Distributed Direction-Optimizing Label Propagation for Community Detection

Xu Liu*†, Jesun Sahariar Firoz†, Marcin Zalewski†, Mahantesh Halappanavar†,
Kevin J. Barker†, Andrew Lumsdaine†§, Assefaw H. Gebremedhin*

* Washington State University, Pullman, WA, USA {xu.liu2, assefaw.gebremedhin}@wsu.edu
† Pacific Northwest National Laboratory, Richland, WA, USA {jesun.firoz,marcin.zalewski,
mahantesh.halappanavar,kevin.barker,andrew.lumsdaine}@pnnl.gov
§ University of Washington, Seattle, WA, USA

***Abstract*—Designing a scalable algorithm for community detection is challenging due to the simultaneous need for both high performance and quality of solution. We propose a new distributed algorithm for community detection based on a novel Label Propagation algorithm. The algorithm is inspired by the *direction optimization* technique in graph traversal algorithms, relies on the use of *frontiers*, and alternates between abstractions called label *push* and label *pull*. This organization creates flexibility and affords us with opportunities for balancing performance and quality of solution. We implement our algorithm in distributed memory with the active-message based asynchronous many-task runtime AM++. We experiment with two seeding strategies for the initial seeding stage, namely, random seeding and degree seeding. With the Graph Challenge dataset, our distributed implementation, in conjunction with the runtime support, detects the communities in graphs having 20 million vertices in less than one second while achieving reasonably high quality of solution.**

I. Introduction

Community detection deals with identifying groups of tightly-knit entities, known as *communities* (or *clusters*), within a given network [1]. Designing a scalable algorithm for community detection is challenging due to the need for both performance and quality of solution.

In response to the 2019 Graph Challenge [2], we propose a new Distributed Direction-optimizing Label Propagation (DDOLP) algorithm. DDOLP relies on the Label Propagation Algorithm (LPA) proposed by Raghavan et al. [3]. Label Propagation is a semi-supervised machine learning algorithm where information/label is propagated from the labeled data to the unlabeled data. In the context of community detection, in the standard LPA, each vertex is initialized with a unique label and at every iteration a vertex adopts a label that is the most common in its neighborhood. As the algorithm progresses, densely connected groups of nodes form a consensus on their labels, and the vertices that have the same labels are grouped as communities.

Another commonly adopted community detection approach is based on modularity maximization such as the Louvain method [4]. This approach fails to detect small clusters in a large network, a phenomenon referred to as *resolution limit*. In contrast, LPA holds the potential to be resolution-limit-free, since it only relies on the local graph structure information.

Other two attractions of LPA are that its runtime is nearly linear in the size of the network and that it requires no *a priori* information about the community structures. Meanwhile, LPA is known for producing unstable results due to *random* label selection in different runs. Also, a dominant label may "flood" large potion of the network resulting in a "monstrous" community [5]. Our proposed algorithm addresses both of these problems.

DDOLP builds a *frontier* queue in each iteration and, depending on the execution stage, *switches* between two execution modes, called a *pull* operation and a *push* operation, to propagate the labels of the vertices. The computational cost of label propagation can be reduced by adopting the *push* operation, while the quality of the solution can be improved by earlier propagation of the labels of important vertices that are the potential cores of the community structures. More algorithmic details are discussed in §III.

Since our algorithm involves remote memory accesses and irregular communication patterns, we implement it in an asynchronous many-task (AMT) runtime, AM++, based on active messages [6]. To facilitates graph computation, AM++ provides termination detection, message coalescing, message reduction and task-based scheduling.

In summary, we make the following **contributions** in this paper:

- We introduce a new distributed direction-optimization label propagation algorithm for community detection; DDOLP stabilizes LPA and provides a tradeoff between performance and quality of solution (§III).
- We present an implementation of DDOLP in an AMT runtime based on active messages (§III).
- We evaluate the performance, solution quality, and the scalability of our distributed implementation using the Graph Challenge dataset with ground truth (§IV).

II. Preliminaries

A. The Standard Label Propagation Algorithm

We outline the standard LPA in Alg. 1. The algorithm starts by assigning a unique label to every vertex in the graph (Ln. 2). In each iteration, neighbors of every vertex are examined to accumulate label counts. Once the counting is finished, the

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

label of the vertex is updated with the maximum label of its neighborhood (Lns. 4–7). A maximum label can be the most common label or the label associated with the highest weight. Alg. 1 terminates when the labels of all the vertices remain unchanged and outputs a group of communities, each including a set of vertices with the same label.

