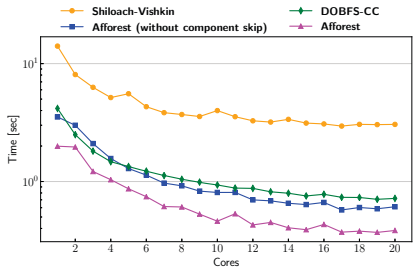
Our CPU implementation of Afforest is derived from the SV implementation found in GAP, as it is the state-of-the-art. As in GAP, C++14 and OpenMP are used for multi-threading and atomic operations, and the CSR matrix format is used for graph representation.

The GPU variant of Afforest is implemented over CUDA, using the Groute [21] framework for CSR accesses and intra thread-block load-balancing. These features are beneficial for unbalanced graphs, especially in the final *link* phase.

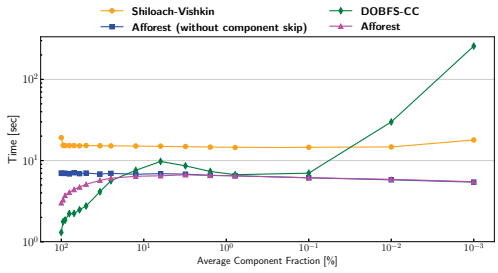
Based on the analysis in Section V, we set the value of *neighbor_rounds* in the Afforest algorithm (Fig. 5) to 2. For random neighbor sampling, we use the graph file structure by choosing the first appearing neighbors of each vertex. This choice is beneficial since the processed edges can be easily tracked to avoid reprocessing.



(a) Performance



(b) Strong Scaling (Broadwell) on `web`



(c) Performance w.r.t. Component Size on `urand`

Figure 8: Afforest Performance

B. Performance and Scaling

Fig. 8a summarizes the performance of Afforest on all three architectures, where the error-bars indicate the 25th and 75th percentiles of all runs. Overall, Afforest admits speedups of **2.49–67.24**× over Shiloach-Vishkin, and between **0.47**× slowdown and **365.97**× speedup over the state-of-the-art (non-SV), with a geometric mean of **4.99**× over all architectures. The instances where Afforest is slower are observed in `urand` vs. DOBFS, and are due to the low-diameter and single component found in the graph, combined with DOBFS-CC's direction-optimized processing.

Observe that the results are consistent between architectures, although each processor is fundamentally different than the others in terms of core count and memory system. The PRAM construction of SV, combined with the increased memory locality (Section V-C), allows Afforest to utilize inherent parallelism and cache hierarchies more efficiently.

On the GPU, Soman et al. implement SV using edge-lists instead of CSR matrices. Although more data is loaded, this representation exhibits higher data-parallelism in edge-based algorithms, trading memory access round-trips for homogeneous-work edge streaming, which is more efficient for GPUs. On the other hand, Afforest is CSR-based, but balances the load by processing the same neighbor index during each *link* round. For completeness, we include a CSR-based SV implementation, which outperforms Soman et al. in `osm-eur` and `road`. In these cases, the vertex degrees are narrowly dispersed and thus the per-vertex load is balanced. We note that the missing `web` result of Soman et al. is due to insufficient memory on the GPU for the edge-list representation.

The strong scaling of Afforest on the Intel CPU is compared with SV and DOBFS-CC in Fig. 8b. The figure shows that for the large-scale `web` graph, all algorithms attain similar speedups over multiple cores, achieving between 4.77× and 6.15× in SV and Afforest (without component skipping) respectively.

Overall, the results suggest that Afforest combines the best of SV and DOBFS: for high-diameter graphs, our algorithm efficiently compresses the graph (as in SV); whereas in high-degree graphs with large components, Afforest is able to defer edge processing or skip it altogether (as in DOBFS).

C. Large Components

In Fig. 8c, we study the effect of component size on Afforest and the other algorithms. Specifically, we generate uniformly random (`urand`) graphs with an additional parameter — average component fraction $f \in (0, 1]$ — s.t. the resulting graph has (in expectation) $\lfloor 1/f \rfloor$ components of size $\lfloor |V| \cdot f \rfloor$ and a component with the remaining vertices.

The figure reaffirms that the work efficiency of tree-hooking algorithms is not affected by the number of components and their size. BFS-based CC algorithms such as DOBFS-CC, however, inherently serialize the identification of each component. Thus, its runtime increases linearly with the number of components for $f \leq 10^{-1}$. On the other hand, the direction-optimizing characteristics of DOBFS-CC ("bottom-up" steps) enable highly efficient operation in graphs with 1–10 large components, in which it is the fastest.

