Direction-Optimizing Label Propagation Framework for Structure Detection in Graphs: Design, Implementation, and Experimental Analysis

XU T. LIU, Washington State University, USA and Pacific Northwest National Lab, USA
ANDREW LUMSDAINE, University of Washington, USA and Pacific Northwest National Lab, USA
MAHANTESH HALAPPANAVAR and KEVIN BARKER, Pacific Northwest National Lab, USA
ASSEFAW H. GEBREMEDHIN, Washington State University, USA

Label Propagation is not only a well-known machine learning algorithm for classification, but it is also an effective method for discovering communities and connected components in networks. We propose a new Direction-Optimizing Label Propagation Algorithm (DOLPA) framework that enhances the performance of the standard Label Propagation Algorithm (LPA), increases its scalability, and extends its versatility and application scope. As a central feature, the DOLPA framework relies on the use of *frontiers* and alternates between label *push* and label *pull* operations to attain high performance. It is formulated in such a way that the same basic algorithm can be used for finding communities or connected components in graphs by only changing the objective function used. Additionally, DOLPA has parameters for tuning the processing order of vertices in a graph to reduce the number of edges visited and improve the quality of solution obtained. We present the design and implementation of the enhanced algorithm as well as our shared-memory parallelization of it using OpenMP. We also present an extensive experimental evaluation of our implementations using the LFR benchmark and real-world networks drawn from various domains. Compared with an implementation of LPA for community detection available in a widely used network analysis software, we achieve at most five times the F-Score while maintaining similar runtime for graphs with overlapping communities. We also compare DOLPA against an implementation of the Louvain method for community detection using the same LFR-graphs and show that DOLPA achieves about three times the F-Score at just 10% of the runtime. For connected component decomposition, our algorithm achieves orders of magnitude speedups over the basic LP-based algorithm on large diameter graphs, up to 13.2× speedup over the Shiloach-Vishkin algorithm, and up to 1.6× speedup over Afforest on an Intel Xeon processor using 40 threads.

CCS Concepts: • **Computing methodologies** → **Shared memory algorithms**; • **Mathematics of computing** → **Graph algorithms**.

Additional Key Words and Phrases: Community Detection, Connected Components, Label Propagation, Direction-optimizing

Authors' addresses: Xu T. Liu, Washington State University, Pullman, WA, USA, 99164 and Pacific Northwest National Lab, Richland, WA, USA, 99354, xu.liu2@wsu.edu; Andrew Lumsdaine, University of Washington, Seattle, WA, USA and Pacific Northwest National Lab, Richland, WA, USA, 99354, al75@uw.edu; Mahantesh Halappanavar, mahantesh.halappanavar@pnnl.gov; Kevin Barker, Pacific Northwest National Lab, Richland, WA, USA, 99354, kevin.barker@pnnl.gov; Assefaw H. Gebremedhin, Washington State University, Pullman, WA, USA, 99164, assefaw.gebremedhin@wsu.edu.

1084-6654/2022/10-ART39 $15.00
https://doi.org/10.1145/3564593

ACM Reference Format:
Xu T. Liu, Andrew Lumsdaine, Mahantesh Halappanavar, Kevin Barker, and Assefaw H. Gebremedhin. 2022. Direction-Optimizing Label Propagation Framework for Structure Detection in Graphs: Design, Implementation, and Experimental Analysis. *ACM J. Exp. Algor.* 9, 4, Article 39 (October 2022), 32 pages. https://doi.org/10.1145/3564593

1 INTRODUCTION

Background. The label propagation algorithm (LPA) is a machine learning approach for data classification where label information is propagated from labeled to unlabeled entities within a network [73]. The method is iterative: starting with an initial (typically small) subset of data points that have labels, the method successively propagates labels to unlabeled data points until all data points are properly labeled. Raghavan et al. [46] showed that LPA could be an effective method for identifying communities in networks. LPA has also been applied for finding other graph structures, including connected components [22, 29, 51].

Community detection is a fundamental structure and function discovery tool in network analysis. Its goal is to identify groups of tightly-knit entities—known as *communities*—in social, biological, technological and other types of complex networks. For example, video sharing services such as YouTube cluster users with similar viewing interests together to enable recommendation systems that provide better services. As sizes of networks continue to increase, we generally need fast algorithms to enable large-scale graph analytics. Two of the main advantages of LPA over many other community detection algorithms are that its run time is nearly linear in the size of the network and that it requires no *a priori* information about community structures in a network. Both of these make it practical for processing graphs with billions of edges.

Finding connected components in graphs is another well-studied fundamental problem in graph theory. Given an undirected, unweighted graph $G = (V, E)$, finding a connected component amounts to finding a labeling L such that for any two vertices u and v, $L(u) = L(v)$ if u and v are in the same connected component and $L(u) \neq L(v)$ if they are not.

Discovering connected components or community structures using LPA are essentially similar tasks. They differ only in how the objective function is defined and used. In both cases, initially, each vertex is assigned a unique label, for example, the ID of the vertex. In finding connected components, the goal then is to select the *minimum label* among a vertex's adjacent vertices. The analogous goal in the case of community detection is to select the *most frequently used* label among a vertex's adjacent vertices (or the label associated with the largest weight in a weighted graph). Therefore, the same algorithmic framework can be used to handle the two problems by appropriately defining the objective function and interpreting the labeling information L when the algorithms converge.

Proposed framework. In this paper, we propose a frontier-based Direction-Optimizing Label Propagation Algorithm (DOLPA) that enforces a desirable processing order on vertices, enhances performance, and offers flexible design for scalable implementations for community detection and connected components identification in graphs within a unified framework.

In parallel graph processing, different graph algorithms use different processing orders on vertices. One extreme is the case where the processing order is random. Another extreme is where the processing order is strictly sequential. The Parallel Label Propagation (PLP) algorithm proposed by Staudt and Meyerhenke [56], which is a parallelization of the standard LPA algorithm, belongs to the first type (random order), achieved via multi-threading. DOLPA offers a tunable middleground between the two extreme processing orders. More specifically, it provides a "knob" through which a trade-off between runtime and quality of solution is achieved. To achieve this, DOLPA maintains *frontiers* and *switches* between two abstractions, called a PULL operation and a PUSH operation. DOLPA can be used for finding either connected components or community

structures by merely customizing the objective functions. In this work, we apply DOLPA to both community detection and connected component decomposition. We experimentally show that the processing order DOLPA enforces reduces the runtime compared to state-of-the-art algorithms in both community detection and connected component decomposition, and improves the quality of solution in community detection.

To jump-start DOLPA, we need to pre-select a set of vertices as "seeds" into the initial frontier using some strategy. Selection of seeds in the context of the LP algorithm has been looked at by a few previous studies but only to a limited extent. As a part of the contributions of this paper, we propose nine seeding strategies and conduct an extensive empirical study on their performance and provide insights on their operations.

We evaluate the performance of our OpenMP DOLPA implementation and the quality of solution produced with both synthetic graphs and real-world graphs. We show that, compared with PLP, DOLPA achieves at most five times the F-Score while maintaining similar runtime. We also compare DOLPA with a state of the art parallel implementation of the Louvain method available in Grappolo [40] and find DOLPA to be both faster (speedup of ten) and more accurate (three times in F-Score). The Louvain method is based on modularity maximization.

Summary of contributions. In summary, in this work, we:

- Introduce a new LP algorithm (called DOLPA) that uses frontier and applies direction optimization to LPA for graph structure detection (§3).
- Apply DOLPA to community detection (§4) and connected component decomposition (§5).
- Propose a variety of seeding strategies for community detection grouped as *random*, *exact*, and *approximate* (§4.4). The approximate seeding strategies use subgraph sampling to approximate a parent graph and save runtime.
- Conduct extensive experimental analysis along three objectives:
  (i) Evaluate the runtime and the number of iterations obtained by DOLPA for connected component decomposition on synthetic and real-world graphs (§6);
  (ii) Empirically study how different seeding strategies behave in DOLPA in terms of time and quality of solution (§8); and
  (iii) Evaluate the runtime and quality of solution obtained by DOLPA for community detection using both synthetic and real-world graphs (§9).

The DOLPA source code, scripts to reproduce all the experiments, and the scripts to generate the plots in this paper are made available at: https://datascience.aeolus.wsu.edu/tlieu/dolpa. A portion of the material presented in this paper has appeared in the conference paper [39] that focused on only community detection. The present work extends the conference paper in two major ways. First, the DOLPA framework is extended to apply to both community detection and finding connected components. In particular, the entire discussion of connected components algorithms (§5) and their expertimental evaluations (§6) is new to this paper. Second, the proposed seeding strategies (§4.4) and their detailed evaluation (§8) is also new to this paper. In addition, the seeding strategies have informed design choices we made in crafting variants of DOLPA tailored for best runtime performance and quality of solution depending on properties of input graph and requirements of the application.

2 LABEL PROPAGATION

Loosely speaking, the goal of community detection is to find a grouping (or clustering) of the vertices in a graph in such a way that intra-connection within a group is maximized and inter-connection between groups is minimized. The problem has attracted a lot of attention in the

Algorithm 1 PLP: Parallel Label Propagation [56].
Input: Graph $G = (V, E)$, termination threshold θ
Output: $\forall v \in V$: $label[v]$ stores the label of v

```
1:  for all v ∈ V do in parallel
2:      label[v] ← id[v]
3:  updated ← |V|, V_active ← V
4:  while updated >= θ do
5:      updated ← 0
6:      for all v ∈ V_active|deg(v) > 0 do in parallel
7:          for all adjacent vertex w of v do
8:              frequency[label[w]]++
9:          l ← ARGMAX(frequency[label[w]])
10:         if label[v] ≠ l then
11:             label[v] ← l
12:             atomically increment updated
13:             V_active ← V_active ∪ {v's neighbors}
14:         else
15:             V_active ← V_active \ {v}
```

literature and a variety of different algorithms have been suggested for solving it. See the survey paper by Fortunato [14] for a comprehensive review.

One of the simplest and fastest algorithms for community detection is label propagation (LP). Garza and Schaeffer give a detailed investigation of LP-based approaches [16] and Section 10 of this paper offers a brief review of related work. Since it forms the basis for our proposed algorithms, we briefly discuss the standard LP algorithm here.

In the standard LP algorithm, each vertex is initially assigned a unique label and at every iteration a vertex adopts a label that is the most commonly used among its adjacent vertices. As the algorithm progresses, densely connected groups of nodes form a consensus on their labels and vertices that attain the same labels are grouped as communities.

Algorithm 1 outlines a parallelization of the standard LP as given in the Parallel Label Propagation (PLP) work [56]. In Line 1, a unique label is initially assigned to each vertex in the graph. In each iteration, each vertex's adjacent vertices are examined in parallel so that the label of the vertex is updated with a maximum label among its adjacent vertices (Line 6 to 15). A maximum label could be the most common label or the label associated with highest weight. The stopping criterion in Line 4 is that the number of updated labels is less than a threshold θ, an input to the algorithm. The algorithm outputs an array of vertex labels, where a set of vertices having the same label signifies membership in the same community.

Line 7–8 takes O($d(v)$) time, where $d(v)$ is the degree of the vertex v. In practice, LPA requires only a few iterations to converge, and therefore the runtime is nearly linear, i.e. $O(k(m+n))$, where k is the number of iterations, m is the number of edges, and n is the number of vertices.

3 DIRECTION-OPTIMIZING LABEL PROPAGATION

In the early iterations of LPA, instead of passively propagating labels, we choose a more "important" label for a vertex v in the network and eagerly broadcast the label to v's adjacent vertices. We abstract this kind of label update as a PUSH operation. Furthermore, in LPA, for each vertex v, the

Algorithm 2 Parallel, Direction Optimizing Label Propagation Algorithm.
Input: Graph $G = (V, E)$, seeding parameter τ, switch threshold ω.
Output: $\forall v \in V$: $label[v]$ stores the label of v

```
1:  for all v ∈ V do in parallel
2:      label[v] ← a unique value in {0, 1, ··· , |V| − 1}
3:      unset bit[v]
4:  frontier ← fraction τ of V                          ▷ Select randomly
5:  pullswitch ← False, iteration ← 0
6:  while (frontier≠ ∅) do
7:      iteration++
8:      if pullswitch = False and ω = iteration then
9:          pullswitch ← True                           ▷ Enter only once
10:     if pullswitch = True then
11:         for all v ∈ frontier do in parallel
12:             unset bit[v]
13:             Pull(v)
14:             if label[v] has changed then
15:                 set bit[v's neighbors] atomically
16:     else
17:         for all v ∈ frontier do in parallel
18:             unset bit[v]
19:             Push(v)
20:             if bit[v's neighbors] is unset then
21:                 set bit[v's neighbors] atomically
22:     frontier ← ∅
23:     for all v ∈ V do
24:         if bit[v] is set then
25:             frontier← frontier ∪ {v}
```

labels of v's adjacent vertices are queried to obtain the maximum label or the minimum label. We abstract this label update operation as a PULL operation.

To make best use of both label update operations, we use a *frontier* as a processing queue. The frontier is a set of paths originating from a given set of start vertices. By pre-selecting "seed" vertices into a frontier, we can jump-start the first iteration (we will discuss seeding strategies in §4.4). When the workload is small in the early iterations, we apply PUSH for label updates; when the workload reaches a specified threshold or the iteration number reaches a *direction switch* threshold, we switch to PULL for label updates. This idea of *direction optimization* was pioneered by Beamer et al. [6] in parallel Breadth-First Search and inspires our work.

