Direction-optimizing Label Propagation and its Application to Community Detection

Xu T. Liu*
Washington State University
Pullman, WA
xu.liu2@wsu.edu

Mahantesh Halappanavar, Kevin J. Barker
Pacific Northwest National Laboratory
Richland, WA
{mahantesh.halappanavar,kevin.barker}@pnnl.gov

Andrew Lumsdaine†
Pacific Northwest National Laboratory
Richland, WA
andrew.lumsdaine@pnnl.gov

Assefaw H. Gebremedhin
Washington State University
Pullman, WA
assefaw.gebremedhin@wsu.edu

ABSTRACT

Label Propagation, while more commonly known as a machine learning algorithm for classification, is also an effective method for detecting communities in networks. We propose a new Direction Optimizing Label Propagation Algorithm (DOLPA) that relies on the use of *frontiers* and alternates between label *push* and label *pull* operations to enhance the performance of the standard Label Propagation Algorithm (LPA). Specifically, DOLPA has parameters for tuning the processing order of vertices in a graph, which in turn reduces the number of edges visited and improves the quality of solution obtained. We apply DOLPA to the community detection problem, present the design and implementation of the algorithm, and discuss its shared-memory parallelization using OpenMP. Empirically, we evaluate our algorithm using synthetic graphs as well as real-world networks. Compared with the state-of-the-art *Parallel Label Propagation* algorithm, we achieve at least two times the F-Score while reducing the runtime by 50% for synthetic graphs with overlapping communities. We also compare DOLPA against state of the art parallel implementation of the Louvain method using the same graphs and show that DOLPA achieves about three times the F-Score at 10% the runtime.

CCS CONCEPTS

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

KEYWORDS

Community Detection, Label Propagation, Direction-optimizing

ACM Reference Format:
Xu T. Liu, Mahantesh Halappanavar, Kevin J. Barker, Andrew Lumsdaine, and Assefaw H. Gebremedhin. 2020. Direction-optimizing Label Propagation and its Application to Community Detection. In *17th ACM International Conference on Computing Frontiers (CF '20), May 11–13, 2020, Catania, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3387902.3392634

*Also with Pacific Northwest National Laboratory.
†Also with University of Washington.

CF '20, May 11–13, 2020, Catania, Italy

ACM ISBN 978-1-4503-7956-4/20/05…$15.00
https://doi.org/10.1145/3387902.3392634

1 INTRODUCTION

The label propagation algorithm (LPA) is a machine learning algorithm for data classification where label information is propagated from labeled to unlabeled entities within a network [42]. The method works iteratively; initially, a (small) subset of the data points have labels, and the labels are gradually propagated to the unlabeled points until all the data points are properly labeled. Raghavan et al. [28] showed that LPA could be an effective method for identifying communities in networks.

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* (or *cluster*)—in social, biological and technological networks. For example, video sharing services such as YouTube cluster users with common viewing interests together to enable recommendation systems to provide better services. As sizes of networks continue to increase dramatically, we generally need fast algorithms to enable large-scale real-time graph processing. 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 graphs with billions of edges.

In parallel graph processing, different graph algorithms use different processing orders on vertices. One extreme is the case where the processing order is completely random. Another extreme is where the processing order is strictly sequential. The Parallel Label Propagation (PLP) algorithm proposed by Staudt and Meyerhenke [34] belongs to the first type (random order). Instead of explicitly randomizing the node order, PLP relies on the randomization introduced via multi-threading. PLP produces different results for different runs due to its random selection on labels whenever there is a tie on the most common labels. Inherited from the "epidemic" behavior of LPA [20], PLP forms a "monster community", where the dominant label of the large component "plagues" the labels of other communities, resulting in low quality solution.

In this paper, we propose a frontier-based Label Propagation Algorithm, called Direction-Optimizing Label Propagation Algorithm (DOLPA), that enforces processing order on vertices (or the

propagation order of the labels). To achieve this, DOLPA maintains *frontiers* and *switches* between two abstractions, called a PULL operation and a PUSH operation. We apply DOLPA to the community detection problem and empirically show that the processing order DOLPA enforces reduces runtime and improves the quality of solution compared to PLP.

Specifically, DOLPA provides a "knob" through which a trade-off between runtime and quality of solution is achieved. The trade-off comes from our frontier-based implementation combined with the direction optimization organization, and has several elements.

First, the frontiers enforce two kinds of the processing orders on the vertices. Each frontier in each iteration enforces *explicit order* on the vertices. The vertices in the front of the frontier have a better chance to propagate their labels than those in the back. In addition, frontiers generated in each iteration enforce an *implicit order* on vertices. The vertices in the initial frontier propagate their labels earlier than their adjacent vertices which are added to the next frontier. This implicit order is found to yield higher quality of solution (§5.3). Second, in addition to reducing the number of the edges visited, PUSH results in a higher probability of forming a stable community core than PULL. Third, by adding the adjacent vertices of a recently updated vertex to the next frontier, we can reach peak workload faster and therefore attain higher performance if that vertex has a high degree.

We evaluate the performance of our OpenMP DOLPA implementation and the quality of solution produced with both synthetic graphs and real-world graphs, and show that, compared with PLP, DOLPA achieves at least two times the F-Score while reducing the runtime by 50%. We also compare DOLPA with a state of the art parallel implementation of the Louvain method available in Grappolo [24] and find DOLPA to be both faster (speedup of four) and more accurate (three times in F-Score). The Louvain method is based on on modularity maximization. Modularity [27] is a broadly adopted metric for evaluating the quality of clustering obtained by a community detection algorithm.