Complexity of LPA In the standard LPA, Lns. 5–6 take 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; specifically, the runtime is $O(km)$, where m is the number of edges and k is the number of iterations.

Algorithm 1: Standard Label Propagation Algorithm.

```
   In  : Graph G = ⟨V, E⟩
   Out : ∀ v ∈ V: label[v] = label of v
1  foreach v ∈ V do
2      label[v] ← a unique value in {0, 1, ··· , |V| − 1}
3  while ∃ any update on label[] do
4      foreach v ∈ V do
5          foreach neighbor w of v do
6              frequency[label[w]]++
7          label[v] ← arg max (frequency[label[w]])
```

B. Related Work

There is a large body of literature on parallelizing or improving LPA. Šubelj [7] provides an extensive survey of the state of LPA in community detection. The works MLPA [8] and MemLPA [9] propose variants where the label information is maintained in the memory and updated in each iteration instead of computing on the fly. Xing et al. [10] compute a node influence value using k-shell value and degree of the vertex and used this value for further maximum label assessment.

Beamer et al. [11] introduced Direction-optimizing for parallel Breadth-First Search, where they use a top-down procedure when the size of frontier is small and apply a bottom-up procedure when the size of frontier is large. Direction optimization has been applied in other graph algorithms. Besta et al. [12] study push-pull dichotomy in performance, speed of convergence and code complexity.

We have implemented our distributed direction-optimizing label propagation algorithm using the AM++ asynchronous, many-task distributed runtime system. AM++ [6] is based on active messages, where the sends are explicit but receives are implicit. A message handler is registered for each message type at the setup phase of the algorithm, which gets triggered each time a message with a certain type is received. Active messages have lower software overhead. In addition, AM++ provides with object-based addressing, message coalescing, caching and built-in termination detection. Such runtime support is helpful for the performance of irregular applications and for the productivity of the programmers.

III. Direction-optimizing Label Propagation

A. Motivations for Direction-optimizing

In designing the direction-optimizing LPA, we consider two operations on vertices, called *push* and *pull* operations.

Initially, each vertex is assigned a unique label. Within the first few iterations of LPA, small dense regions emerge as the cores of the communities. At later stages of the algorithm, these cores gain momentum and propagate their labels to nearby vertices. However, instead of passively forming community cores in the early iterations, we select a set of important vertices based on some criteria and broadcast these labels to their neighbors to form community cores. This kind of label update is referred to as a *push* operation.

On the other hand, in LPA, the labels of the neighbors of each vertex can be queried to obtain the "maximum" labels. The maximum label can be defined as the most frequently assigned label or the highest weighted label. We abstract such label update operation as a *pull* operation.

During both label update operations, we maintain a *frontier* of active vertices in a distributed queue. In the initialization step, pre-selected "seed" vertices are inserted into the frontier to jump start the core community formation. We broadcast the labels of the seed vertices to their neighbors with *push* operations in the first several iterations to form community cores faster. Later, we switch to *pull* operations for growing community cores to improve quality.

B. Direction-optimizing Label Propagation Algorithm using Active Message

We present our Distributed Direction-optimizing Label Propagation algorithm using active messages in Alg. 2. The PUSH operation is listed at Ln. 22 and the PULL operation at Ln. 27.

Lns. 1–2 constitute the pre-processing steps. The step to assign a unique label to each vertex in the graph is omitted here. Two seeding strategies can be chosen separately to enqueue a fraction τ of vertices to the initial frontier: random selection or a set of high-degree vertices. τ is called a seeding parameter passed as a part of the input. The *pullswitch* is initially set to $False$. The algorithm maintains two frontier queues: *Frontier* for processing the current active vertices and *nextFrontier* to store vertices that will be processed in the next iteration.

The direction optimization is triggered at Ln. 4 by a switch threshold ω that is passed as a part of the input. Specifically, from 1st to $(\omega-1)$-th iteration, when *pullswitch* is $False$, labels are propagated using PUSH operations. From ω-th iteration to convergence, label updates are done in PULL operations. *pullswitch* will be set to $True$ at the beginning of ω-th iteration. Notice that if ω is set to one, no PUSH operation will be applied therefore no direction optimization will happen in this case.