DOLPA is summarized in Algorithm 2. The PUSH abstraction is outlined in Algorithm 3 and the PULL abstraction is given in Algorithm 4. In both Algorithm 3 and Algorithm 4, the operations are specialized for community detection. The PUSH abstraction tailored for finding connected components is given in Algorithm 5 and the variant of PULL specialized for connected components is given in Algorithm 6.

Line 1 to 5 in Algorithm 2 constitute the pre-processing steps. The purpose of Line 1 is to give a unique label to each vertex in the graph. We select a fraction τ of vertices randomly as seed vertices

Algorithm 3 Pseudocode of PUSH for community detection.

```
1: function PUSH(vertex v)
2:     for all adjacent vertex w of v do in parallel
3:         label[w] ← label[v]
```

Algorithm 4 Pseudocode of PULL for community detection.

```
1: function PULL(vertex v)
2:     for all adjacent vertex w of v do
3:         frequency[label[w]]++
4:     label[v] ← ARGMAX(frequency[label[w]])        ▷ Find the most frequent label
```

Algorithm 5 Pseudocode of PUSH for finding connected components.

```
1: function PUSH(vertex v)
2:     change ← False
3:     for all adjacent vertex w of v do in parallel
4:         if label[w] > label[v] then
5:             label[w] ← label[v]
6:             change ← True
7:     return change
```

Algorithm 6 Pseudocode of PULL for finding connected components.

```
1: function PULL(vertex v)
2:     minlabel ← label[v]
3:     for all adjacent vertex w of v do
4:         minlabel ← min(minlabel, label[w])
5:     if minlabel ≠ label[v] then
6:         label[v] ← minlabel
7:         return True
8:     return False
```

and add them to the initial frontier. The fraction τ is a *seeding parameter* passed as a part of the input. The *pullswitch* Boolean variable is initially turned off.

In each iteration, we process each vertex in the frontier and add all the adjacent vertices of the vertex to the next frontier if there is any label update. The direction optimization is triggered at Line 8 by the *switch threshold* ω, which is an input parameter. Specifically, when the number of iteration is equal to ω, *pullswitch* is turned on. Note that if ω is equal to one, no PUSH operation is applied. The stopping criterion for iterations at Line 6 is an empty frontier. The output of DOLPA is a set of labels associated with vertices, where vertices having the same label indicates that they belong to the same structure.

Complexity of DOLPA. We first analyze DOLPA's complexity in the sequential setting. Both PUSH and PULL take $O(d(v))$ time each, where $d(v)$ is the degree of the vertex v. Selecting vertices and adding to the frontier is $O(n)$ in time and also in space, where n is the number of vertices. The worst case performance of DOLPA happens when the frontier holds all of the vertices in the graph, in which case the space complexity of the frontier is $O(n)$. Therefore, the runtime of DOLPA is

bounded by $O(k(m+n))$, where k is the number of iterations, m is the number of edges, and n is the number of vertices.

Let us consider next the upper bound for k. For finding connected components, the upper bound for k is the diameter of the graph D, which is the maximum steps required for a label to propagate from one end of a connected component to the other end. For community detection, the upper bound for k is in practice close to D. However, there may be cases where two labels keep swapping (discussed shortly in Section 4.2) and k could be much larger than D.

Now we analyze DOLPA in the parallel setting using the work-depth model [20]. In the work-depth model, the work w is the total number of operations executed by a computation, and the depth d is the longest chain of sequential dependencies in the computation. If p processors are available, with a randomized work-stealing scheduler, Brent's scheduling principle dictates that the runtime is $O(w/p + d)$. In DOLPA, the overall work is $O(k(m+n))$, and the depth is $O(D \cdot logn)$. In the worst case, each frontier contains at most n vertices and there are at most D frontiers built throughout the algorithm. In this case, the depth is $O(Dn)$, and the total work is $O(D(m+n))$.

4 APPLICATION TO COMMUNITY DETECTION

We discuss in this section in more detail how we adopt DOLPA for community detection. We also outline and address various challenges that arise in parallelizing DOLPA. Finally, we discuss nine seeding strategies for community detection proposed in this work.

4.1 Frontier Expansion

In DOLPA, the next frontier expands as either a PUSH or PULL operation updates the label of a vertex. For each vertex, the algorithm inserts all of the vertex's neighbors into the next frontier without knowing whether any of its neighbors have been inserted. This situation happens in both the serial and the parallel cases. In DOLPA, we use a bitmap [43] to avoid duplicate entries in frontier expansion. A set bit indicates an ensuing insertion of that vertex, hence latter insertion attempt is aborted. Line 15 and Line 21 in Algorithm 2 show how we test and set the bitmap.

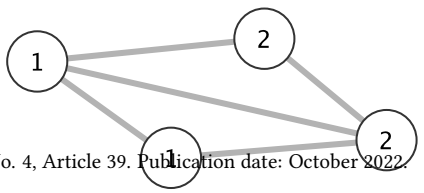
The frontier expansion rate of DOLPA determines how fast the workload increases in each iteration. The rate is decided by the branching factor of the vertices in the current frontier. (A branching factor of a vertex is the number of unvisited vertices of it.)

4.2 Local Maxima and Label Swapping

Figure 1 shows a scenario where two vertices reach a local maxima and swap labels in parallel. The upper left vertex finds label 2 as the most common in its neighborhood, while the lower right vertex finds label 1 as the most common label in its neighborhood. In this scenario, these two vertices keep swapping their labels in each iteration. This is called *label oscillation* [46]. An oscillation also happens in the serial algorithm when the input graph is bipartite. To detect and prevent label oscillation, before a label is updated, the maximum label currently received is compared with the previous label. If the maximum label is the same as the previous label, a label swap is detected. When this happens, the vertex is manually marked as inactive and the algorithm moves on.

A more practical way to handle label swapping is to ignore it. This can be achieved by applying the same stopping criterion as PLP: if the amount of labels updated in the previous iteration is less than a certain threshold, the iteration stops (Line 6). However, this stopping criterion depends on the condition that the termination threshold is greater than the number of label swapping.

Fig. 1. An illustration of swapping labels.



4.3 Parallelizing Push and Pull

To effectively parallelize PUSH and PULL, we exploit two levels of parallelism in Algorithm 2: task level (coarse-grained) and vertex level (fine-grained). The coarse-grained parallelism is reflected in Line 11 and Line 17, where a PUSH or PULL task is performed on each vertex in the frontier. The fine-grained parallelism happens *within* PUSH and PULL, where the label propagation is performed on the neighborhood of a vertex v. The reason we introduce two levels of parallelism is to address workload imbalance at the task level, where the vertices could have various number of neighbors. With the help of the two level parallelism, DOLPA achieves balanced workload.

When multiple vertices perform PUSH concurrently, they access their neighborhood in parallel. Because of the shared neighbors, these vertices will "fight" for the labels of their common neighbors. This is a benign race that does not harm correctness of the algorithm but can hurt performance [32]. Benign race entails repeated work and makes the algorithm non-deterministic.

Each PULL operation is almost data independent disregarding insertion of vertices into the next frontier. But if we allow concurrent insertions into a frontier, the order of insertion into the frontier will force a processing order on the PULL operations in the next iteration. We want to prevent this from happening. Therefore, we do not physically insert vertices into the next frontier during label propagation. Instead, we simply mark the bit belonging to that vertex. The frontier insertion actually happens, and does so only once, at the end of each iteration (at Line 25).

The frontier insertion is implemented in serial at the end of each iteration (at Line 25), instead of within multi-threading. A possible parallel implementation here is to let each thread insert the vertices into a thread-local frontier and then merge these into the global frontier in a critical section. Each thread would randomize the order of the insertion into the frontier, which is the same randomization that PLP algorithm uses. In practice, we observed that such a parallel implementation does converge faster than the serial implementation, however, the quality of the solution the parallel implementation produces is much lower than that of the serial implementation. Hence, we adopt the serial implementation in DOLPA.

Within PULL, there are concurrent reads on the labels of the neighbors of a vertex v, which is embarrassingly parallelizable. In practice, we accumulate the frequencies of the labels using std::unordered_map which does not support concurrent write. Since we already have sufficient parallelism at the coarse-grained level, we adopt serial PULL in our implementation.

4.4 Seeding Strategies

A set of seed (or influential) vertices forms the potential core of a community structure in the network. Nodes with high degree or clustering coefficient, fully-connected cliques, and maximal cliques are usually treated as seeds of a community structure [72]. In general, a node with more connections could be viewed as more important in the network [64]. It has been proven that the membership contribution of a vertex to a community is highly related to its degree [48]. However, there has not been a principled study on how a seeding strategy should be chosen and how the combination of seeding strategy and parameter affect the performance of a community detection algorithm. We propose and investigate nine seeding strategies, grouped under three categories: *random seeding strategy*, *exact seeding strategies* and *approximate seeding strategies*. Table 1 gives an overview of the strategies.

4.4.1 Random Seeding Strategy. Our baseline strategy, called *random seeding strategy*, selects seed vertices randomly. In particular, each vertex is selected to be a seed independently with equal

Table 1. Nine seeding strategies in three categories: Random, Exact and Approximate seeding.

Seeding Strategy	Description
1.1. Random	Randomly sample a fraction τ of vertices as seeds
2.1. High-degree	Sort vertices on degrees and select a fraction τ of high-degree vertices
2.2. Low-degree	Sort vertices on degrees and select a fraction τ of low-degree vertices
2.3. High-total degree	Count total degree of each vertex, sort the vertices on their total degrees and select a fraction τ of high-total-degree vertices
2.4. Low-total degree	Count total degree of each vertex, sort the vertices on their total degrees and select a fraction τ of low-total-degree vertices
3.1. High-degree sampling distribution	Randomly sample a fraction τ of vertices and sort them on degrees in descending order
3.2. Low-degree sampling distribution	Randomly sample a fraction τ of vertices and sort them on degrees in ascending order
3.3. High-total degree sampling distribution	Randomly sample a fraction τ of vertices, count total degree of these vertex, sort them on total degrees in descending order
3.4. Low-total degree sampling distribution	Randomly sample a fraction τ of vertices, count total degree of these vertex, sort them on total degrees in ascending order

probability. Since random seeding strategy has no bias over the vertices, it maintains a random order of the vertices in the frontier.

4.4.2 Exact Seeding Strategies. Kloumann and Kleinberg [24] found that performance was higher when a large fraction of seed vertices' edges were to vertices that lie within the same community. This suggests that a seed should have a good internal connectivity within the community relative to the rest of the graph. We adopt this idea and select high-degree vertices as seeds, a strategy we refer to as *High-degree seeding strategy*.

In contrast to the High-degree seeding strategy, we also propose *Low-degree seeding strategy*. The rationale for considering low-degree as a seeding strategy is that the label of a low-degree vertex has a high probability to join the community in which its neighbors belong. Due to the nature of a low-degree vertex—that it has a small number of adjacent vertices—a low-degree vertex often maintains its label information throughout the propagation process.

Gao and Zhang [15, 67] proposed selecting seed vertices based on the *total degree* of the neighborhood of a vertex. The total degree of a vertex is the sum of the degrees of its adjacent vertices. A neighborhood owning a high total degree thus means that it has a relatively high number of links to the community structure it resides in. We adopt this idea and propose selecting vertices with a high total degree of their neighborhood as a strategy, a strategy we call *High-total-degree seeding strategy*. For a similar reason as in low-degree seeding strategy, we also propose selecting vertices having a low total degree of the neighborhood as a strategy, which we call *Low-total-degree seeding strategy*.

Because the High-degree, Low-degree, High-total-degree and Low-total-degree seeding strategies all compute the exact ordering of the selected seeds within the vertex set of the graph and maintain this ordering in the initial frontier, we categorize the four strategies as *exact seeding strategies*.

4.4.3 Approximate Seeding Strategies. As a part of a pre-processing step, the cost of an exact seeding strategy can be quite high as the size of the network increases, due to the need to sort vertices based on either their degrees or the total degrees of their neighborhoods. A remedy here is to randomly sample a small portion of the graph that approximates the original graph. Two most

common sampling schemes are *subgraph sampling* and *neighborhood sampling* [25]. A subgraph sampling scheme samples each vertex independently with equal probability and observes the subgraph induced by the sampled vertices. A neighborhood sampling scheme further observes the edges between the sampled vertices and their adjacent vertices.

Combining these sampling schemes with the exact seeding strategies, we propose several *approximate seeding strategies*. These strategies are: *High-degree sampling seeding strategy*, *Low-degree sampling seeding strategy*, *High-total-degree sampling seeding strategy* and *Low-total-degree sampling seeding strategy*. The first two of these use the subgraph sampling scheme while the latter two use the neighborhood sampling scheme. In all of these approximate seeding strategies, we sample a fraction τ of the vertices as seeds and sort them based on either the degrees of the sampled vertices or the total degrees of the sampled neighborhood. Then we insert seeds into the initial frontier.

5 APPLICATION TO CONNECTED COMPONENT DECOMPOSITION

We discuss in this section in more detail how we adopt DOLPA for connected component decomposition. Finding connected components in a graph is an extensively studied problem. We focus on only parallel algorithms for the problem in this paper. We group the algorithms into four categories: 1) algorithms based on using breadth-first search graph traversal [51, 53]; 2) the Shiloach-Vishkin (SV) algorithm [50] and its variants such as Afforest [59], which is the state-of-the-art SV algorithm specifically optimized for power-law degree graphs; 3) the Minimum Label Propagation for connected component decomposition (LPCC) [9, 36, 57, 65]; and 4) others [52]. The authors of these algorithms propose many practical optimizations for finding connected components. We discuss some of these optimizations in this section and apply them to our algorithm. We also show in our discussion that the SV algorithm and LPCC are not that different; most of the optimizations that are applicable to SV can also be applied to LPCC. We discuss the ones we adopt in our algorithms.