Summary of contributions. In this work, we:

- Introduce a new LP algorithm (DOLPA) that uses frontier and applies direction optimization to LPA (§3).
- Apply DOLPA to community detection and parallelize our algorithm under shared-memory programming model (§4).
- Demonstrate the trade-off between time and quality provided by DOLPA using carefully designed microbenchmarks and show the benefits of the push and pull abstractions (§5.3).

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 literature and a variety of different algorithms have been suggested for solving it. See the survey paper by Fortunato [9] for a comprehensive review.

One of the simplest and fastest algorithms for community detection is label propagation (LP). Garza and Schaeffer have a detailed investigation of LP-based approaches [10]. Section 6 offers a brief review of related work. Since it forms the basis for our proposed algorithm we briefly discuss the standard LP algorithm here.

Algorithm 1 PLP: Parallel Label Propagation [34].
Input: Graph $G = (V, E)$, termination threshold θ
Output: $\forall\, v \in V$: $label[v]$ = 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:             atomic increment updated
13:             V_active ← V_active ∪ {v's neighbors}
14:         else
15:             V_active ← V_active \ {v}
```

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) [34]. 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 the 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. $O(km)$, where k is the number of iterations and m is the number of edges.

3 DIRECTION-OPTIMIZING LABEL PROPAGATION

In the early iterations of LPA, instead of passively propagating labels, we choose a more "important" label of a vertex v in the network and broadcast the label to v's adjacent vertices aggressively. We abstract this kind of label update as a PUSH operation. Furthermore, in LPA, for each vertex v, the labels of v's adjacent vertices are queried to obtain the maximum 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 from one set of start vertices. By pre-selecting "seed" vertices (§4.1) into a frontier, we jump start the first iteration. When

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

Algorithm 3 Pseudocode for PUSH.

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

Algorithm 4 Pseudocode for PULL.