In each iteration, each rank propagates the label of each vertex in the frontier using either PUSH or PULL. During PUSH, the label of vertex v is sent to each of its neighbor w as an active message (Ln. 26). Once a message is received by the owner rank of w, a pre-registered message handler $relax$ (Ln. 14) is

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

Algorithm 2: Distributed Direction-optimizing Label Propagation Algorithm using Active Message.

```
   In  : Graph G = ⟨V, E⟩, seeding parameter τ, switch threshold
         ω, ∀ v ∈ V: owner[v] = rank that owns v
   Out : ∀ v ∈ V: label[v] = label of v
1  nextFrontier ← ∅, pullswitch ← False, iteration ← 0
2  enqueue fraction τ of {v ∈ V} → Frontier
3  while Frontier not empty do
4      if pullswitch = False and ω = iteration++ then
5          pullswitch ← True                          ▷ Enter only once
6      active message epoch
7          parallel foreach (v) ∈ Frontier do
8              if pullswitch = False then
9                  push(v)
10             else
11                 pull(v)
12     Frontier ← nextFrontier
13     nextFrontier ← ∅
14 message handler relax(Vertex w, Label newlabel)
15     while newlabel ! =label[w] do
16         oldlabel ← label[w]
17         atomic
18             success ← CAS(label[w], oldlabel, newlabel)
19         if success then
20             parallel foreach neighbor u of w do
21                 enqueue (u) → nextFrontier
22 procedure push(Vertex v)
23     labelv ← label[v]
24     parallel foreach neighbor w of v do
25         m ← buildMessage(w, labelv)
26         send m to owner(w)
27 procedure pull(Vertex v)
28     parallel foreach neighbor w of v do
29         receivedLabel ← get label[w] from owner(w)
30         frequency[receivedLabel]++
31     newlabel ← arg max (frequency[receivedLabel])
32     if newlabel ! =label[v] then
33         label[v] ← newlabel
34         parallel foreach neighbor w of v do
35             enqueue (w) → nextFrontier
```

executed to process the message. A label update is performed atomically with a compare-and-swap (CAS) operation. If the update is successful, the neighbors of w are added to the next frontier. Similarly in PULL, the labels of the neighbors w of vertex v are fetched from the owner rank of w (Ln. 29). Next, the label of v is updated locally based on the maximum label among v's neighbors.

The algorithm terminates when the next frontier is empty (Ln. 3). On termination, Alg. 2 associates a label with each vertex in the graph. Vertices with the same label are members of the same community.

Complexity of DDOLP. Both PUSH and PULL operations take O($d(v)$) time, where $d(v)$ is the degree of vertex v. The worst case performance of DDOLP happens when the frontier holds all of the vertices in the graph, and the space cost of the frontier is O(n). The runtime of DDOLP is bounded by O(km), where k is the number of iterations. Hence, DDOLP is nearly linear, just as the standard LPA.

C. Discussion around DDOLP

In this subsection, we discuss several design issues around DDOLP implementation and our approaches for addressing them. The discussion includes selection of seed vertices, comparison between PUSH and PULL operations, hyper-parameter study, frontier expansion and label swapping.

1) Seeding Strategies: An influential or seed vertex is the potential core of a community structure in the network. Nodes with high degrees, high clustering coefficient, fully-connected cliques, and maximal cliques are usually treated as the seed of a community [13]. In general, a node with better connectivity is more important in the network [10]. It has been proven that the membership contribution of a vertex to a community is highly related to its degree [14]. Populating the initial frontier with the seed vertices raises the chance of the labels of the seed vertices to be propagated earlier than the other labels to form the cores of the communities in the first several iterations.

We provide two seeding strategies in DDOLP: random seeding and degree seeding. It has been observed that better performance can be achieved if a large fraction of the seed vertices are connected to other vertices within the same community [15]. This suggests that it is imperative to choose a seed vertex that has good internal connectivity within the community relative to the rest of the graph. By sorting the degrees of local vertices on each rank, we select a fraction of the highest-degree vertices as seeds. We denote this as the *degree seeding* strategy. To compare the performance of degree-seeding strategy, we also randomly select vertices as seed vertices. We refer this as the *random seeding* strategy.

2) Label PUSH *versus Label* PULL*:* We examine and compare the parallel behaviors of PUSH and PULL operations. PUSH works well for quickly propagating label information (assuming initial seeds are good) by pushing active changes. However, PUSH does not directly take the neighbors' labels into consideration. In this regard, PULL works well in the later stages of the execution of the algorithm. At the very beginning, we use PUSH to formulate the communities faster and then switch to PULL after some iterations, so that the quality of labels of the neighbors are also taken into consideration when choosing a label.