5.1 Fast SV Algorithm

Shiloach and Vishkin [50] introduced the first parallel connected components algorithm on the Parallel Random Access Machine (PRAM) model, as shown in Algorithm 7. The SV algorithm relies on two operations on a union-find data structure: "hook" (also known as "union") and "compress" (also known as "pointer jumping"). The two operations are outlined in Line 9 and Line 18 of Algorithm 7, respectively. For a given edge, the hook operation combines vertices into trees such that the vertices in the same component belong to the same tree. The compress operation shortens the *find* path in the data structure by pointing the current vertex's parent pointer directly to the root of the tree. Both operations can be performed in parallel. The complexity of this algorithm is $O(mlogn)$, where m is the number of edges and n is the number of vertices.

5.2 PUSH, PULL, and HOOK

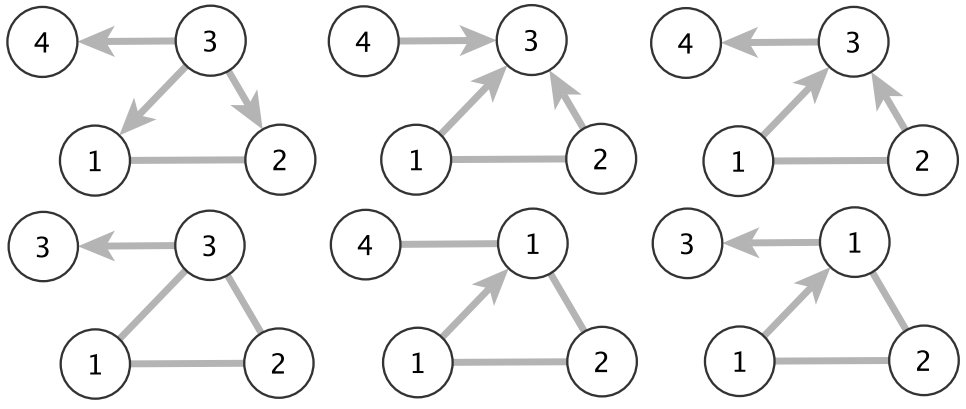
Notice that the purpose of the hook operation is essentially to "hook" the labels of two vertices: if the label of a vertex u is smaller than its adjacent vertex w, the HOOK operation pushes the label of u to vertex w; if the label of u is bigger than w, the hook operation pulls the label of w to u. In contrast, the PUSH function in Algorithm 5 only pushes the label of u to w if label[u] is smaller than that of w. The PULL function in Algorithm 6 only pulls the label of w to u if label[w] is smaller than that of u. Therefore, HOOK is a bidirectional operation combining both PUSH and PULL operations. Notice that HOOK may propagate labels two hops per iteration, where it pulls the label to itself first, then pushes the newly updated label.

Figure 2 shows how the labels are updated on a vertex (vertex 3 in the example) and its neighbors using PUSH, PULL, and HOOK operations, respectively. As we can see in the figure, each vertex starts with its own vertex ID as a label. PUSH(vertex 3) pushes label 3 to vertices 1, 2 and 4, but only vertex

Algorithm 7 Fast Shiloach-Vishkin (SV) Algorithm [50].
Input: Graph $G = (V, E)$.
Output: $\forall v \in V$: $label[v]$ stores the label of v

```
1:  for all v ∈ V do in parallel
2:      label[v] ← a unique value in {0, 1, ··· , |V| − 1}
3:  change ← True
4:  while change do
5:      for all v ∈ V do in parallel
6:          change ← Hook(v, label)
7:      for all v ∈ V do in parallel
8:          Compress(v, label)
9:  function Hook(v, label)
10:     change ← False
11:     for all adjacent vertex w of v do in parallel
12:         l ← min(label[w], label[v])
13:         h ← max(label[w], label[v])
14:         if l ≠ h then
15:             update label[h] to be l atomically
16:             change ← True
17:     return change
18: function Compress(v, label)
19:     while label[v] ≠ label[label[v]] do
20:         label[v] ← label[label[v]]
```

Fig. 2. An illustration of labels update on vertex 3 and its neighbors using PUSH, PULL, and HOOK respectively. Each vertex starts with its own vertex ID as the initial label. The arrow in the figure represents the direction of the label propagation. The subfigures at top are the labels of vertices at the beginning of each operation. The subfigures at bottom are the labels of the vertices at the end of each operation.



(a) PUSH(vertex 3) (b) PULL(vertex 3) (c) HOOK(vertex 3)

4 updates its label because it has a bigger label than 3. PULL(vertex 3) pulls labels from vertices 1, 2 and 4, and update its label to be the smallest one among them, 1. HOOK(vertex 3) pushes label 3 to

Algorithm 8 Pseudocode of a combination of PUSH and PULL, aka HOOK.

```
1:  function Hook(vertex v, label)
2:      change ← False
3:      for all adjacent vertex w of v do in parallel
4:          if label[w] > label[v] then
5:              label[w] ← label[v]
6:              change ← True
7:          if label[w] < label[v] then
8:              label[v] ← label[w]
9:              change ← True
10:     return change
```

vertex 4, which has a bigger label than vertex 3, and pulls labels of vertices 1 and 2, which have smaller labels than vertex 3. The label update in all three operations are *monotonic*, meaning that the labels being updated by these three operations never increase. Because of this, the order of visiting vertex 1 and vertex 2 does not matter, because the label of vertex 3 is going to end up with the smaller label among vertex 1 and vertex 2 at the end of the neighborhood traversal.

If we combine PUSH (Algorithm 5) with PULL (Algorithm 6) for connected component decomposition, we end up with Algorithm 8, which is equivalent to the hook operation in SV algorithm. Based on the observations we have made about the relationships among the operations PUSH, PULL, and HOOK, the SV algorithm is indeed a LP-based algorithm for connected component decomposition. Therefore, any optimizations/heuristics that are proposed for the SV algorithm are also applicable to LP-based algorithms.

5.3 Combining PUSH, PULL, and HOOK with Compaction Methods

Recent research [21] proposed randomized concurrent algorithms for disjoint set union, which combines the compaction methods with find operations. There are three classical compaction methods: *path compression*, *path splitting*, and *path halving*. The works [12, 71] have shown that the HOOK operation can also be combined with various compaction methods. With these compaction methods, the authors have shown that SV can take fewer iterations to reach convergence. We refer to this method as HOOKANDCOMPRESS.

Adopting the same idea, we propose to combine one of the compaction methods with our PUSH and PULL operations. We show in Algorithm 9 the PULL operation combined with full path compression; the algorithm is referred to as PULLANDCOMPRESS. Similarly, the algorithm PUSHANDCOMPRESS combines PUSH with full path compression; its pseudocode is omitted for space considerations. Algorithm PULLANDCOMPRESS pulls the smallest label among v's neighbors, then updates the labels of the labeling path until its root to be the smallest label met along the path. Similarly, PUSHANDCOMPRESS pushes v's label to its neighbors as well as every vertices on the labeling path until the root, only if its label is smaller. Previously, without path compression, each label can only propagate one hop per iteration. With full path compression, each label can propagate O($logn$) hops per iteration, where O($logn$) is the depth of the labeling path from the root to the leaf. In the best case, such as a line graph, each label can propagate O(D) hops per iteration, where D is the diameter of the graph, making the LP-based algorithm with path compression converge in one iteration. Other compaction methods can also be combined with the PUSH and PULL operations. We leave this as the future work. Because the full path compression will flatten the connected

Algorithm 9 Pseudocode of PULL with full path compression for finding connected components.

```
1:  function pullAndCompress(vertex v)
2:      minlabel ← label[v]
3:      for all adjacent vertex w of v do
4:          minlabel ← min(minlabel, label[w])
5:      p1 ← label[v]
6:      p2 ← minlabel
7:      while p1 ≠ p2 do
8:          l ← min(p1, p2)
9:          h ← max(p1, p2)
10:         label_h ← label[h]
11:         if label_h = l or compareAndSwap(label[h], h, l) then
12:             return True
13:         p1 ← label[label[h]]
14:         p2 ← label[l]
15:     return False
```

Algorithm 10 Parallel, Pull-based Label Propagation Algorithm with Compress.
Input: Graph $G = (V, E)$.
Output: $\forall v \in V$: $label[v]$ stores the label of v

```
1: for all v ∈ V do in parallel
2:     label[v] ← a unique value in {0, 1, ··· , |V| − 1}
3: change ← True
4: while change do
5:     for all v ∈ V do in parallel
6:         change ← pull(v)          ▷ pull can also be replaced with push or hook
7:     for all v ∈ V do in parallel
8:         compress(v, label)
```

component labeling tree into a star, we expect that pullAndCompress/pushAndCompress would require fewer iterations to reach convergence compared to the basic push and pull operations.

5.4 Slow Convergence for Large Diameter Graphs

A known issue of LPCC is its slow convergence for large diameter graphs [57], where the diameter D of the graph is a large integer. Recall that the goal of finding connected components in a graph using Label Propagation is to propagate the minimum label of each component to its members. At the end of the algorithm, every vertex having the same label belongs to the same component. For LPCC, each label is propagated one hop per iteration. For any graph, the minimum label requires $O(D)$ steps to propagate from one end of the largest component to the other. Therefore, a basic LPCC would converge in $O(D)$ iterations, resulting $O(D(m + n))$ total work. In the worst case, such as in line graphs, where the diameter is n, LPCC converges in $O(n)$ iterations.

One solution to address this issue is to contract the original graph repeatedly. However, contraction would mutate the input graph. We are interested in solutions that do not mutate the input graph. Another solution is to use a compress operation in the SV algorithm. Sterggiou et al. [57] introduce this in their distributed LPCC algorithm as "shortcut". They have proven this would

Algorithm 11 Pseudocode of subgraph sampling to find the largest component ID.
Input: Intermediate component IDs label, an integer *neighbor_rounds* (default value = 2)
Output: the largest component ID l

```
1:  function GenerateSubgraphViaPush(label, neighbor_rounds)
2:      for all v ∈ V do in parallel
3:          for all first neighbor_rounds adjacent vertex w of v do
4:              label[w] ← label[v]
5:      for all v ∈ V do in parallel
6:          compress(v, label)
7:      return label
8:  function GenerateSubgraphViaPull(label, neighbor_rounds)
9:      for all i ← 1 to neighbor_rounds do
10:         for all v ∈ V do in parallel
11:             Pull(v)
12:     for all v ∈ V do in parallel
13:         compress(v, label)
14:     return label
15: function SampleFrequentElement(label)
16:     frequency ← []
17:     for all i ← 1 to 1024 do
18:         generate a random vertex v from 0 to |V|
19:         frequency[label[v]]++
20:     l ← argmax(frequency[label[v]])          ▷ Find the most frequent label
21:     return l
```

guarantee LPCC to converge in O($logn$) iterations with O($mlogn$) total work. We implement this approach in our algorithms to speedup convergence (see Algorithm 10).

5.5 Skipping the Largest Component in Scale-free Graphs

In scale-free graphs, the degree distribution follows a power law, resulting in a large number of vertices in the largest component [59]. Based on this observation, Afforest proposes to use subgraph sampling (we use a similar technique in approximate seeding strategy for community detection discussed in Section 4.4) to identify the largest component. In the later stages of hooking, Afforest skips the rest of the unprocessed edges in the largest component. This would either defer edge processing or skip it altogether. It first uses neighbor sampling method to generate a subgraph. Within each neighborhood, a fixed amount of neighbors are randomly selected (for example, two neighbors are selected in Afforest). In this way, a subgraph consisting of O(n) random edges is created. This subgraph is a forest consisting of spanning trees of each connected component. Next, within the subgraph, a fixed number of component labels is selected (for example, 1024 labels are randomly selected in Afforest) among the intermediate component labels. The most frequent label is the component ID of the largest component in the graph. Assume that the largest component contains m_L edges and the subgraph sampling phase processes $m_{process}$ edges. It has been shown that by skipping the largest component, the connected component algorithm only processes $m_{process} + m - m_L$ edges [12]. In a scale-free graph, m_L contains substantial amount of edges.

Algorithm 12 Direction Optimizing Label Propagation Algorithm with Subgraph Sampling, Compress, and Path Compression.
Input: Graph $G = (V, E)$, an integer *neighbor_rounds*.
Output: $\forall v \in V$: $label[v]$ stores the label of v

```
1:  for all v ∈ V do in parallel
2:      label[v] ← a unique value in {0, 1, ··· , |V| − 1}
3:  GenerateSubgraph(label, neighbor_rounds)          ▷ either via push or pull
4:  l ← SampleFrequentElement(label)
5:  change ← True
6:  while change do
7:      for all v ∈ V do in parallel
8:          if l ≠ label[v] then
9:              change ← pullAndCompress(v)           ▷ can be replaced with pushAndCompress
10:     for all v ∈ V do in parallel
11:         compress(v, label)
```

We adopt the subgraph sampling heuristic to find the largest component ID as shown in Algorithm 11. There are two steps in the subgraph sampling: generate subgraph (via PUSH or PULL) and sample the most frequent label (the largest component ID). Since there are two ways to propagate the labels in DOLPA, we propose two ways to generate the subgraph in Algorithm 11, one using PULL and the other using PUSH. The input parameter *neighbor_rounds* controls how many number of neighbors are randomly selected in each sampling step. In practice, we do not randomly select *neighbor_rounds* neighbors, but select the first *neighbor_rounds* neighbors of a vertex in the subgraph generation.