```
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]])
```

the workload is small in the early iterations, we apply PUSH for label updates; when the iteration number reaches a *direction switch* threshold, we switch to PULL for label updates. This idea has been called *direction optimization* by Beamer et al. [4] in their parallel Breadth-First Search work and inspires our work.

DOLPA is summarized in Algorithm 2. PUSH is listed as Algorithm 3 and PULL as Algorithm 4. 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 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 a *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. Vertices having the same label belong to the same group.

Complexity of DOLPA. We analyze DOLPA's complexity in the sequential setting here. 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, and the space cost of the frontier is $O(n)$. Therefore, the runtime of DOLPA is bounded by $O(km)$, where k is the number of iterations and m is the number of edges. In practice, we notice DOLPA requires more iterations than LPA to converge. But k still remains constant. Hence, DOLPA is nearly linear, just as the standard LPA.

4 APPLICATION TO COMMUNITY DETECTION

We discuss in this section how we adopt DOLPA for community detection. We begin by defining what seed vertices are and discussing how other works select them. Then we outline and address various challenges that arise in parallelizing DOLPA for community detection.

4.1 Seeding Strategies

A seed (or an influential) vertex is 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 the seed of a community [41]. In general, a node with more connections could be viewed as more important in the network [37]. It has been proven that the membership contribution of a vertex to a community is highly related to its degree [30].

Kloumann and Kleinberg [15] found that the 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 vertex 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. We refer to this as a *high degree seeding strategy*. As a baseline, we also randomly select vertices as seeds and refer to the approach as a *random seeding strategy*. In both cases, we select a fraction τ of the vertices as seeds in DOLPA.

4.2 Challenges in Parallelizing DOLPA

We discuss below issues that arise in parallelizing DOLPA for community detection and our approaches for addressing them.

4.2.1 Frontier Expansion. In DOLPA, the next frontier expands as either PUSH or PULL updates a 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. In DOLPA, we use a bitmap [26] to avoid duplicate entries

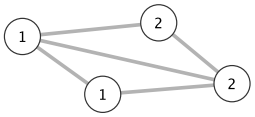

Figure 1: An illustration of swapping labels.

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. Hence, the frontier expansion rate under the high degree seeding strategy is higher than that under the random seeding strategy.

4.2.2 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 most common in its neighborhood. 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* [28]. It also happens in the serial algorithm on a bipartite graph. To detect and prevent this, 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, we detect a label swap. Then we manually mark the vertex as inactive and move on.

Ignoring label swapping is feasible 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 beats the number of label swapping.

4.2.3 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 PUSH or PULL task is performed on each vertex in the frontier. The fine-grained parallelism is within PUSH and PULL, where the label propagation is performed on the neighborhood of the vertex v. The reason to introduce two levels of parallelism is to solve the workload imbalance at the task level, where the vertices could have various number of neighbors. With two level parallelism, DOLPA balances the workload.

If 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, which does not harm correctness of the algorithm but hurts performance. Benign race entails repeated work and makes the algorithm non-deterministic.

Each PULL operation is almost data independent without considering inserting 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 but just 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).

Within PULL, there are concurrent reads on the labels of the neighbors of the 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.

5 EXPERIMENTAL EVALUATION

We present our experimental evaluation results in this section. We use the Lancichinetti-Fortunato-Radicchi (LFR) benchmark [19] 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 benchmark 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. We begin the section detailing the experimental setup and then present the results.

5.1 Experimental Setup

We used one node with two Intel Xeon E5-2699v3 processors at 2.3 GHz, 18 physical cores per socket, 72 logical cores per node with hyper-threading, 48 MB L3 cache, and with 128 GB main memory. We used GCC 8.2.0 compiler with -O3 compilation option to build the codes. We used OpenMP 4.0 for parallelization with *guided* scheduling.

5.2 Metrics for Quality of Solution

We quantify the quality of solutions using Precision, Recall and F-Score, which are defined below. We list all possible pairs of vertices in a community structure obtained from DOLPA as C_D and the ground truth information as C_G. Community assignment of a vertex x in C_D is denoted as C_D^x, and similarly, that in C_G is denoted as C_G^x. Each pair x, y of vertices with respect to the community assignments of C_D and C_G belong to one of the three categories:

- True Positive (TP): x, y belong to the same community in C_D and C_G, i.e., $C_D^x = C_D^y$ and $C_G^x = C_G^y$;
- False Negative (FN): x, y belong to the same community only in C_G, i.e., $C_D^x \neq C_D^y$ and $C_G^x = C_G^y$;
- False Positive (FP): x, y belong to the same community only in C_D, i.e., $C_D^x = C_D^y$ and $C_G^x \neq C_G^y$;

Based on the above definitions, we define the metrics of Precision, Recall and F-Score as follow:

- Precision $P = \frac{TP}{TP+FP}$
- Recall $R = \frac{TP}{TP+FN}$
- F-Score $F = \frac{2*P*R}{P+R}$

F-Score is the harmonic mean of Precision and Recall.

5.3 Benchmarking Direction Optimization