The figure also shows that like SV, Afforest is unaffected by CC size. However, the large component skip heuristic is

beneficial for giant component graphs, for which it exhibits performance that is competitive with DOBFS-CC.

VII. Conclusions

The paper presented a tree-hooking connected component identification algorithm, based on the one given by Shiloach and Vishkin [3]. The proposed algorithm exhibits higher memory locality and enables processing in subgraph batches. The latter property is used to introduce a set of complementary optimizations that affect subgraph choice and decrease overall traversed edges. The evaluation suggests that the algorithm combines advantages of tree-hooking and traversal approaches, setting a new state-of-the art for both high-diameter and high-degree graphs on CPUs as well as GPUs.

The research can be extended in several directions. First, it may be possible to use insights gained from this paper to generalize the algorithm to distributed memory environments. Second, the empirical results presented in this paper demand more in-depth analysis into the computational complexity and performance model of Afforest in the average case, as well as for certain graph structures.

Acknowledgment

We thank Hussein Harake, Colin McMurtrie, and the whole CSCS team granting access to the Greina machines, and for their excellent technical support. This research was supported in part by a grant from Dr. and Mrs. Silverston, Cambridge, the UK and by the ETH Fellows postdoctoral fellowship program.

References

[1] G. M. Slota, S. Rajamanickam, and K. Madduri, "BFS and coloring-based parallel algorithms for strongly connected components and related problems," in *IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 550–559.

[2] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1821–1832, Oct. 2014.

[3] Y. Shiloach and U. Vishkin, "An O(log n) parallel connectivity algorithm," *J. Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[4] J. Shun, L. Dhulipala, and G. Blelloch, "A simple and practical linear-work parallel algorithm for connectivity," in *Proc. 26th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '14. ACM, 2014, pp. 143–153.

[5] P. Sao, O. Green, C. Jain, and R. Vuduc, "A self-correcting connected components algorithm," in *Proc. ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, ser. FTXS '16. ACM, 2016, pp. 9–16.

[6] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on GPUs," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, 2013, pp. 463–474.

[7] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE Computer Society Press, 2012, pp. 12:1–12:10.

[8] D. A. Bader, G. Cong, and J. Feo, "On the architectural requirements for efficient execution of graph algorithms," in *Proc. 2005 International Conference on Parallel Processing*, ser. ICPP '05. IEEE Computer Society, 2005, pp. 547–556.

[9] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[10] M. Sutton, T. Ben-Nun, A. Barak, S. Pai, and K. Pingali, "Adaptive work-efficient connected components on the GPU," *CoRR*, vol. abs/1612.01178, 2016. [Online]. Available: http://arxiv.org/abs/1612.01178

[11] A. Frieze, M. Krivelevich, and R. Martin, "The emergence of a giant component in random subgraphs of pseudo-random graphs," *Random Structures & Algorithms*, vol. 24, no. 1, pp. 42–50, 2004.

[12] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," Jun. 2014. [Online]. Available: http://snap.stanford.edu/data

[13] J. Kunegis, "KONECT: The koblenz network collection," in *Proc. 22Nd International Conference on World Wide Web*, ser. WWW '13. ACM, 2013, pp. 1343–1350.

[14] S. Janson, D. E. Knuth, T. Łuczak, and B. Pittel, "The birth of the giant component," *Random Structures and Algorithms*, vol. 4, no. 3, pp. 233–358, 1993.

[15] J. Soman, K. Kothapalli, and P. J. Narayanan, "A fast GPU algorithm for graph connectivity," in *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2019*, 2010, pp. 1–8.

[16] R. McColl, O. Green, and D. A. Bader, "A new parallel algorithm for connected components in dynamic graphs," in *20th Annual International Conference on High Performance Computing*, Dec 2013, pp. 246–255.

[17] "9th DIMACS Implementation Challenge." [Online]. Available: http://www.dis.uniroma1.it/challenge9/download.shtml

[18] "Karlsruhe Institute of Technology, OSM Europe Graph," 2014. [Online]. Available: http://i11www.iti.uni-karlsruhe.de/resources/roadgraphs.php

[19] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi, "Measuring user influence in Twitter: The million follower fallacy," *ICWSM*, vol. 10, no. 10-17, p. 30, 2010.

[20] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.

[21] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," in *Proc. 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. ACM, 2017, pp. 235–248.