5.6 DOLPA with Subgraph Sampling, Path Compression, and Compress

With the above optimizations, we present our DOLPA set of algorithms for connected component decomposition as shown in Algorithm 12. Our algorithms adopt a combination of PUSH and PULL (with or without path compression) to propagate labels, incorporate compress operation to speedup the convergence rate (especially for large diameter graphs), and include subgraph sampling operation to skip the largest component for scale-free graphs.

6 EXPERIMENTAL EVALUATION: CONNECTED COMPONENT DECOMPOSITION PERFORMANCE

In this first experimental section, we evaluate the speed of convergence as well as the runtime performance obtained by our connected components algorithms in comparison with other parallel connected component algorithms listed in Table 2. We use the synthetic graphs and real-world graphs listed in Table 3 for the evaluation. We also implement the frontier-based LPCC using PUSH and HOOK to evaluate the cost of creating an active frontier in each iteration.

6.1 Experimental Setup

For connected component decomposition experimental evaluation, our experiments are run on a machine with a two-socket Intel Xeon Gold 6230 processor, having 20 physical cores per socket, each running at 2.1 GHz, and 28 MB L3 cache. The system has 188 GB of main memory. Our code is compiled with GCC 10.2 compiler and `-Ofast -march=native` compilation flags. We used OpenMP 4.0 for parallelization with *guided* scheduling.

Table 2. Connected component decomposition algorithms studied. Notations of different algorithms are generated with the combinations of different setups and techniques applied.

Notation	Description
PULL	PULL (Algorithm 6)
PUSH	PUSH (Algorithm 5)
HOOK	HOOK (Algorithm 8)
PUSHF	PUSH, and Frontier-based
HOOKF	HOOK, and Frontier-based
PSPLCC	GenerateSubgraphviaPUSH, PULL, Path Compression, and Compress (Algorithm 12)
PLPSCC	GenerateSubgraphviaPULL, PUSH, Path Compression, and Compress (Algorithm 12)
PSPSCC	GenerateSubgraphviaPUSH, PUSH, Path Compression, and Compress (Algorithm 12)
PLPLCC	GenerateSubgraphviaPULL, PULL, Path Compression, and Compress (Algorithm 12)
SV	SV algorithm (our rewriting) [50]
Afforest	SV algorithm with subgraph sampling [59]

Table 3. Synthetic and real-world graphs for performance evaluation.

Input	Description	$\|V\|$	$\|E\|$	Ref.
road_usa	US road network	1M	9.5M	[4]
europe_osm	European Open Street Map road networks	51M	108M	[4]
rgg_n_2_24_s0	Random geometric graphs	17M	265M	[11]
kron_g500-logn20	Synthetic graphs from the Graph500 benchmark	1M	89M	[11]
uk-2005	2005 crawl of the .uk domain performed by Ubi-Crawler	39M	936M	[8]
LiveJournal	LiveJournal social network	4M	69M	[66]
Orkut	Orkut social network	3M	234M	[66]

6.2 Dataset

We conducted experiments with real-world from various domains and synthetic graphs, listed in Table 3. We selected two road networks from USA and Europe from the 10th DIMACS Implementation Challenge [4], two synthetic random graphs [11], one web graphs [8, 11], and two social network datasets obtained from Stanford Large Network Dataset Collection (SNAP) [33]. The road networks are large diameter graphs. The social networks are scale-free graphs.

6.3 Speed of Convergence and Runtime Performance

In this subsection, we present the runtime results and the number of iterations each algorithm in Table 2 takes to converge on the datasets in Table 3. We omit the first iteration at the beginning of the algorithm when each vertex initializes its label to be its vertex ID. We set the *neighbor_rounds* parameter to two when generating subgraphs, i.e., we propagate two labels in each vertex's neighborhood. All the results are obtained as the average of the runtime divided by the number of iterations of five runs using 40 threads.

In Table 4, we can see that our algorithm (PSPSCC) is 3.7 to 2303.2 times faster than the basic LP-based algorithm (HOOK), especially on large diameter datasets (road networks). DOLPA is 4.8 to 13.2 times faster than the SV algorithm, and 1.1 to 1.6 times faster than Afforest.

Table 4. The runtime and the number of iterations obtained on the connected component algorithms in Table 2 using the datasets from various domains in Table 3. The best runtime results are in bold. As we can see, the subgraph sampling does not reduce the number of iterations to reach convergence. Path compression dramatically reduces the number of iterations. The compress operation at the end of each iteration also reduces the number of iterations greatly.

Algo.	Road Network				Random		Web		Social Network			
	road_usa		europe_osm		rgg_n_2_24_s0		uk-2005		LiveJournal		Orkut	
	Time/s	#It.	Time/s	#It.	Time/s	#It.	Time/s	#It.	Time/s	#It.	Time/s	#It.
PULL	314.947	3071	353.232	4031	28.596	287	19.230	79	0.432	7	0.244	3
PUSH	346.139	3071	335.507	4034	33.360	288	40.681	247	0.489	7	0.361	3
HOOK	198.277	1613	313.284	3860	19.820	74	6.866	11	0.325	5	0.180	2
PUSHF	128.754	6259	550.423	17323	20.562	1664	6.436	199	0.136	8	0.170	3
HOOKF	113.551	6250	552.083	17309	24.507	1878	0.928	15	0.126	6	0.163	2
PSPLCC	0.109	3	0.177	3	0.069	4	0.597	35	0.020	4	0.017	4
PLPSCC	0.103	3	0.173	3	0.068	3	**0.339**	3	0.087	3	0.125	3
PSPSCC	**0.086**	3	**0.154**	3	**0.064**	3	0.518	3	**0.017**	3	**0.014**	3
PLPLCC	0.130	3	0.169	2	0.065	1	0.844	34	0.088	1	0.117	1
SV	0.768	9	1.409	7	0.849	6	1.626	7	0.154	3	0.200	2
Afforest	0.097	3	0.196	3	0.069	3	0.381	3	0.027	3	0.019	3

i) Subgraph Sampling. Subgraph sampling works well when combining with pushAndCompress, pullAndCompress, and hookAndCompress. If we compare PSPSCC with push (or PLPLCC with pull) in Table 4, we can see that the number of iterations as well as the runtime of PSPSCC is greatly reduced. With path compression, in GenerateSubgraphViapushAndCompress, each vertex propagates a label which now travels O($logn$) hops. The newly constructed subgraph consists of neighbor_rounds×$logn$ edges, as well as a large intermediate component and a number of small components. Many vertices within each component hold the minimum component label. When sampling the most frequent label within this subgraph, the task is most likely going to detect the largest component of this graph.

ii) Path Compression and the compress Operation. Path compression dramatically reduces the number of iterations, especially on large diameter graphs. In Table 4, comparing PSPSCC with push, which does not apply path compression or the compress operation, PSPSCC has approximately 0.065% of the number of iterations of push on road_usa graph. The path compression makes LP-based algorithm propagate labels more than one hop per iteration. In fact, with full path compression, each label is propagated O($logn$) hops per iteration, which is the depth of the labeling tree within each component.

iii) Direction Optimization. To evaluate the effect of direction optimization (switching between push and pull operations), we compare PSPLCC, PLPSCC, PSPSCC, and PLPLCC. The method PSPSCC performs best overall, while PLPSCC gives the best performance on Web. In general, we find direction optimization does not give connected components algorithms substantial performance benefits. This is because we are propagating labels conditionally, i.e., we only propagate the (minimum) label if the label is smaller than others. This means that if there exists only one label that is smaller than v's label in v's neighborhood, both push and pull would visit all neighbors but propagate one label per operation. Switching between push and pull does not contribute to less number of processed edges or more efficient label propagation. However, push is more efficient in propagating labels (with less amortized cost per label propagation) compared with pull; that's why PSPSCC has the best overall performance.

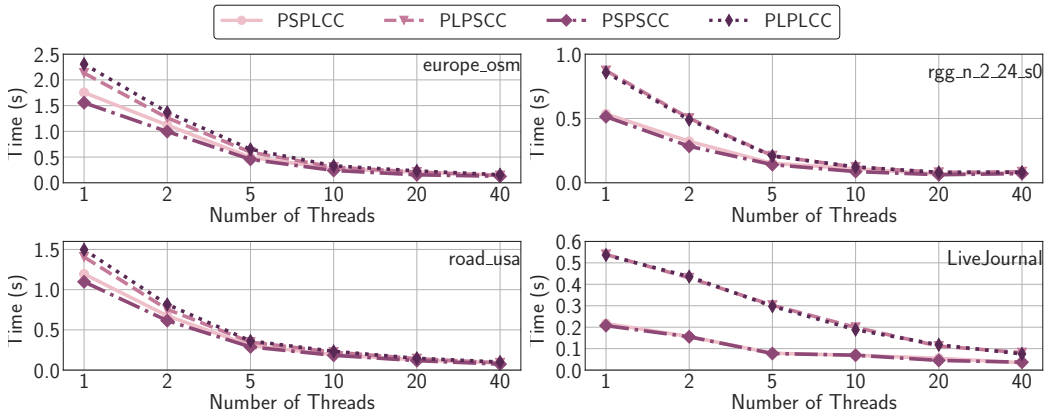


Fig. 3. Strong scaling results on up to 40 threads on four graphs for the connected components algorithm variants listed in Table 2. The variant PSPSCC has the best performance on four graphs. PSPLCC has close to the best performance.

iv) Frontier. The frontier-based (PUSHF vs PUSH, HOOKF vs HOOK) algorithms are noticeably faster than the non-frontier-based ones (except for europe_osm), even though the number of iterations increases. However, after we apply the frontier to DOLPA (not reported in Table 4), we find the frontier-based ones are always slower than their non-frontier-based variants. This is because path compression and the compress operation substantially reduce the number of iterations to converge. The overhead of creating a shared frontier and making sure there is no duplicate entry in it overcompensates its benefit, which only processes the vertices with recently updated labels.

6.4 Strong Scaling

Figure 3 shows the runtime scaling results of four parallel implementations of connected component algorithms on the real-world networks listed in Table 3. In the plot, PSPSCC has the best performance on four input graphs, followed by PSPLCC. PUSH-based has better performance than PULL-based on social networks. Our algorithms make the basic implementation more versatile and more adaptable to different types of graphs, as our implementations show no performance degradation on type of graphs ranging from road to random to social graphs.

7 EXPERIMENTAL SETUP: COMMUNITY DETECTION

In this and the subsequent two sections, we present various experimental evaluation results around community detection. This section presents the experimental setup and datasets. Section 8 focuses on evaluation of the proposed seeding strategies. Section 9 deals with evaluating performance and scalability of DOLPA in comparison with other community detection methods.

We use the Lancichinetti-Fortunato-Radicchi (LFR) benchmark [30] with ground truth communities to study the behavior of DOLPA and real-world graphs to evaluate the quality of solution and performance. The LFR benchmark is the most commonly used graph generator to evaluate community detection algorithms. We compare our implementation with PLP, and with the state-of-the-art parallel Louvain method implementation in Grappolo [40]. We begin the present section by detailing the experimental setup and the dataset used in this paper. We quantify the quality of solutions (of community detection) using Precision, Recall and F-Score (or F-Measure). We omit definitions for these metrics; for details on definitions, please refer to [39].

Table 5. Synthetic and real-world graphs for performance and quality of solution evaluation. The number of vertices ($|V|$) and edges ($|E|$) along with the maximum degree (Δ) for the inputs are tabulated here.

Input	Description	$\|V\|$	$\|E\|$	Δ_v	Ground truth	Ref.
B0	Generate using the LFR benchmark with $\mu = 0$	1M	9.5M	100	Yes	[30]
B1	Generate using the LFR benchmark with $\mu = 0.1$	1M	9.5M	100	Yes	[30]
B3	Generate using the LFR benchmark with $\mu = 0.3$	1M	9.5M	100	Yes	[30]
B5	Generate using the LFR benchmark with $\mu = 0.5$	1M	9.5M	100	Yes	[30]
B7	Generate using the LFR benchmark with $\mu = 0.7$	1M	9.5M	100	Yes	[30]
B9	Generate using the LFR benchmark with $\mu = 0.9$	1M	9.5M	100	Yes	[30]
S1	Generate using the LFR benchmark with $\mu = 0.1$	1M	9.5M	100	Yes	[30]
S3	Generate using the LFR benchmark with $\mu = 0.3$	1M	9.5M	100	Yes	[30]
LL	Low Block Overlap and Low Block Size Variation	1M	24M	122	Yes	[23]
LH	Low Block Overlap and high Block Size Variation	1M	24M	137	Yes	[23]
HL	High Block Overlap and Low Block Size Variation	1M	24M	104	Yes	[23]
HH	High Block Overlap and High Block Size Variation	1M	24M	180	Yes	[23]
fbnt	Facebook network	4M	24M	4,915	No	[47]
dblp	Coauthor-ship from DBLP	0.5M	15M	3,299	No	[4]
zbrp	Zhishi Baidu related pages	416K	2.4M	127,090	No	[27]
cond	Condensed matter collaborations	40K	176K	278	No	[4]

7.1 Experiment Platform

We used a system having one node with two Intel Xeon E5-2699v3 processors operating at 2.3 GHz. The system has 18 physical cores per socket, 72 logical cores per node with hyper-threading, 48 MB L3 cache per processor, and 128 GB total main memory. We used GCC 9.3.0 compiler with -O3 compilation option to build the codes. We used OpenMP 4.0 for parallelization with *guided* scheduling.