To gain insight into the performance of the PUSH and PULL operations in practice, we design a microbenchmark to analyze their behaviors. In this subsection, we describe the design, present the raw results and then provide analysis of the results. The insights gained give a better understanding of DOLPA.

Table 1: DOLPA parameter settings in microbenchmark

Version	τ	ω	Description
DO	1	1	$\|V\|$ seeds, direction switch at #iter 1
DO1	$\|V\|/\|E\|$	1	$\tau \cdot \|V\|$ seeds, direction switch at #iter 1
DO2	$\|V\|/\|E\|$	2	$\tau \cdot \|V\|$ seeds, direction switch at #iter 2
DO3	$\|V\|/\|E\|$	3	$\tau \cdot \|V\|$ seeds, direction switch at #iter 3
DO4	$\|V\|/\|E\|$	4	$\tau \cdot \|V\|$ seeds, direction switch at #iter 4

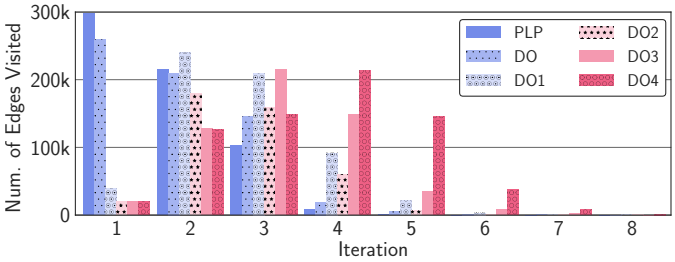

Figure 2: Iteration versus edge in PLP and DOLPA variants on a LFR graph with 10,000 vertices and 76,754 edges. DO and DO1 switch from PUSH to PULL at iteration 1, i.e., DO and DO1 use no PUSH. DO has all the vertices as seeds while DO1 randomly selects 1,302 vertices as the seeds. DO is simply PLP using the frontiers. This is why PLP and DO have similar workload for each iteration.

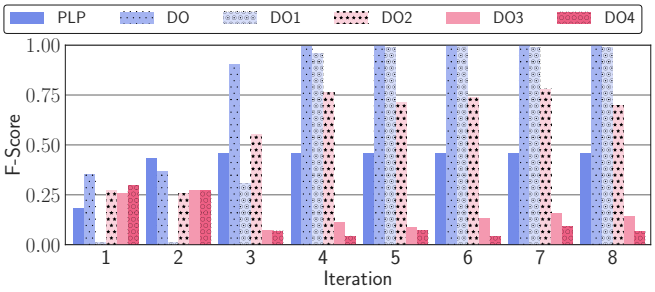

Figure 3: Iteration versus F-Score in PLP and DOLPA variants on a LFR benchmark. After the convergence (at iteration 8), the F-Score of DO and DO1 is nearly 1 and the F-Score decreases as more PUSH are applied. The F-Score of DO2 oscillates from iteration 4 to iteration 8 due to the label swapping mentioned in Sec. 4.2.2.

5.3.1 Microbenchmark Design. We explore the combination of the seeding parameter τ and the switch threshold ω so that the switch from PUSH to PULL in DOLPA happens at iteration ω. We adopt the random seeding strategy to be fair. Table 1 lists the parameter settings and names of five benchmarks. We use the seed parameter τ of $|V|/|E|$ as default. The design choice for this is omitted for space considerations. We compare these against each other and against PLP, our re-implementation of the PLP algorithm.

The experiments are run on an LFR benchmark with 10,000 vertices and 76,754 edges, average degree 15, max degree 50, 20 minimum communities, 50 maximum communities and a mixing parameter of 0.3. We clock the time for three major steps of the algorithm and collect statistical results on the workload, i.e., the number of edges being processed at each iteration. The experiments are run in single thread to avoid any non-deterministic behaviors.

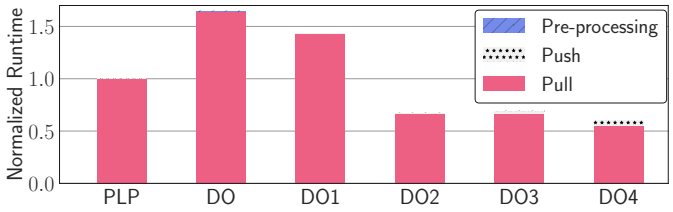

Figure 4: Normalized runtime (relative to PLP) of PLP and DOLPA variants on a LFR benchmark in single thread. Three types of computation are clocked. DO4 has best runtime. DO has the worst runtime. The runtime decreases as more PUSH are applied.

5.3.2 The Results. Figure 2 shows the number of edges visited versus iteration of PLP and the five DOLPA benchmark cases. It can be seen that: PLP and DO process all the edges in the first iteration and converge in the 6th iteration; The workloads of DO1 and DO2 peak at iteration 2; DO3's workload peaks at iteration 3; and DO4's workload peaks at iteration 4.

The iteration versus F-Score is shown in Figure 3 and the normalized runtime is shown in Figure 4. The variants DO and DO1 have nearly perfect F-Score. DO4 takes half the runtime of PLP. Both the F-Score and runtime decrease as more PUSH operations are applied.

5.3.3 Analysis. We make several important observations from the results in Figure 2, Figure 3 and Figure 4 and discuss them below.

i) DOLPA without PUSH is LPA: DO is LPA that uses frontiers. The close numbers of edges visited at each iteration of PLP and DO in Figure 2 confirm this. Since we randomly insert the vertices of the entire network into the frontier of DO, the processing orders of the vertices of PLP and DO are different. However, the final F-Score of DO is over two times the F-Score of PLP in Figure 3, which leads to our next insight.

ii) Early update on more important vertices yields higher accuracy: PUSH or PULL advocate seed vertices labels in the early iterations. This will produce a candidate label search space where the labels in this space are more important than the labels in later spaces. Furthermore, a label updated from this space at this stage will most likely survive. This is validated by DO2 in Figure 2, where its workload decreases dramatically at iteration 4 because there are few label updates at iteration 3 where DO2 has its peak workload. With a peak workload and a small number of label update, which means most of the vertices in iteration 3 already have true maximum labels, we can conclude that the label search space that iteration 2 produces is highly efficient.

iii) PUSH visits less edges but entails benign race: The runtime decreases as more PUSH steps applied in Figure 4. This is because PUSH visits less edges than PULL. The amortized cost for each label update of PUSH is O(1) while that of PULL is O($d(v)$). For each label update, the number of edges visited by PUSH on the vertex v is $d(v) \times \overline{d}(N(v))$, where $\overline{d}(N(v))$ is the average degree in the neighborhood $N(v)$ of the vertex v. With 1 iteration of PUSH on 1,302 seeds, DO2 saves on average $1,302 \times 15$ edges visited, where 15 is the average degree of this graph. With 2 iterations of PUSH on the vertices in the frontier, DO3 visits at most $(1,302 + 1,302 \times 15) \times 15$ less edges. This is the reason why DO2 performs better than DO1 and DO3 performs better than DO2 in Figure 4. Comparing the

Table 2: Community detection algorithms studied

Version	Description
PLP	Parallel Label Propagation algorithm [34]
PUR	DOLPA using PULL only with random seeds
PUH	DOLPA using PULL only with high degree seeds
DOR	DOLPA using PUSH & PULL with random seeds
DOH	DOLPA using PUSH & PULL with high degree seeds
LV	parallel Louvain method implementation [24]

runtime difference between DO1 and DO2 versus DO3 and DO4 in Figure 4, the gap dramatically decreases due to the benign race introduced by PUSH.

In our microbenchmark, we adopt random seeding strategy. We predict that PUSH visits even less number of edges with high degree seeding strategy. Hence the runtime under high degree seeding strategy would be lower than that under random seeding strategy.