If multiple vertices execute PUSH concurrently, they access their neighbors in parallel. Because of the shared neighbors, these vertices will contend for updating the labels of their common neighbors. This causes a benign race [16] that does not influence the correctness of the algorithm but can hurt performance. Benign race entails repeated work and makes the algorithm non-deterministic. PULL does not cause benign race.

In comparison with PUSH, PULL is a more expensive label update operation. PULL updates one label per operation but triggers $d(v)$ active messages, where $d(v)$ is the degree of vertex v. In contrast, PUSH propagates $d(v)$ labels per operation, hence one label per active message.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

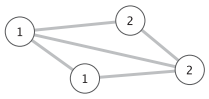

Fig. 1: An illustration of swapping labels between the upper left vertex and the lower right vertex.

3) Hyper-parameters Study and Frontier Expansion: There are two hyper-parameters in DDOLP: *seeding parameter* τ and *switching threshold* ω. $\tau \in [0, 1]$ controls the number of seed vertices to enqueue in the initial frontier. $\omega \in [1, k]$ chooses the iteration for the direction of label propagation to reverse, where k is the number of iterations.

DDOLP propagates the labels of the seed vertices at least one iteration earlier than the other vertices. These labels have a better chance to form community cores. If the seed vertices are truly influential, earlier propagation of those labels will improve the quality of the solution. In addition, the frontier can enforce propagation order on vertices. The higher τ is, the more seed vertices will be added to the initial frontier. This will make the propagation order of the labels more deterministic therefore a more stable algorithm.

PUSH and PULL expand the frontier in a similar fashion. For each PUSH or PULL operation, it adds d neighbors of a vertex to the next frontier in each iteration, where d is the average degree of the graph. With $n * \tau$ seed vertices in the initial frontier, the frontier expansion rate is $d * n * \tau = m * \tau$, where n is the number of vertices and m is the number of edges. The best performance of DDOLP can be achieved when the frontier holds exactly m edges therefore $\tau = 1/d$. If DDOLP adopts degree seeding strategy, the selected seed vertices are highest-degree vertices, where their degrees are larger than d. In this scenario, the frontier expansion rate under degree seeding strategy will be larger than $m * \tau$ and the best performance is attained when τ is less than $1/d$.

To avoid duplicate entries in frontier expansion, in DDOLP, we use a bitmap. A set bit for a vertex indicates an existing insertion in the next frontier, hence any insertion attempt for the same vertex in the current iteration is aborted.

4) Local Maxima and Label Swapping: Fig. 1 illustrates a scenario where two vertices reach a local maxima and swap labels in parallel. The upper left vertex finds the most common label 2 in its neighborhood. The lower right vertex finds the most common label 1 in its neighborhood. The upper left vertex and the lower right vertex keep swapping their labels in each iteration. This is called *label oscillation* [3]. LPA may fail to converge due to label swapping. To ensure convergence, our algorithm takes a threshold value as a parameter. If the amount of labels updated in the previous iteration is less than this threshold value, the algorithm stops.

IV. Experimental Results

We evaluate the quality of solution on a cluster of 16 compute nodes. Each node is equipped with a single Intel Xeon Ivy Bridge E5-2680v2 CPU with 10 physical cores at 2.80GHz. Each node has 128GB main memory and is connected over an Infiniband network. We use OpenMPI 3.1.3 and GCC 8.2.0 compiler with "-O0" as compilation option. The performance is evaluated on a Cray XC50 system of 120 compute nodes, which we only scale up to 64 nodes. Each node is equipped with dual Intel Xeon Broadwell CPU with 44 cores at 2.2GHz and 128GB DDR4 main memory. We use Cray MPICH 7.7.8 and GCC 8.3.0 compiler with "-O3" as compilation option. Tab. I lists the four variants of DDOLP (PU and DO with two seeding strategies). The datasets are all downloaded from the 2019 Graph Challenge website (http://graphchallenge.mit.edu/data-sets). The *low* block overlap and *low* block size variation static graphs are referred as LOLO for short in the rest of the section. Similarly, LOHI, HILO and HIHI are used for other three groups of static graphs. We compute precision, recall and f-score based on the formula in [17].