7.2 Datasets

We generated two groups of the LFR benchmarks—**B**ig and **S**mall: the community size in the Big group ranges from 20 to 200 and the community size in the Small group ranges from 10 to 100. The fraction of overlapping vertices is ten, and the number of memberships of the overlapping vertices is two. We generated six benchmarks in the Big group with mixing parameter values of $\mu = \{0, 0.1, 0.3, 0.5, 0.7, 0.9\}$ and two benchmarks in the Small group with $\mu = \{0.1, 0.3\}$. The mixing

parameter μ of the LFR benchmark indicates the amount of noise in the network, as it controls the fraction of edges that are between communities. The higher the mixing parameter, the more difficult it is for the algorithm to detect communities.

Each benchmark was generated with 1 million vertices, 9.5 million edges, average degree 20, and max degree 100. Both of the Big and Small groups have the same parameters except the mixing parameter. The synthetic graphs generated using the LFR benchmark are listed in the top portion in Table 5. The generating scripts for LFR benchmarks, including the parameters to generate the benchmarks, are provided in the Gitlab repo[1].

The middle portion of Table 5 lists synthetic graphs we downloaded as images from the 2019 Stochastic Block Partitioning Graph Challenge [23]. The bottom portion of the table lists the real-world graphs in our testbed. Since these graphs do not have ground truth communities, we obtained the community structure data from fast-tracking resistance (FTR) algorithm as ground-truth data. The FTR algorithm is a hierarchical multiresolution method to overcome the resolution limit of a community detection algorithm in a complex network [17]. It is not true ground-truth, but we use it as a simple reference.

7.3 Parameter Choice for PUSH–PULL Switch

In results reported in detail in previous work [39], we designed a microbenchmark to study the behaviors of PUSH and PULL. Based on the results of our microbenchmark, we find the best switch threshold ω for DOLPA. To achieve the best runtime with reasonable quality of solution, we set $\omega = 2$. To obtain the best quality of solution in reasonable runtime, we set $\omega = 1$. Note that when $\omega = 1$, DOLPA does PULL only for label propagation.

8 EXPERIMENTAL EVALUATION: SEEDING STRATEGIES AND PARAMETERS

We present in this section results of experiments conducted to understand the behavior and performance of the proposed seeding strategies in conjunction with the choice of the seeding parameter τ.

8.1 Experimental Design

For each of the nine seeding strategies, we run DOLPA on the LFR benchmark Big (B0 - B9) group and collect the average of their runtime and F-Score. While we report results from only one of three groups of the benchmarks, we note that similar results were observed in the other two groups.

To determine the seeding parameter τ that produces best performance for each seeding strategy, we do a grid search on τ through a manually specified subset of τ where the lower bound is selecting one vertex as seed and the upper bound is selecting the whole V as seeds. We provide the subset of τ as follows.

Let the density of the graph be defined as $\delta_G = \frac{2|E|}{|V|(|V|-1)}$ and the average degree of the graph be defined as $\overline{\mathrm{d}}_G = \frac{2|E|}{|V|}$. We run DOLPA with $\tau = \{1e-6, 0.025, 0.05, 0.1, 0.2, 0.4, 0.6, 0.8, 1, \frac{1}{\overline{\mathrm{d}}_G}, \frac{2}{\overline{\mathrm{d}}_G}, \delta_G\}$. We do so 5 times for each input. Considering that selecting one vertex as seed out of 1 million vertices and running this for five times may not fairly reveal the actual results, we run DOLPA 20 times when $\tau = 1e-6$. Notice that for the benchmarks we generated (B0-B9), $\frac{1}{\overline{\mathrm{d}}_G} \approx 0.05$, $\frac{2}{\overline{\mathrm{d}}_G} \approx 0.1$.

8.2 Runtime and Quality of Solution Results

We collect two metrics of performance: runtime and F-Score. Figure 4 shows runtime (in log scale) versus mixing parameter results and Figure 6 shows F-Score versus mixing parameter results for

[1] https://datascience.aeolus.wsu.edu/tlieu/dolpa

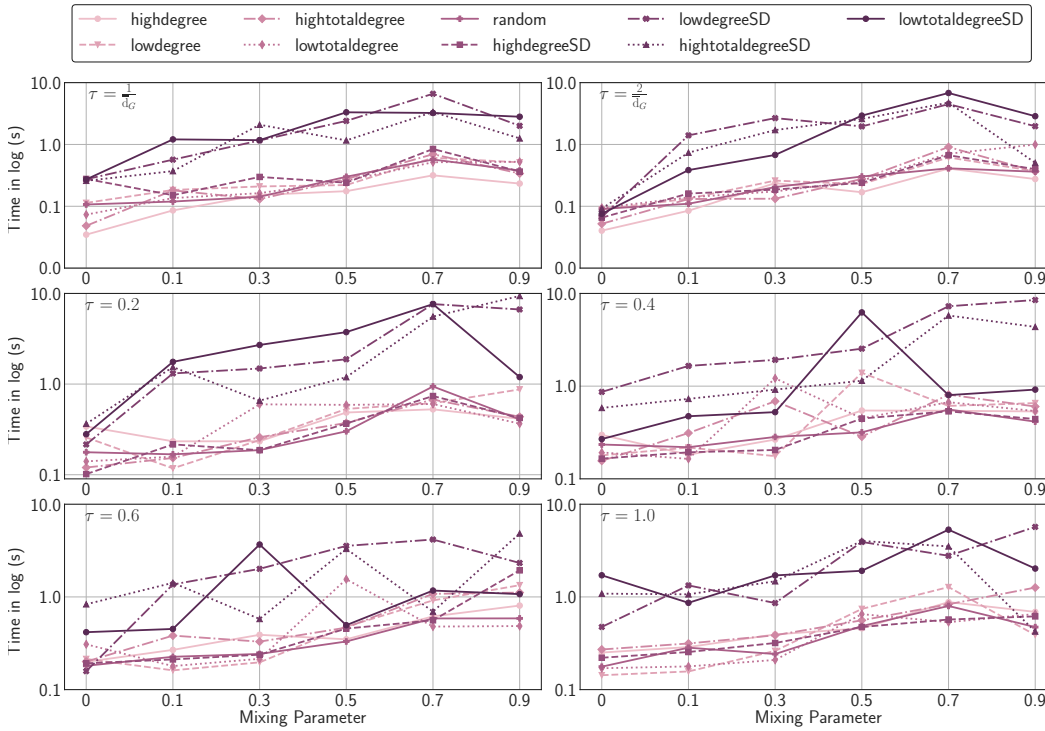


Fig. 4. Runtime results (in log scale) versus mixing parameter for the nine seeding strategies under various seeding parameter τ and various LFR benchmarks. For τ between 0.025 and 0.1, we observe that the High-degree seeding strategy achieves the best runtime compared to all other seeding strategies. For τ greater than 0.2, there is no clear winner in terms of runtime among all nine seeding strategies.

the nine seeding strategies under various values of τ. We also plot runtime versus the seeding parameter τ of the nine seeding strategies under various LFR benchmarks; these are shown in Figure 5. Each subplot in Figure 5 has the same mixing parameter. In the experiments that led to results in Figure 6, similar patterns were observed for $\tau = 0.2, 0.4, 0.6, 0.8, 1.0$. Hence we plot only for $\tau = 0.2$. We do not report the F-Score results for $\tau = 1e - 6$ and $\tau = \delta_G$ because the F-Score results are too low to be meaningful.

The approximate seeding strategies (except for highdegreeSD) have much slower runtime results than those of the random and the exact seeding strategies as can be seen in Figure 4. Figure 5 shows that the normalized runtime results of all the strategies when $\tau \geq 0.6$ are much larger than the corresponding results when $\tau \leq 0.6$. It can also be seen that the runtime of every strategy gets larger when the benchmark graphs become harder for an algorithm to tackle (i.e. the mixing parameter increases).

In Figure 6, the F-Score results of Random, Low-degree and Low-total-degree seeding strategies deteriorate slower than others when τ is between 0.025 and 0.1 and the mixing parameter of the LFR benchmark varies from 0 to 0.7. The F-Score results are similar when $\tau \geq 0.2$ for all strategies. Low F-Score results are observed in Figure 6 on the approximate strategies when the mixing parameter of the LFR benchmark is 0.

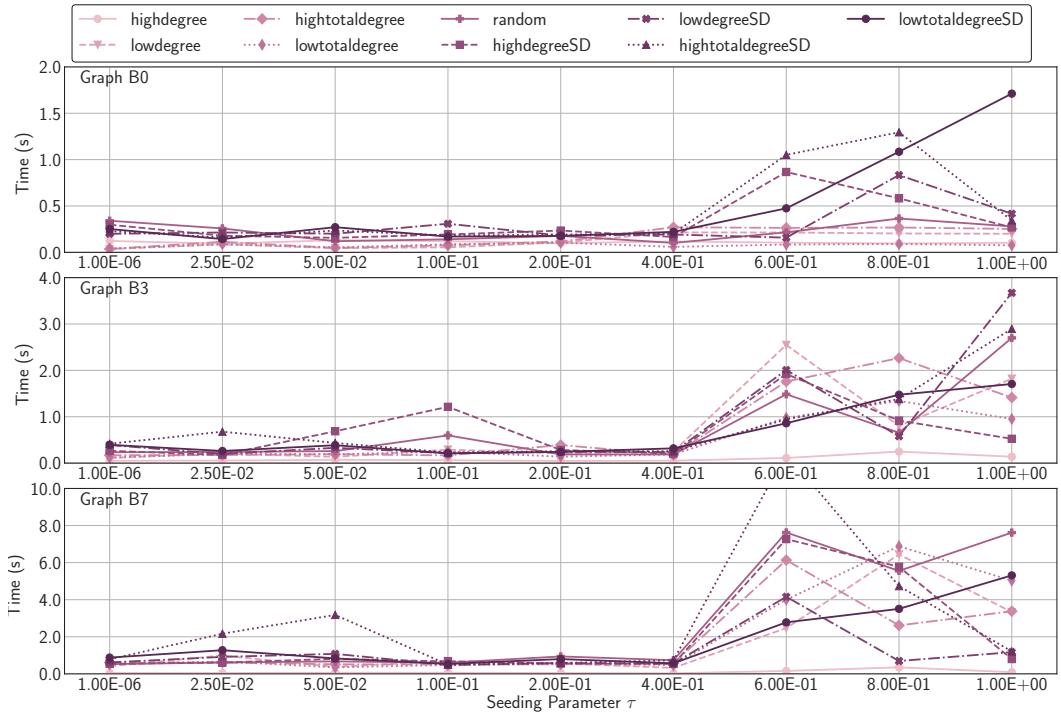


Fig. 5. Runtime versus seeding parameter τ results for the nine seeding strategies under various LFR benchmarks generated under different mixing parameters. The mixing parameters μ from the top subfigure to the bottom are $0, 0.3, 0.7$. A clear pattern can be observed that when $\tau > 0.4$, every seeding strategy has a dramatically increased runtime within each subfigure. From the top subfigure to bottom the y-axis of each subfigure is also increasing. While we do not report the F-Score when $\tau = 1e-6$ and $\tau = 0.025$ in Figure 6, we report their runtime results as references.

8.3 Analysis

Below we make several important observations from an analysis of the results in Figure 4, Figure 5 and Figure 6.

i) A tiny τ does not produce reasonable solution. The F-Score results for $\tau = 1e-6$ and $\tau = \delta_G$ are too low to be reported in Figure 6 even though both cases converge fairly fast in Figure 5. This shows that a tiny τ (i.e., only one seed or a few seeds are selected) is not practical for DOLPA to achieve reasonable quality of solution. It is impractical to assume that there are only one or a few community structures in the graphs.

ii) A large τ dramatically increases runtime. Figure 5 shows that the runtime of every seeding strategy increases dramatically when $\tau \geq 0.6$ (we called this a large τ). The approximate seeding strategies are particularly highly impacted by this, especially when the mixing parameter is 0. This indicates that τ should not be made greater than 0.6 to achieve reasonable runtime. In particular, we conclude that τ should always be smaller than 0.5. Let us elaborate on this. When $\tau > 0.5$, more than 50% of the vertices are treated as seeds of a community structure. This means there exists an edge $e \in E$ where at least one end point of e has its label updated twice. The larger τ is, the more labels are updated redundantly. This results in slow convergence.