iv) Too many PUSHs deteriorate the quality of solution: Without *a priori* information, PUSH may contribute negatively to a community especially when the vertex is on the border of two community structures, as the label of one community can "poison" the labels of the vertices belonging to the other. If we compare DO2 with DO3 in Figure 3, the F-Score of DO2 and that of DO3 are close at iteration 2. Then the F-Score of DO2 at iteration 3 doubles after switching to PULL. In contrast, the F-Score of DO3 drops dramatically with one more iteration of PUSH than DO2. For this reason, we recommend label PUSH at the early stages of community forming. For latter stages when the maximum label propagates near the border of two communities, PULL is more delicate for maximum-label selection for high-quality community expansion. This is also the reason why we choose not to interleave PUSH and PULL.

5.3.4 PUSH or PULL, What to choose? Since PUSH is cheaper than PULL in updating labels, we prefer PUSH than PULL for fast label propagation. However, too many PUSHs can degrade the quality of solution. The decision for PUSH is made *a priori*, i.e., we select a set of seeds during pre-processing step and apply PUSH on their labels. In contrast, the decision for PULL is made on the fly, i.e., we perform an on-the-fly most common labels query in the neighborhood, calculate the degree/weight associated with those labels, then select the maximum label among them.

Combining the runtime results in Figure 4 and the final F-Score at iteration 8 in Figure 3, we find the best parameter combination of τ and ω for DOLPA. To achieve the best runtime with reasonable quality of solution, we set DOLPA as DO2 in Table 1 where τ is $|V|/|E|$ and ω is 2. To obtain the best quality of solution in reasonable runtime, we set DOLPA as DO1 in Table 1 where τ is $|V|/|E|$ and ω is 1. Recall we adopt random seeding strategy in our microbenchmark. We notice these two parameter setting rules also apply to the high degree seeding strategy.

5.4 Runtime and Solution Quality Evaluation

We evaluate the quality of solution and runtime using the synthetic graphs and real-world graphs listed in Table 3.

Table 2 lists the variants of parallel LPA we study (PLP, PU and DO) and the Louvain method (LV). PUR and DOR select random vertices as seeds. PUH and DOH select high degree vertices as seeds. Both PU and DO use the seeding parameter τ of $|V|/|E|$. We use the switch threshold ω of 1 for PU and that of 2 for DO. Recall when ω is 1, there is no PUSH.

5.4.1 Experiment Methodology. We generated two groups of the LFR benchmarks—**B**ig and **S**mall: the community size of group Big ranges from 20 to 200 and the community size of group Small ranges from 10 to 100. The fraction of overlapping vertices is 10; the number of memberships of the overlapping vertices is 2. Since the methods we studied (Table 2) perform well on the non-overlapping community detection, we only pay attention to the overlapping communities. We generated a group of low overlapping density network B1 and S1 (mixing parameter 0.1), and a group of medium overlapping density network B3 and S3 (mixing parameter 0.3). B1 and B3 have the same parameters except the mixing parameter; likewise for S1 and S3. We generated 10 for each benchmark. The synthetic graphs B1, B3, S1 and S3 are listed in the top portion in Table 3. The middle portion of the table lists synthetic graphs we downloaded as images from the 2019 Stochastic Block Partitioning Graph Challenge [14]. The bottom portion of Table 3 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) method [11] as ground truth data. It is not true ground-truth, simply a reference.

5.4.2 Runtime and Quality of Solution Results. Table 4 shows the F-Score and runtime results on the six methods listed in Table 2. The results show that DOLPA outperforms PLP as well as the Louvain method in both runtime and F-Score. DOLPA achieves 10 best runtimes out of 11 and 8 best F-Scores out of 11. DOLPA adopting high degree seeds achieves 6 best runtimes among 10 (all 6 by DOH). DOLPA adopting random seeding strategy achieves 6 best F-Scores among 8 best F-Scores (each 3 best F-Scores by PUR and DOR). We achieve at least two times the F-Score while reducing the runtime by 50% for the LFR graphs. Compared with Louvain method using the same graphs, the best results achieved by DOLPA have an average of three times the F-Score at 1/10 the runtime. We provide further discussions on Table 4 in the remainder of this subsection.

i) Frontier: The frontier of DOLPA is beneficial. A frontier can process the vertices in the order we want them to. During the initialization of DOLPA, we add the seeds to the initial frontier such that the labels of seeds can be propagated first. This makes the labels of the seeds to have a higher probability to form a strong community core without being eliminated. As we observed from the experiments, PLP and PU have the similar numbers of the label updates, the propagate steps and the processed edges. The high Precision of PU proves PU to be more efficient and accurate in finding the "right" maximum label. In Table 4, PUH and PUR has 2.5 and 3.2 times the F-Score than PLP on the graphs B3 and S3. PUH has comparable runtime with PLP. Note runtime of PUH includes the steps such as degree sorting and frontier insertion. PUR has an average 40% of runtime decrease without degree sorting in the pre-processing step comparing with PUH.

ii) Seed Vertices: The "right" seeds improve accuracy. The seeds propagate before the 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

Table 3: Synthetic and real-world graphs for performance and quality of solution evaluation

Input	Description	$\|V\|$	$\|E\|$	Max Degree	Reference
B1	Generate using the LFR benchmark	1M	9.5M	100	[19]
B3	Generate using the LFR benchmark	1M	9.5M	100	[19]
S1	Generate using the LFR benchmark	1M	9.5M	100	[19]
S3	Generate using the LFR benchmark	1M	9.5M	100	[19]
LL	Low Block Overlap and Low Block Size Variation	1M	24M	122	[14]
LH	Low Block Overlap and high Block Size Variation	1M	24M	137	[14]
HL	High Block Overlap and Low Block Size Variation	1M	24M	104	[14]
HH	High Block Overlap and High Block Size Variation	1M	24M	180	[14]
fbnt	Facebook network	4M	24M	4,915	[29]
dblp	Coauthor-ship from DBLP	0.5M	15M	3,299	[2]
zbrp	Zhishi Baidu related pages	416K	2.4M	127,090	[17]
cond	Condensed matter collaborations	40K	176K	278	[2]