TABLE I: Four variants of DDOLP studied under two seeding strategies and with or without the usage of Direction-optimizing

Version	Applied Direction-optimizing	Seeding Strategy
PUR	No, PULL only	*Random seeding*
PUD	No, PULL only	*Degree seeding*
DOR	Yes	*Random seeding*
DOD	Yes	*Degree seeding*

A. Quality of Solution

To understand how the seeding parameter τ affects the quality of solution in practice, we perform a hyper-parameter study on DDOLP using PULL only — PUR and PUD, with two seeding strategies. τ ranges from $[0.2, 0.4, 0.6, 0.8, 1]$. The experiments are performed on all four groups of small-medium graphs.

Horizontally, Fig. 2 shows that precision, recall and f-score increase as τ increases. Vertically, Fig. 2 shows how different two seeding strategies affect the quality of solution on the same graph. All four instances under random seeding strategy have lower precision, recall and f-score than instances under degree seeding strategy. This empirically shows that the degree seeding strategy contributes positively to the quality of solution. The high degree vertices we enqueue are more likely to be a true seed vertices than a random vertex. For each instance, irrespective of the seeding strategy, the higher τ is, the more accurate the solutions are. This proves the propagation orders that the frontier enforces improve the quality of the solution.

Tab. II summarizes the best f-scores on the four different groups of small-medium static graphs with ground truth partitions. The table is sorted such that the f-scores increase from the left to the right (PUR, DOD, DOR and PUD).

Most of the highest f-scores are obtained by PUD when $\tau = 1$. In comparison with PUR, the degree seeding strategy does contribute lower false negative number, hence higher recall. On the other hand, most of the lowest f-scores are obtained by PUR. By selecting random vertices as seed vertices, both the precision and recall are lower comparing with degree seeding strategy. This again confirms the true seed vertices improve the quality of solution.

For DOD, there is a clear drop of precision and f-score drop on HIHI and LOHI when the size of the graph increases. Both HIHI and LOHI have high block size variation. As the size of the graphs increases, the block size variation becomes even higher. The quality of solution of DDOLP under degree seeding strategy is heavily influenced by the block size changes. This is because the high degree vertices enqueue to the initial

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

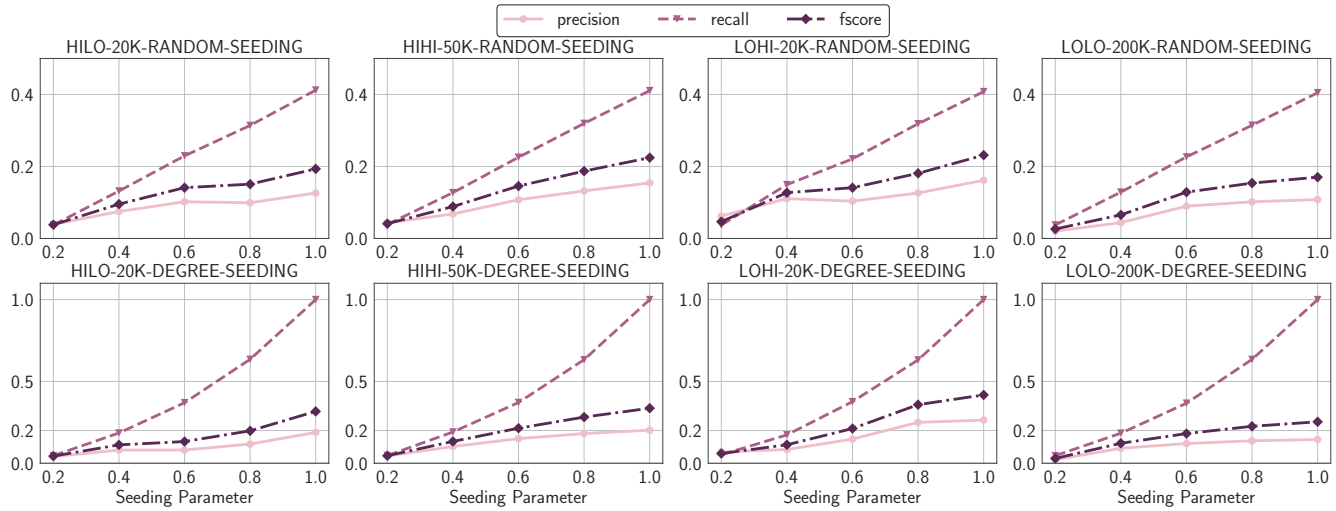

Fig. 2: Precision, recall and f-score results under two seeding strategies on four graphs with PULL only DDOLP when τ increases from $[0.2, 0.4, 0.6, 0.8, 1]$. The top four instances adopt random seeding strategy, while the bottom four instances use degree seeding strategy. $\tau = 0.2$ means to add 20% of the vertices as seed vertices to the initial frontier.