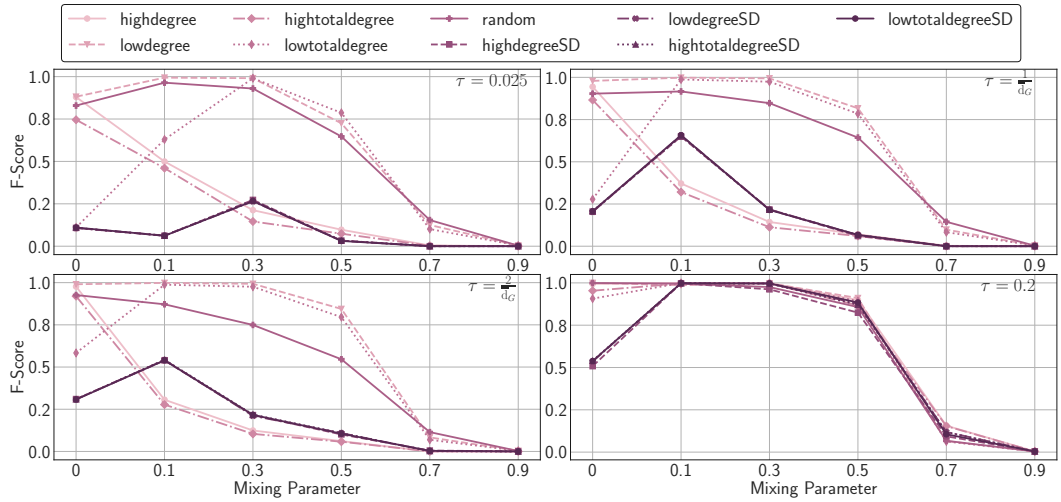


Fig. 6. F-Score versus mixing parameter plots for the nine seeding strategies under various seeding parameter τ and various LFR benchmarks. Similar patterns are observed for $\tau \geq 0.2$, hence we only plot results for $\tau = 0.2$. Moreover, we do not report the F-Score results for $\tau = 1e-6$ and $\tau = \delta_G$ because their results are too low.

iii) The approximate seeding strategies perform poorly when sample size is small. When the sample size is small, there is a higher probability that the sampled vertex set or the sampled neighborhood set has a bias over the original graph, either in terms of the vertex degree or the total degree of a neighborhood. Hence, a small τ for an approximate seeding strategy often fails to represent the parent graph faithfully. When the sample size is large enough, the sampled vertex set or the sampled neighborhood set can properly approximate the parent graph. This is why when $\tau \geq 0.2$, the approximate seeding strategies become as robust as the exact seeding strategies and achieve similar F-Scores.

The runtime results of the approximate seeding strategies in Figure 4 are (surprisingly) higher than those of the exact seeding strategies as τ increases. As τ increases, the advantage of saving time on sorting is overcompensated by the increased time for sampling. In addition, the approximate strategies involve random selection of vertices/neighborhoods during sampling, which in turn results in slower convergence.

iv) Random, Low-degree, and Low-total-degree seeding strategies are robust. First, we observe that Random, Low-degree and Low-total-degree seeding strategies are not sensitive to the value of τ. They have stable F-Score results no matter the value of τ (excluding a tiny $\tau = 1e-6$ or δ_G), as can be seen in Figure 6. Second, we notice that these strategies are also not sensitive to noise (for benchmark graphs whose mixing parameter $\mu > 0$). Their F-Score results do not deteriorate dramatically as the mixing parameter increases as can be seen in Figure 6. Among these three strategies, Low-degree and Low-total-degree (the top two lines in Figure 6) have better performance than Random.

However, the reasons behind the robustness of the three seeding strategies are different. Random seeding strategy is robust because it has no bias over the vertices in the graph. Each vertex is selected independently with equal probability. Low-degree and Low-total-degree seeding strategies are robust because the labels of a low-degree vertex or a low-total-degree neighborhood has a higher probability to survive the early propagation stages. A low-degree vertex or a low-total-degree neighborhood has a small number of neighbors, making their labels having less competitors. This

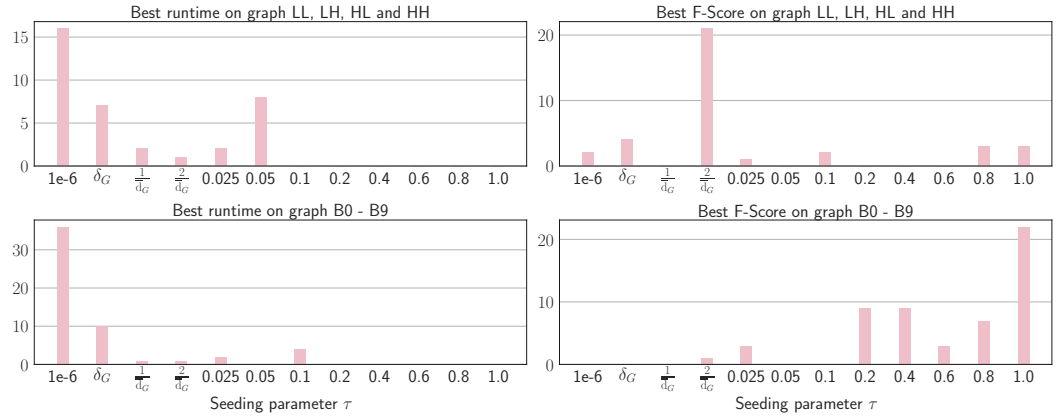


Fig. 7. We count the number of cases a specific τ value produces best runtime results (left pane) and best F-Score results (right pane) as τ is varied. Results for the graphs LL, LH, HL and HH (top row) and the graph B0-B9 (bottom row) are plotted separately. The setting $\tau = 1e-6$ produced 16 best runtime results out of 36 cases on graphchallenge data, and 36 best runtime results out of 54 cases on the LFR graphs. The setting $\tau = \frac{2}{\bar{d}_G}$ produced 21 (out of 36) best F-Scores for the graphs LL, LH, HL and HH, while the setting $\tau = 1.0$ produced 22 (out of 54) best F-Score results for the LFR graphs.

makes their labels vigorous and thus makes a low-degree vertex or a low-total-degree neighborhood an effective seed.

v) Noise makes DOLPA converge slower. No matter which seeding strategy is used in Figure 5, DOLPA converges slower (the y-axis scale of each subplot increases) as the mixing parameter μ of the LFR benchmark graph increases. It requires more time for DOLPA to get rid of noises when the noises increase. We observe similar linear correlation between the runtime of DOLPA and the mixing parameter in Figure 4 as was observed in Lancichinetti et al. [30].

8.4 What is a Good Seeding Parameter τ Value?

To find out the best τ setting for best performance (both runtime and quality of solution), we summarize the number of τ settings that produce the best runtime and best F-Score results on the graphs LL, LH, HL, HH and the graphs B0-B9 separately in Figure 7. The figure shows that $\tau = 1e-6$ is clearly the winner for producing best runtime. However, the F-Scores of $\tau = 1e-6$ are too low to be useful. We therefore recommend a larger τ for a reasonable combination of runtime and F-Score.

In Figure 7, we notice that the best F-Score results of the LFR graphs and the graphchallenge graphs are achieved at two extremely different τ values. Most of the best F-Scores of the graphchallenge graphs cluster around $\tau = \frac{2}{\bar{d}_G}$, which is a small τ. In contrast, most of the best F-Scores of the LFR graphs are achieved at $\tau = 1.0$, which is a big τ. However, the runtime results of $\tau = 1.0$ is extremely high due to redundant label updates.

To conclude, a small τ such as $\tau = \frac{2}{\bar{d}_G}$ gives the best runtime at a reasonable (even best F-Score results) quality of solution; a large τ such as $\tau = 1.0$ can provide best quality of solutions at the cost of slow convergence. With this conclusion, we can adjust τ to achieve a balance between runtime and quality of solution.

Table 6. Community detection algorithms studied.

Version	Description
PLP	Parallel Label Propagation algorithm [56]
PUR	DOLPA using PULL only and with Random seeding
PUH	DOLPA using PULL only and with High-degree seeding
PUL	DOLPA using PULL only and with Low-degree seeding
DOR	DOLPA using PUSH & PULL and with Random seeding
DOH	DOLPA using PUSH & PULL and with High-degree seeding
DOL	DOLPA using PUSH & PULL and with Low-degree seeding
LV	Parallel Louvain method implementation [40]

9 EXPERIMENTAL EVALUATION: COMMUNITY DETECTION RUNTIME AND SOLUTION QUALITY

In this final experimental section, we evaluate the quality of solution and runtime obtained by DOLPA in comparison with other community detection methods using the synthetic graphs and real-world graphs listed in Table 5.

Table 6 lists the variants of parallel LPA we study (PLP, PU and DO) and the Louvain method (LV). The methods PUR and DOR employ the Random seeding strategy. Methods PUH and DOH employ the High-degree seeding strategy, and methods PUL and DOL use the Low-degree seeding strategy. In both PU and DO, we set $\tau = \frac{2}{\mathrm{d}_G}$ to achieve good runtime results with reasonable quality of solution. We use the switch threshold ω of 1 for PU and that of 2 for DO. Recall that when ω is 1, no PUSH operation is applied.

Each experiment is run 10 times and the average of the results is reported. We omit the scaling and runtime results in this section due to space limit. For details of them, please refer to [39].

9.1 Runtime and Quality of Solution Results

Table 7 shows the F-Score and runtime results on the eight methods listed in Table 6. The results show that DOLPA outperforms PLP as well as the Louvain method in most of the runtime and F-Score results. In particular, DOLPA achieves eight best runtime results out of eleven cases, and nine best F-Score results out of the eleven cases. DOLPA adopting a High-degree seeding strategy (DOH and PUH) achieves five best runtime results among eight cases. DOLPA adopting Random seeding strategy (PUR) achieves three best F-Score results among nine best F-Scores, while DOLPA adopting Low-degree seeding strategy (DOL and PUL) achieves six best F-Scores among nine best F-Scores. Compared with PLP, DOLPA achieves at least two times the F-Score while maintaining similar runtime for the LFR graphs; DOLPA adopting random seeding strategy has up to 48x the F-Score on the graph HL. Compared with Louvain method using the same graphs, the best results achieved by DOLPA have an average of three times the F-Score at a tenth of the runtime. We provide further analysis on the results summarized in Table 7 in the remainder of this subsection.

i) Frontier. The use of frontier in DOLPA is beneficial. A frontier can process the vertices in any order desired. During the initialization of DOLPA, the seeds are added to the initial frontier so that the labels of seeds can be propagated first. This makes the labels of the seeds have a higher probability of forming a strong community core without being eliminated. As can be observed from the experiments, PLP and PU have similar numbers of label updates, propagated steps, and processed edges. The high Precision of PU shows that PU is more efficient and accurate in finding the "right" maximum label. In Table 7, the variants PUL and PUR respectively have 2.3 and 2.5 times the F-Score of PLP on the graphs B3 and S3. The method PUR obtains more than ten times

Table 7. Runtime and F-Score results of the eight methods listed in Table 6 on eight synthetic graphs with ground truth information and three real-world graphs. The real-world graphs use as "ground truth" the results obtained from the FTR method [17]. All results are obtained under 64 threads.

	Input	B1	S1	B3	S3	LL	LH	HL	HH	fnbt	dblp	cond
PLP	Time	**0.093s**	**0.110s**	0.108s	0.234s	0.573s	0.291s	**0.165s**	0.231s	2.668s	0.248s	0.044s
	Fscore	0.8867	0.4264	0.824	0.3871	0.1432	0.166	0.0201	0.0559	0.1038	0.0751	0.034
	Prec.	0.7965	0.271	0.7007	0.2401	0.0772	0.0905	0.0102	0.0288	0.0565	0.0392	0.0196
	Recall	1	0.9999	1	0.9999	1	0.9994	0.9999	1	0.9277	0.9064	0.431
PUH	Time	0.121s	0.158s	0.116s	0.166s	**0.246s**	0.248s	0.337s	0.232s	1.857s	0.217s	0.037s
	Fscore	0.3103	0.1285	0.285	0.0817	0.0255	0.0348	0.0414	0.0269	0.0686	0.0034	0.0117
	Prec.	0.1837	0.0687	0.1662	0.0426	0.0129	0.0177	0.0211	0.0136	0.0355	0.0017	0.0059
	Recall	0.9994	0.9999	0.9997	0.9999	1	1	0.9999	1	0.9971	0.9008	0.4348
DOH	Time	0.094s	0.256s	0.113s	0.164s	0.316s	**0.200s**	0.231s	**0.173s**	**0.998s**	0.299s	**0.024s**
	Fscore	0.161	0.06	0.1304	0.0293	0.0877	0.05	0.0241	0.0348	0.0673	0.0029	0.0068
	Prec.	0.0876	0.0309	0.0697	0.0149	0.0462	0.0258	0.0122	0.0177	0.0348	0.0014	0.0034
	Recall	0.9999	0.9998	1	0.9998	1	1	0.9999	0.9999	0.9956	0.9052	0.458
PUR	Time	0.158s	0.179s	0.138s	0.172s	0.370s	0.479s	0.504s	0.417s	2.619s	**0.193s**	0.027s
	Fscore	0.8813	0.7465	0.8621	0.7101	**0.997**	0.8277	**0.8095**	0.127	0.1574	**0.3531**	0.0436
	Prec.	0.8028	0.5965	0.7873	0.5522	0.9941	0.7072	0.6954	0.0681	0.0977	0.2212	0.0257
	Recall	0.9768	0.9983	0.9525	0.9945	1	0.9996	1	0.9999	0.8947	0.8749	0.4001
DOR	Time	0.106s	0.168s	**0.106s**	0.181s	0.279s	0.312s	0.319s	0.278s	2.186s	0.301s	0.029s
	Fscore	0.2651	0.096	0.2594	0.0518	0.7898	0.6069	0.0812	0.0613	0.0648	0.3505	0.0507
	Prec.	0.1528	0.0504	0.149	0.0266	0.6529	0.4437	0.0429	0.0317	0.0335	0.2193	0.0305
	Recall	1	0.9998	1	0.9999	1	0.9997	0.9999	0.9999	0.9872	0.8757	0.4125
PUL	Time	0.126s	0.214s	0.127s	0.166s	0.418s	1.066s	0.630s	0.416s	2.485s	0.463s	0.031s
	Fscore	**0.999**	**0.9946**	**0.9966**	**0.9909**	0.3979	0.2494	0.0713	0.029	0.0679	0.2072	**0.0558**
	Prec.	0.998	0.9894	0.9939	0.9821	0.271	0.1466	0.0381	0.0147	0.0352	0.1171	0.0348
	Recall	1	0.9998	0.9994	0.9999	1	0.9997	0.9969	1	0.9822	0.9285	0.4153
DOL	Time	0.106s	0.168s	0.178s	**0.162s**	0.348s	0.297s	0.326s	0.285s	2.118s	0.436s	0.026s
	Fscore	0.3223	0.1211	0.3815	0.0764	0.9198	**0.9003**	0.2135	0.0984	0.134	0.2155	0.0425
	Prec.	0.1922	0.0645	0.2358	0.0397	0.8519	0.8218	0.1318	0.0519	0.0775	0.1219	0.024
	Recall	0.9999	0.9999	1	0.9999	1	0.9997	0.9998	0.9996	0.8829	0.9281	0.4131
LV	Time	4.224s	4.135s	5.698s	5.038s	15.980s	21.904s	21.748s	17.098s	13.671s	1.002s	0.084s
	Fscore	0.1519	0.0772	0.0043	0.0014	0.3449	0.4205	0.1876	**0.1759**	**0.2547**	0.0105	0.0378
	Prec.	0.0822	0.0402	0.0022	0.0007	0.2084	0.2843	0.108	0.0999	0.1545	0.0053	0.0193
	Recall	0.9982	0.9984	0.9816	0.9865	0.9995	0.8071	0.7139	0.7331	0.7237	0.9776	0.935