Table 4: The runtime and F-Score of the six methods listed in Table 2 on eight synthetic graphs with ground truth information. The real-world graphs use the "ground truth" data obtained from FTR method [11]. All results are obtained under 64 threads.

	Input	B1	S1	B3	S3	LL	LH	HL	HH	fnbt	dblp	cond
PLP	Time	2.30s	3.71s	3.03s	2.64s	1.76s	1.54s	**0.94s**	1.26s	24.85s	0.72s	**0.04s**
	Prec.	0.7301	0.5712	0.2526	0.1847	0.1305	0.0979	0.0097	0.0192	0.0784	0.0031	0.0008
	Recall	0.9998	1.0000	0.9999	0.9997	1.0000	0.9998	1.0000	1.0000	0.8852	0.9402	0.9690
	FScore	0.8440	0.7271	0.4033	0.3118	0.2309	0.1784	0.0192	0.0377	0.144	0.0062	0.0015
PUH	Time	2.49s	2.98s	2.56s	3.38s	1.47s	1.25s	1.04s	**0.89s**	4.24s	0.28s	0.06s
	Prec.	0.9991	0.9981	0.9951	0.9892	0.0125	0.0149	0.0188	0.0130	0.1154	0.0007	0.0013
	Recall	0.9999	1.0000	0.9998	0.9998	0.9999	0.9999	0.9998	0.9999	0.8159	0.9627	0.9293
	FScore	0.9995	**0.9990**	0.9974	**0.9945**	0.0248	0.0293	0.0370	0.0256	0.2022	0.0014	0.0026
DOH	Time	1.81s	2.72s	2.35s	2.95s	**0.72s**	**0.91s**	0.97s	0.94s	**2.45s**	**0.16s**	**0.04s**
	Prec.	0.1503	0.1262	0.0542	0.0294	0.0115	0.0149	0.0164	0.0139	0.1533	0.0004	0.0014
	Recall	0.9993	0.9991	0.9998	0.9997	0.9999	0.9999	0.9998	0.9999	0.7682	0.9105	0.5784
	FScore	0.2613	0.2083	0.1028	0.0571	0.0227	0.0294	0.0323	0.0275	0.2556	0.0008	0.0027
PUR	Time	1.48s	1.63s	2.08s	2.32s	1.35s	1.77s	1.61s	1.77s	8.52s	0.19s	0.06s
	Prec.	0.9998	0.9900	0.9968	0.9885	0.2018	0.0938	0.0286	0.0181	0.0870	0.2142	0.0044
	Recall	0.9998	0.9998	0.9997	0.9997	0.9999	0.9998	0.9998	0.9999	0.2620	0.8677	0.4116
	FScore	**0.9998**	0.9949	**0.9983**	0.9941	0.3358	0.1715	0.0556	0.0356	0.1306	**0.3436**	0.0088
DOR	Time	**1.36s**	**1.10s**	**1.51s**	**1.50s**	0.82s	1.14s	0.95s	**0.89s**	4.60s	**0.16s**	0.05s
	Prec.	0.8285	0.7848	0.6476	0.5706	0.9911	0.6770	0.8483	0.0530	0.1484	0.0914	0.0056
	Recall	0.9863	0.9511	0.9987	0.9960	0.9999	0.9993	0.9999	0.9998	0.2023	0.9171	0.4066
	FScore	0.9005	0.8600	0.7857	0.7255	**0.9955**	**0.8072**	**0.9179**	0.1007	0.1712	0.1663	0.0110
LV	Time	12.16s	10.46s	22.02s	16.38s	15.71s	13.32s	15.54s	22.06s	108.03s	10.43s	4.94s
	Prec.	0.2002	0.0732	0.0911	0.0536	0.2226	0.3417	0.1403	0.0802	0.3103	0.0072	0.0235
	Recall	1.0000	1.0000	1.0000	1.0000	0.8365	0.8062	0.5304	0.7353	1.0000	1.0000	1.0000
	FScore	0.3336	0.1365	0.1669	0.1018	0.3517	0.4800	0.2219	**0.1447**	**0.4736**	0.0143	**0.0460**

The input graphs in this table are listed from easy to hard for community detection algorithms. The results in bold are the best results among the six methods under the same graph. The runtime includes pre-processing steps such as degree sorting, frontier insertion, etc in PU/DO/LV. There was none in LP.

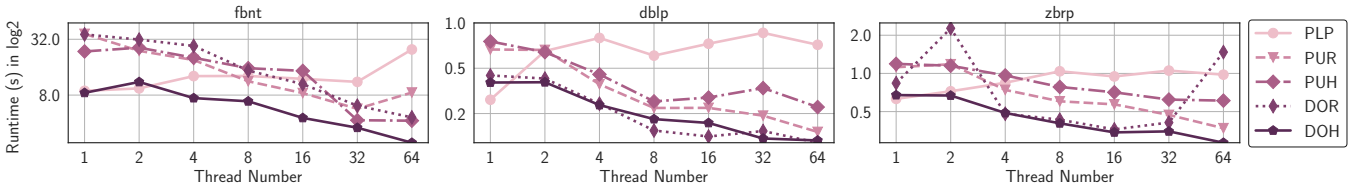

Figure 5: Strong scaling results up to 64 threads on three graphs for LP variants in Table 2. DOH has the best performance on three graphs.

will promote that label as the "true" maximum label because it now appears twice in this neighborhood. If the label the seed chooses survives in the next few iterations, the core of the community is formed. With the "right" choice on seeds, the probability of the

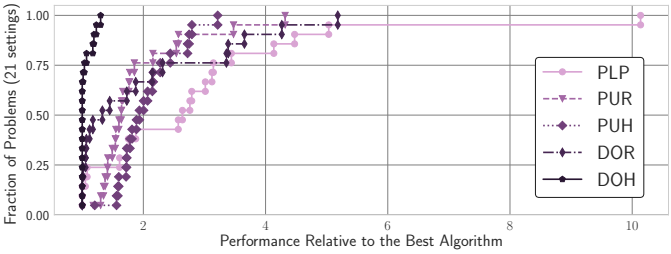

Figure 6: Performance profiles of PLP, DOH, DOR, PUH and PUR under various graph-thread counts settings.

seeds to form a strong and stable core is higher than other vertices. This proves that to update more important vertices in the network earlier than others works [37].

The "wrong" seeds decrease accuracy. DOH has the worst F-Score in most instances in Table 4. Pushing labels of the high degree vertices to their neighbors contribute negatively to the quality of the solution even though this guarantees a good runtime. DOR has less chance to "poison" other communities' labels when DOLPA uses the random seeds instead of the high degree seeds. This is shown by a greatly F-Score increase of DOR comparing with DOH.

iii) Direction Optimization: Direction optimization provides a trade-off for 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; DOR is faster than PUR. Comparing with PLP, DOR has an average 50% of runtime decrease on the LFR benchmarks and DOH has an average 15% of runtime decrease on graph challenge graphs. With a better seed selection, DOR has an average of 14 times F-Scores than PLP on graph challenge datasets.

Hence we can adjust ω for a good balance on different computation scenarios. DO fits best for a time-sensitive scenario. An appropriate switch threshold reduces work by applying certain amount of the PUSH operations on seeds hence higher performance. However, PUSH is harmful to the quality of solution in general as it forces an unwilling label choice to all its neighbors. The amortized cost for each label update of PUSH is constant, while PULL is $O(d(v))$. PU are 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.

iv) Scaling and Runtime Results: Figure 5 shows performance of the scaling runtime results of five parallel implementations (DOR, DOH, PUR, PUH and PLP) on real-world networks listed in Table 3. PLP sometimes fails to converge due to label swapping in multi-threading. We have to set its termination threshold to 0.001 in order to have a comparable result with DOLPA on all three graphs. Figure 6 summarizes the comparison between the algorithms for the different parallel settings in a performance profile plot [7]. We can see that DOLPA outperforms PLP in 75% of the settings with maximum speedup up to 10×.

6 RELATED WORK

Since LPA was first introduced for community detection, many other works that improve or extend it have been proposed. We list only some here. Xie et al. stabilize LPA by eliminating the need of tie breaking [36]. Other works alleviate the randomness of tie breaking with other node preference or edge preference instead of treating each node/edge with the same preference based on k-shell value [37], local cycle [40], or modularity [3, 23]. Leung et al. use hop weight to prevent the occurrence of a "monster" community [20]. Common measures for node preference include degree centrality and clustering coefficient [32]. 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 quality of solution produced by LPA can be improved by selecting influential nodes [41] or community kernels [21] in the pre-processing phase and then growing community structures from them. The algorithm can be made faster by maintaining all label information in memory instead of computing on the fly [8, 12]. But this approach require data synchronization for the label information and it's not scalable for large network processing. The state-of-the-art parallel LPA is the Parallel Label Propagation (PLP) algorithm implemented on multi-core architecture [34]. Other works parallelize LPA on multi-core [18], on GPU [16, 33], in Map-Reduce model [39] and in distributed memories [1]. Liu et al. [22] also combine LPA with direction optimization technique, but their work is implemented in distributed memory with active-message based runtime system.

Direction optimization has been applied in other graph algorithms, including PageRank[35], Betweenness Centrality[25], Connected Components [13] and Single Source Shortest Path [6] and in graph frameworks such as Polymer[38] and Ligra[31]. Besta et al. [5] study push-pull dichotomy in graph computations in terms of performance, speed of convergence and code complexity.

7 CONCLUSION

We presented a new Label Propagation algorithm, called Direction-optimizing Label Propagation Algorithm (DOLPA), and showed its efficacy as a method for community detection in networks. We introduced a new label update heuristic called PUSH, and abstracted the current 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. Using a carefully designed microbenchmark, we summarized the characteristics of PUSH and PULL, and 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-to-solution trade-off that our algorithm provides enables many community detection scenarios. For fast community detection, PUSH saves the time, but too many PUSH operations harm the precision. For accurate community detection, PULL is precise but costly.

ACKNOWLEDGMENTS

The research is supported by the NSF CAREER award IIS 1553528, and by the U.S. DOE ExaGraph project, a collaborative effort of U.S. DOE SC and NNSA at DOE PNNL. This research used computing resources at the Aeolus High Performance Computing cluster at college of VCEA at WSU. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

REFERENCES

[1] 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

[2] 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

[3] 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

[4] 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

[5] 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 (HPDC '17)*. ACM, 93–104. https://doi.org/10.1145/3078597.3078616

[6] 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 (July 2017), 2031–2045. https://doi.org/10.1109/TPDS.2016.2634535

[7] E. D. Dolan and J. J. Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical Programming* 91, 2 (Jan. 2002), 201–213. https://doi.org/10.1007/s101070100263

[8] 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://link.springer.com/chapter/10.1007/978-3-030-05411-3_14

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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 (HPDC '18)*. ACM, 92–104. https://doi.org/10.1145/3208040.3208041

[14] 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

[15] 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* (2014) *(KDD '14)*. ACM, 1366–1375. https://doi.org/10.1145/2623330.2623621

[16] 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 (CIKM '17)*. ACM, 567–576. https://doi.org/10.1145/3132847.3132960

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

[18] 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

[19] 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

[20] 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 (June 2009), 066107. https://doi.org/10.1103/PhysRevE.79.066107

[21] 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

[22] X. 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

[23] 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 (April 2010), 1493–1500. https://doi.org/10.1016/j.physa.2009.12.019

[24] 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

[25] 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*. 1–8. https://doi.org/10.1109/IPDPS.2009.5161100

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

[27] M. E. J. Newman and M. Girvan. 2004. Finding and evaluating community structure in networks. *Phys. Rev. E* 69 (Feb 2004), 026113. Issue 2. https://doi.org/10.1103/PhysRevE.69.026113

[28] 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 (Sept. 2007), 036106. https://doi.org/10.1103/PhysRevE.76.036106

[29] 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 (AAAI'15)*. AAAI Press, 4292–4293. http://dl.acm.org/citation.cfm?id=2888116.2888372

[30] 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

[31] J. Shun and G. E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (Feb. 2013), 135–146. https://doi.org/10.1145/2517327.2442530

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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 (May 2015), 1550029. https://doi.org/10.1142/S0217979215500290

[41] 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

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

A ARTIFACT DESCRIPTION

After downloading the compressed package from the Zenodo 10.5281/zenodo.3748897, one needs to extract our package. Our package has 5 parts: source code (src), unit test (test), tool, scripts, and Jupyter Notebooks for plotting figures (plot). Please refer to the Readme file in each part in the package.

A.1 Software Prerequisites

CMake >= 3.13
GCC >= 6
Anacoda (only for plotting using Jupyter Notebook)
bash
wget

A.2 Dataset Preparation

A.2.1 Downloading the GraphChallenge and Real-world Datasets. Download the GraphChallenge and real-world graphs using *scripts/data-prep/download.sh*. This script automatically downloads all 4 GraphChallenge graphs and the real-world graphs and extracts the graphs from the downloaded archives into a data directory.

A.2.2 Generating the LFR Graphs. Download the LFR-benchmark source code, compile and generate the LFR graphs using *scripts/data-prep/lfr-gen.sh*. This script generates 5 groups of LFR graphs: micro, small-0.1, small-0.3, big-0.1 and big-0.3. Each group has 10 graphs. (Note: it may take a long time to generate.)

A.3 Compiling and Running

A.3.1 How to compile. Microbenchmark experiments use Debug build for result collection. Strong scaling and quality of solution evaluation use Release build. Release build is the default build. To compile *Debug* build for microbenchmark, including *-DENABLE_DEBUG=ON*:

Listing 1: How to compile