The input graphs in this table are listed from easy to hard to detect for community detection algorithms. Among the eight methods, the best runtime and the best F-Score results are shown in **bold** text. The runtime includes pre-processing steps such as degree sorting, frontier insertion, etc in PU/DO/LV. There are no such steps in the PLP algorithm.

the F-Score of PLP on the graphs LL and HL. The method PUL has comparable runtime with that of PLP. Note that the runtime of PUL includes steps such as degree sorting and frontier insertion.

In addition, during the later iterations, frontier guarantees DO only processes nodes that were activated in the previous iteration. On the contrary, PLP also processes nodes that were activated in the current iteration if they are visited after being activated. Doing so is harmful to the quality of the solution, because the importance of the seeds' labels are overlooked. Further, PLP only deactivates nodes that were not moved while DO always deactivates a processed node.

ii) Seed Vertices. The "right" seeds improve accuracy. Seeds propagate before others in the first iteration. With a unique label initially, each label is a maximum label in the first iteration. When a seed applies PULL for the first time, it abandons its own label by randomly selecting a maximum label in its neighborhood. This promotes that label as the "true" maximum label because it now appears twice in the neighborhood. If the label of the seed chooses survives in the next few iterations, a core of the community is formed. With the "right" choice on seeds, the likelihood of the seeds to form a strong and stable core is higher than other vertices. This shows that updating more important vertices in the network earlier than others is effective [64].

Conversely, the "wrong" seeds decrease accuracy. The method DOH has the worst F-Score in most instances in Table 7. Pushing labels of high degree vertices to their neighbors contributes negatively to the quality of the solution even though this almost guarantees a good runtime. The method DOR has less chance to "poison" other communities' labels when DOLPA uses random seeds instead of high degree seeds. The method DOL behaves similarly. This is reflected in the much greater F-Score value of the methods DOR and DOL compared to the method DOH.

iii) Direction Optimization. Direction optimization provides a trade-off between time and accuracy by adjusting the switch threshold ω to obtain a balance of PUSH and PULL operations. When selecting the seeds in the same strategy, DO has shorter runtime than PU; DOH is faster than PUH; and DOR is faster than PUR. Compared with PLP, the method DOR has an average 50% of runtime decrease on the LFR benchmarks and DOH has an average 15% of runtime decrease on the graph-challenge graphs. With a better seed selection, DOR has an average of 14 times F-Scores compared to PLP on the graph-challenge datasets.

Hence, we can adjust ω for a good balance in different computation scenarios. The method DO fits best for a time-sensitive scenario. An appropriate switch threshold reduces work by applying certain amount of the PUSH operations on seeds, thus providing higher performance. However, PUSH is harmful to the quality of solution in general since it forces an undesirable label choice to all neighbors. The amortized cost for each label update of PUSH is constant, while the cost of PULL is $O(d(v))$. The method PU is well suited for a precision-demanding scenario, where the switch threshold can be set as small as one so that there is no PUSH operation in pursuit of higher accuracy.

10 RELATED WORK

Community detection. Since LPA was first introduced for community detection, many other works that improve or extend it have been proposed. We discuss only a few here. Xie et al. introduce a method to "stabilize" LPA by eliminating the need for tie breaking [63]. Other works alleviate the randomness of tie breaking with other node preference or edge preference method instead of treating each node/edge with the same preference. Preference methods studied include k-shell value [64], local cycle [70], or modularity [5, 37]. Leung et al. use hop weight to prevent the occurrence of a "monster" community [34]. Yet other common measures for node preference include degree centrality and clustering coefficient [54]. The works using node preferences are proven to produce a deterministic solution but the approaches have high computational cost and/or evaluation metric bias.

The label propagation algorithm can be made faster by maintaining all label information in memory instead of computing on the fly [13, 18]. But this approach requires data synchronization for the label information and it is not scalable. The state-of-the-art parallel LPA is the Parallel Label Propagation (PLP) algorithm implemented on multi-core architecture [56]. Other works parallelize LPA on multi-core [28], on GPU [26, 55], in Map-Reduce model [69] and in distributed memories [3]. Liu et al. [38] also combine LPA with direction optimization technique, but their work is implemented in distributed memory with active-message based runtime system.

The quality of solution produced by LPA can be improved by selecting influential nodes [72] or community kernels [35] in the pre-processing phase and then growing community structures from them. This method is similar to a main stream of method in overlapping community detection called *seed set expansion*. Some works select maximal cliques as seeds [45, 49], some other works select high degree vertices [42, 61] and yet others select random vertices [31]. Stoica et al. [58] did a comprehensive study on seed set selection in the context of social influence maximization.

Direction optimization has been applied in other graph algorithms, including PageRank[62], Betweenness Centrality[41], Connected Components [19] and Single Source Shortest Path [10] and in graph frameworks such as Polymer[68] and Ligra[51]. Besta et al. [7] study push-pull dichotomy in graph computations in terms of performance, speed of convergence and code complexity. Tithi et al. [60] propose push-pull based Louvain method that can prune a significant amount of edges to speedup performance.

Connected component decomposition. Shiloach and Vishkin [50] introduced the first parallel connected components algorithm on the Parallel Random Access Machine (PRAM) model. The algorithm relies on the disjoint-set data structure based on the union-find algorithm. Afforest [59] is a variant of this approach but focusing on small-world graphs. It identifies the dominant component ID via subgraph sampling, then skipping those vertices residing in the component with dominant component ID. LACC [1, 71] is the latest distributed implementation of fast SV using linear algebra operations in GraphBLAS.

Minimum label propagation approach is another parallel method [44, 65]. Each vertex is initialized to its vertex ID and holds as a label of its component ID. Repeatedly, each vertex update its label to the smallest label among its neighbors until no further update is possible. In the end, the smallest label in each component serves as the unique component ID.

Given a graph G, let D denote the diameter, i.e. the maximum distance in G, where distance is the length of the shortest path between two vertices. Paul Burkhardt [9] introduces a method that works in log-D steps and $O((m+n)logD)$ work with $O(m+n)$ processors using label propagation in the PRAM model that does not require pointer operation. Andoni et al. [2] proposed a $O(logDloglog_{m/n}n)$ time connectivity algorithm for diameter-D graphs using $\Theta(m)$ total memory in massive parallel computing (MPC) model. Liu and Tarjan [36] introduced a method that works in $O(logn)$ steps and sends $O(mlogn)$ total messages in the MPC model. Stergiou et. al [57] implement this approach with shortcutting in bulk synchronous parallel (BSP) model. Label propagation based methods are implemented in graph processing frameworks such as Ligra[51], PEGASUS[22] and GraphChi[29].

Shun et. al [52] implement the breadth-first search approach for finding connected components in the PRAM model. Each unprocessed vertex is processed in parallel BFS and mark all the reached vertices as the same component. This approach runs in $O(m)$ time. ConnectIt [12] studies all the above connected component decomposition algorithms in depth, and summarizes several subgraph sampling strategies in their framework.

11 CONCLUSION

We presented a new Label Propagation algorithm, called Direction-optimizing Label Propagation Algorithm (DOLPA), for graph structure detection and showed its efficacy as a method for community detection and connected component decomposition in networks. We introduced a new label update heuristic called PUSH, and abstracted the currently known label update operation as PULL. The algorithm applies PUSH for label update in the early iterations and switches to PULL for label update in later iterations. We incorporated several heuristics for connected component decomposition, and combined them with PUSH and PULL. Using a carefully designed microbenchmark, we analyzed the

characteristics of PUSH and PULL. We proposed a total of nine seeding strategies and extensively studied their performance.

We validated our implementation on benchmarks with known ground truth and demonstrated increased accuracy and decreased runtime compared to the state-of-the-art parallel implementations. The time-to-solution/quality-of-solution trade-off that our algorithm provides (a combination of seeding parameter τ and switching threshold ω) enables effectively addressing many community detection scenarios. For fast community detection, PUSH saves time but too many PUSH operations can harm the precision; also, a small τ guarantees best runtime. For accurate community detection, PULL is precise but can be costly; a large τ will most likely produce reasonable quality of solution.

We investigated our algorithm for finding connected components on datasets from various domains and demonstrated orders of magnitudes speedup over the basic LP-based algorithm, up to 13.2 speedup over the SV algorithm, and competitive performance as the Afforest algorithm.

ACKNOWLEDGEMENTS

This research was supported by NSF awards CAREER IIS 1553528 and SI2-SSE 1716828, by the U.S. DOE Exascale Computing Project's (ECP) (17-SC-20-SC) ExaGraph codesign center, and by the Segmented Global Address Space (SGAS) LDRD under the Data Model Convergence (DMC) initiative at the U.S. Department of Energy's Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. This research used computing resources from the Aeolus High Performance Computing cluster at Washington State University and from the Hyak computing cluster at the University of Washington.

REFERENCES

[1] A. Azad and A. Buluç. 2019. LACC: A Linear-Algebraic Algorithm for Finding Connected Components in Distributed Memory. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS.2019.00012

[2] A. Andoni, Z. Song, C. Stein, Z. Wang, and P. Zhong. 2018. Parallel Graph Connectivity in Log Diameter Rounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. 674–685. https://doi.org/10.1109/FOCS.2018.00070

[3] J. Attal, M. Malek, and M. Zolghadri. 2019. Parallel and distributed core label propagation with graph coloring. *Concurrency and Computation: Practice and Experience* 31, 2 (2019), e4355. https://doi.org/10.1002/cpe.4355

[4] D. A. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. 2017. *Benchmarking for Graph Clustering and Partitioning*. Springer New York, New York, NY, 1–11. https://doi.org/10.1007/978-1-4614-7163-9_23-1

[5] M. J. Barber and J. W. Clark. 2009. Detecting network communities by propagating labels under constraints. *Phys. Rev. E* 80, 2 (Sept. 2009), 026129. https://doi.org/10.1103/PhysRevE.80.026129

[6] S. Beamer, K. Asanović, and D. Patterson. 2013. Direction-Optimizing Breadth-First Search. *Scientific Programming* 21, 3-4 (2013), 137–148. https://doi.org/10.3233/SPR-130370

[7] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA) *(HPDC '17)*. ACM, 93–104. https://doi.org/10.1145/3078597.3078616

[8] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[9] P. Burkhardt. 2021. Graph Connectivity in Log Steps Using Label Propagation. *Parallel Processing Letters* 31, 04 (2021), 2150021. https://doi.org/10.1142/S0129626421500213 Publisher: World Scientific Publishing Co..

[10] V. T. Chakaravarthy, F. Checconi, P. Murali, F. Petrini, and Y. Sabharwal. 2017. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2017), 2031–2045. https://doi.org/10.1109/TPDS.2016.2634535

[11] T. A. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[12] L. Dhulipala, C. Hong, and J. Shun. 2020. ConnectIt: a framework for static and incremental parallel graph connectivity algorithms. *Proceedings of the VLDB Endowment* 14, 4 (2020), 653–667. https://doi.org/10.14778/3436905.3436923

[13] A. M. Fiscarelli, M. R. Brust, G. Danoy, and P. Bouvry. 2019. A Memory-Based Label Propagation Algorithm for Community Detection. In *Complex Networks and Their Applications VII (Studies in Computational Intelligence)*. Springer International Publishing, 171–182. https://doi.org/10.1007/978-3-030-05411-3_14

[14] S. Fortunato. 2010. Community detection in graphs. *Physics Reports* 486, 3 (Feb. 2010), 75–174. https://doi.org/10.1016/j.physrep.2009.11.002

[15] Y. Gao, X. Yu, and H. Zhang. 2020. Uncovering overlapping community structure in static and dynamic networks. *Knowledge-Based Systems* 201-202 (2020), 106060. https://doi.org/10.1016/j.knosys.2020.106060

[16] S. E. Garza and S. E. Schaeffer. 2019. Community detection with the Label Propagation Algorithm: A survey. *Physica A: Statistical Mechanics and its Applications* 534 (2019), 122058. https://doi.org/10.1016/j.physa.2019.122058

[17] C. Granell, S. Gómez, and A. Arenas. 2012. Hierarchical multiresolution method to overcome the resolution limit in complex networks. *International Journal of Bifurcation and Chaos* 22, 7 (2012), 1250171. https://doi.org/10.1142/S0218127412501714

[18] R. Hosseini and R. Azmi. 2015. Memory-based label propagation algorithm for community detection in social networks. In *2015 The International Symposium on Artificial Intelligence and Signal Processing (AISP)*. IEEE, 256–260. https://doi.org/10.1109/AISP.2015.7123488

[19] J. Jaiganesh and M. Burtscher. 2018. A High-performance Connected Components Implementation for GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA) *(HPDC '18)*. ACM, 92–104. https://doi.org/10.1145/3208040.3208041

[20] J. Jájá. 1992. *An Introduction to Parallel Algorithms*. Addison-Wesley.

[21] S. V. Jayanti and R. E. Tarjan. 2016. A Randomized Concurrent Algorithm for Disjoint Set Union. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing* (Chicago, Illinois, USA, 2016-07-25) *(PODC '16)*. Association for Computing Machinery, 75–82. https://doi.org/10.1145/2933057.2933108

[22] U. Kang, C. E. Tsourakakis, and C. Faloutsos. 2009. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM '09)*. IEEE Computer Society, USA, 229–238. https://doi.org/10.1109/ICDM.2009.14

[23] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, D. Staheli, and S. Smith. 2017. Streaming graph challenge: Stochastic block partition. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–12. https://doi.org/10.1109/HPEC.2017.8091040

[24] I. M. Kloumann and J. M. Kleinberg. 2014. Community Membership Identification from Small Seed Sets. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2014) *(KDD '14)*. ACM, 1366–1375. https://doi.org/10.1145/2623330.2623621

[25] J. M. Klusowski and Y. Wu. 2018. Counting Motifs with Graph Sampling. In *Proceedings of the 31st Conference On Learning Theory (Proceedings of Machine Learning Research, Vol. 75)*, S. Bubeck, V. Perchet, and P. Rigollet (Eds.). PMLR, 1966–2011. http://proceedings.mlr.press/v75/klusowski18a.html

[26] Y. Kozawa, T. Amagasa, and H. Kitagawa. 2017. GPU-Accelerated Graph Clustering via Parallel Label Propagation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (New York, NY, USA) *(CIKM '17)*. ACM, 567–576. https://doi.org/10.1145/3132847.3132960

[27] J. Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22Nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13 Companion)*. ACM, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173

[28] K. Kuzmin, M. Chen, and B. K. Szymanski. 2015. Parallelizing SLPA for scalable overlapping community detection. *Scientific Programming* 2015 (2015), 4:4. https://doi.org/10.1155/2015/461362

[29] A. Kyrola, G. Blelloch, and C. Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 31–46. https://doi.org/10.5555/2387880.2387884

[30] A. Lancichinetti, S. Fortunato, and F. Radicchi. 2008. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E* 78, 4 (Oct. 2008), 046110. https://doi.org/10.1103/PhysRevE.78.046110

[31] A. Lancichinetti, F. Radicchi, J. J. Ramasco, and S. Fortunato. 2011. Finding Statistically Significant Communities in Networks. *PLOS ONE* 6, 4 (2011), e18961. https://doi.org/10.1371/journal.pone.0018961

[32] C. E. Leiserson and T. B. Schardl. 2010. A Work-efficient Parallel Breadth-first Search Algorithm (or How to Cope with the Nondeterminism of Reducers). In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2010) *(SPAA '10)*. ACM, 303–314. https://doi.org/10.1145/1810479.1810534

[33] J. Leskovec and A. Krevl. 2016. SNAP datasets: Stanford large network dataset collection; 2014. *http://snap. stanford. edu/data* (2016).

[34] I. X. Y. Leung, P. Hui, P. Liò, and J. Crowcroft. 2009. Towards real-time community detection in large networks. *Phys. Rev. E* 79, 6 (2009), 066107. https://doi.org/10.1103/PhysRevE.79.066107

[35] Z. Lin, X. Zheng, N. Xin, and D. Chen. 2014. CK-LPA: Efficient community detection algorithm based on label propagation with community kernel. *Physica A: Statistical Mechanics and its Applications* 416 (2014), 386–399. https://doi.org/10.1016/j.physa.2014.09.023

[36] S. Liu and R. E. Tarjan. 2018. Simple Concurrent Labeling Algorithms for Connected Components. In *2nd Symposium on Simplicity in Algorithms (SOSA 2019) (OpenAccess Series in Informatics (OASIcs), Vol. 69)*, Jeremy T. Fineman and Michael Mitzenmacher (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:20. https://doi.org/10.4230/OASIcs.SOSA.2019.3

[37] X. Liu and T. Murata. 2010. Advanced modularity-specialized label propagation algorithm for detecting communities in networks. *Physica A: Statistical Mechanics and its Applications* 389, 7 (2010), 1493–1500. https://doi.org/10.1016/j.physa.2009.12.019

[38] X. T. Liu, J. S. Firoz, M. Zalewski, M. Halappanavar, K. J. Barker, A. Lumsdaine, and A. H. Gebremedhin. 2019. Distributed Direction-Optimizing Label Propagation for Community Detection. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2019.8916215

[39] X. T. Liu, M. Halappanavar, K. J. Barker, A. Lumsdaine, and A. H. Gebremedhin. 2020. Direction-Optimizing Label Propagation and Its Application to Community Detection. In *Proceedings of the 17th ACM International Conference on Computing Frontiers* (Catania, Sicily, Italy) *(CF '20)*. ACM, New York, NY, USA, 192–201. https://doi.org/10.1145/3387902.3392634

[40] H. Lu, M. Halappanavar, and A. Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel Comput.* 47 (2015), 19 – 37. https://doi.org/10.1016/j.parco.2015.03.003

[41] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *2009 IEEE International Symposium on Parallel Distributed Processing*. IEEE, 1–8. https://doi.org/10.1109/IPDPS.2009.5161100

[42] A. McDaid and N. Hurley. 2010. Detecting Highly Overlapping Communities with Model-Based Overlapping Seed Expansion. In *2010 International Conference on Advances in Social Networks Analysis and Mining* (Odense, Denmark). IEEE, 112–119. https://doi.org/10.1109/ASONAM.2010.77

[43] D. Merrill, M. Garland, and A. Grimshaw. 2012. Scalable GPU Graph Traversal. *SIGPLAN Not.* 47, 8 (2012), 117–128. https://doi.org/10.1145/2370036.2145832

[44] S. M. Orzan. 2004. *On Distributed Verification and Verified Distribution.* Ph.D. thesis. VRIJE UNIVERSITEIT. http://dare.ubvu.vu.nl/handle/1871/10338

[45] G. Palla, I. Derényi, I. Farkas, and T. Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *Nature* 435, 7043 (2005), 814–818. https://doi.org/10.1038/nature03607

[46] U. N. Raghavan, R. Albert, and S. Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E* 76, 3 (2007), 036106. https://doi.org/10.1103/PhysRevE.76.036106

[47] R. A. Rossi and N. K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) *(AAAI'15)*. AAAI Press, 4292–4293. https://doi.org/10.5555/2888116.2888372

[48] J. Scripps, P. Tan, and A. Esfahanian. 2007. Exploration of Link Structure and Community-Based Node Roles in Network Analysis. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 649–654. https://doi.org/10.1109/ICDM.2007.37

[49] H. Shen, X. Cheng, K. Cai, and M. Hu. 2009. Detect overlapping and hierarchical community structure in networks. *Physica A: Statistical Mechanics and its Applications* 388, 8 (2009), 1706–1712. https://doi.org/10.1016/j.physa.2008.12.021

[50] Y. Shiloach and U. Vishkin. 1982. An O(logn) parallel connectivity algorithm. *Journal of Algorithms* 3, 1 (1982), 57–67. https://doi.org/10.1016/0196-6774(82)90008-6

[51] J. Shun and G. E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Notices* 48, 8 (2013), 135–146. https://doi.org/10.1145/2517327.2442530

[52] J. Shun, L. Dhulipala, and G. Blelloch. 2014. A Simple and Practical Linear-work Parallel Algorithm for Connectivity. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA) *(SPAA '14)*. ACM, 143–153. https://doi.org/10.1145/2612669.2612692

[53] G. M. Slota, S. Rajamanickam, and K. Madduri. 2014. BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 550–559. https://doi.org/10.1109/IPDPS.2014.64

[54] S. N. Soffer and A. Vàzquez. 2005. Network clustering coefficient without degree-correlation biases. *Phys. Rev. E* 71, 5 (2005), 057101. https://doi.org/10.1103/PhysRevE.71.057101

[55] J. Soman and A. Narang. 2011. Fast Community Detection Algorithm with GPUs and Multicore Architectures. In *2011 IEEE International Parallel Distributed Processing Symposium*. IEEE, 568–579. https://doi.org/10.1109/IPDPS.2011.61

[56] C. L. Staudt and H. Meyerhenke. 2016. Engineering Parallel Algorithms for Community Detection in Massive Networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2016), 171–184. https://doi.org/10.1109/TPDS.2015.2390633

[57] S. Stergiou, D. Rughwani, and K. Tsioutsiouliklis. 2018. Shortcutting Label Propagation for Distributed Connected Components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (New York, NY, USA) *(WSDM '18)*. ACM, 540–546. https://doi.org/10.1145/3159652.3159696

[58] A. Stoica, J. X. Han, and A. Chaintreau. 2020. Seeding Network Influence in Biased Networks and the Benefits of Diversity. In *Proceedings of The Web Conference 2020* (New York, NY, USA) *(WWW '20)*. ACM, 2089–2098. https://doi.org/10.1145/3366423.3380275

[59] M. Sutton, T. Ben-Nun, and A. Barak. 2018. Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 12–21. https://doi.org/10.1109/IPDPS.2018.00012

[60] J. J. Tithi, A. Stasiak, S. Aananthakrishnan, and F. Petrini. 2020. Prune the Unnecessary: Parallel Pull-Push Louvain Algorithms with Automatic Edge Pruning. In *49th International Conference on Parallel Processing - ICPP* (New York, NY, USA) *(ICPP '20)*. ACM, 1–11. https://doi.org/10.1145/3404397.3404455

[61] J. J. Whang, D. F. Gleich, and I. S. Dhillon. 2016. Overlapping Community Detection Using Neighborhood-Inflated Seed Expansion. *IEEE Transactions on Knowledge and Data Engineering* 28, 5 (2016), 1272–1284. https://doi.org/10.1109/TKDE.2016.2518687

[62] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali. 2015. Scalable Data-Driven PageRank: Algorithms, System Issues, and Lessons Learned. In *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 438–450. https://doi.org/10.1007/978-3-662-48096-0_34

[63] J. Xie and B. K. Szymanski. 2013. LabelRank: A stabilized label propagation algorithm for community detection in networks. In *2013 IEEE 2nd Network Science Workshop (NSW)*. IEEE, 138–143. https://doi.org/10.1109/NSW.2013.6609210

[64] Y. Xing, F. Meng, Y. Zhou, M. Zhu, M. Shi, and G. Sun. 2014. A Node Influence Based Label Propagation Algorithm for Community Detection in Networks. *The Scientific World Journal* 2014 (2014). https://doi.org/10.1155/2014/627581

[65] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. 2014. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. *Proc. VLDB Endow.* 7, 14 (2014), 1821–1832. https://doi.org/10.14778/2733085.2733089

[66] J. Yang and J. Leskovec. 2015. Defining and Evaluating Network Communities Based on Ground-Truth. *Knowl. Inf. Syst.* 42, 1 (2015), 181–213. https://doi.org/10.1007/s10115-013-0693-z

[67] H. Zhang, Y. Gao, and Y. Zhang. 2018. Overlapping communities from dense disjoint and high total degree clusters. *Physica A: Statistical Mechanics and its Applications* 496 (2018), 286–298. https://doi.org/10.1016/j.physa.2017.12.146

[68] K. Zhang, R. Chen, and H. Chen. 2015. NUMA-aware Graph-structured Analytics. *SIGPLAN Not.* 50, 8 (Jan. 2015), 183–193. https://doi.org/10.1145/2858788.2688507

[69] Q. Zhang, Q. Qiu, W. Guo, K. Guo, and N. Xiong. 2016. A social community detection algorithm based on parallel grey label propagation. *Computer Networks* 107 (2016), 133–143. https://doi.org/10.1016/j.comnet.2016.06.002

[70] X. Zhang, F. Song, S. Chen, X. Tian, and Y. Ao. 2015. Label propagation algorithm based on local cycles for community detection. *International Journal of Modern Physics B* 29, 5 (2015), 1550029. https://doi.org/10.1142/S0217979215500290

[71] Y. Zhang, A. Azad, and A. Buluç. 2020. Parallel algorithms for finding connected components using linear algebra. *J. Parallel and Distrib. Comput.* 144 (2020), 14–27. https://doi.org/10.1016/j.jpdc.2020.04.009

[72] Y. Zhao, S. Li, and F. Jin. 2016. Identification of influential nodes in social networks with community structure based on label propagation. *Neurocomputing* 210 (2016), 34–44. https://doi.org/10.1016/j.neucom.2015.11.125

[73] X. Zhu and Z. Ghahramani. 2002. *Learning from Labeled and Unlabeled Data with Label Propagation.* Technical Report. CMU.