```
$ cd dolpa
$ mkdir build
$ cd build
$ CC=/usr/bin/gcc CXX=/usr/bin/g++
cmake .. –DENABLE_DEBUG=ON
$ make
```

After compilation, the binary *LPA* and *DOLPA* will be generated in the *bin* directory within *build*. Also, the tool *clusterComp* is generated in the *bin* directory.

As the real-world data has no ground truth information, we use the fast-tracking resistance (FTR) method implementation in Grappolo to generate the ground truth communities of the real-world graphs. Use *evaluation/compile_louvain.sh* to download the source code of Grappolo and compile it. The script generates executable LV and executable FTR respectively.

A.3.2 How to run. Basic LPA/DOLPA running option:
-g, –graph-type $< 1 - 5 >$
1) - LFR Graph
2) - Matrix Market
3) - Metis/DIMACS#10
4) - SNAP
5) - Edge List
-i, –input-graph $< filename >$
-v, –groundtruth-community $< filename >$
-n, –thread-number $< value >$
-t, –termination-threshold $< value >$: large value will result early termination
-m, –max-iteration $< value >$: to assure convergence

Other DOLPA running options are related to the seeding parameter, switch threshold and two seeding strategies. The seeding parameter default value 0 ($|V|/|E|$) is the expected best solution/runtim combo provider.
-r, –random-seeding $< value >$
-h, –highdegree-seeding $< value >$
-s, –switch-threshold $< value >$: default value 0 is the expected best solution/runtim combo provider.

A.3.3 Microbenchmark. Now we run LPA/DOLPA for the microbenchmark. There are two scripts in *scripts/microbench*: *random.sh* and *random_per_iter.sh*. The results from the former script include runtime breakdown where the pre-process, push and pull step runtime can be collected. In the latter script, the number of edges visited (under metric **touch**) at each iteration is collected here, so is the F-Score at each iteration for different variants.

A.3.4 QoS Evaluation. After compiling the Release build, we run LPA/DOLPA for the quality-of-solution evaluation results using the scripts under *scripts/evaluation*. We divide the evaluation into 3 parts for 3 different groups of graphs. For real-world graph, before we evaluate on the real-world data, we generate the ground truth information of the real-world graphs using FTR method. This step is automatically done using *evaluation/realworld.sh*. *evaluation/realworld.sh* also generates the evaluation results on the real-world graphs.

A.3.5 Strong Scaling. Using Release build, we run strong scaling experiments using the scripts in *scripts/scaling*.

A.4 Plotting

After collecting raw data into comma-separated values (csv) files, we plotted the figures appearing in the paper. We provide the template of the csv files with respect to each notebook. To reproduce the figures in our paper, run the notebook with the same name of the csv files that are in the same directory. The notebook generates figure in both pdf and ps (used in our paper).