TABLE II: Best precision, recall and f-score results on four groups of graphs with ground truth partitions. The best f-scores are marked in bold. For both PUR and PUD, the f-scores are achieved under seeding parameter $\tau = 1$.

			PUR			DOD			DOR			PUD		
Input	|V|	|E|	prec.	recall	f-score	prec.	recall	f-score	prec.	recall	f-score	prec.	recall	f-score
HIHI	1k	8k	19.15%	40.94%	26.09%	25.46%	49.68%	33.66%	23.97%	52.96%	33.00%	22.81%	99.92%	**37.14%**
HIHI	5k	51k	29.38%	43.08%	34.93%	30.16%	43.14%	35.50%	16.42%	58.90%	25.69%	36.56%	100%	**53.54%**
HIHI	20K	473k	12.80%	40.53%	19.45%	21.03%	53.87%	30.25%	17.90%	90.88%	29.91%	18.12%	100%	**30.68%**
HIHI	50K	1.2M	15.44%	41.06%	22.44%	6.83%	57.82%	12.21%	19.88%	91.89%	32.69%	20.21%	100%	**33.63%**
HIHI	200K	4.8M	14.12%	40.78%	20.97%	4.98%	75.61%	9.35%	16.72%	94.37%	**28.41%**	16.46%	100%	28.27%
HILO	1k	8k	16.77%	39.03%	23.46%	21.36%	61.04%	31.64%	19.60%	54.99%	28.90%	21.39%	100%	**35.24%**
HILO	5k	51k	16.28%	40.97%	23.30%	19.14%	65.99%	29.68%	25.13%	70.61%	**37.07%**	15.97%	100%	27.54%
HILO	20K	475k	12.62%	41.20%	19.32%	17.75%	77.98%	28.91%	18.82%	96.27%	31.49%	18.83%	100%	**31.69%**
HILO	50K	1.2M	11.18%	40.76%	17.54%	13.89%	89.59%	24.05%	13.86%	97.00%	24.25%	14.08%	100%	**24.69%**
HILO	200K	4.8M	12.77%	40.55%	19.43%	14.10%	83.61%	24.14%	14.43%	96.70%	**25.11%**	14.07%	100%	24.67%
LOHI	1k	8k	25.65%	43.97%	32.40%	20.89%	25.27%	22.87%	22.10%	49.36%	30.53%	31.12%	100%	**47.47%**
LOHI	5k	51k	18.11%	41.40%	25.20%	13.24%	28.59%	18.10%	18.01%	62.57%	27.97%	30.54%	100%	**46.79%**
LOHI	20K	476k	16.15%	40.78%	23.14%	7.24%	26.74%	11.39%	12.31%	85.39%	21.52%	26.36%	100%	**41.73%**
LOHI	50K	1.2M	14.35%	40.65%	21.21%	5.64%	41.43%	9.94%	17.68%	91.55%	29.64%	17.66%	100%	**30.02%**
LOHI	200K	4.7M	12.77%	40.55%	19.43%	3.89%	36.45%	7.03%	18.41%	89.14%	**30.52%**	14.07%	100%	24.67%
LOLO	1k	8k	17.70%	41.78%	24.87%	20.36%	35.17%	25.79%	23.80%	54.76%	33.18%	20.60%	100%	**34.17%**
LOLO	5k	51k	20.53%	41.77%	27.53%	17.17%	62.15%	26.91%	14.22%	70.61%	23.67%	19.15%	100%	**32.15%**
LOLO	20K	473k	13.89%	41.04%	20.76%	17.13%	60.69%	26.71%	13.79%	96.28%	24.12%	18.37%	100%	**31.04%**
LOLO	50K	1.2M	10.63%	40.65%	16.86%	17.17%	62.15%	**26.91%**	14.22%	70.61%	23.67%	14.21%	100%	24.89%
LOLO	200K	4.8M	10.79%	40.48%	17.03%	12.93%	56.80%	21.07%	13.37%	96.03%	23.48%	14.50%	100%	**25.33%**

TABLE III: Best performance results of PUR and PUD under 16 processes and 20 threads per node. Best performance is achieved at the seeding parameter $\tau = 1/d = 20/475 \approx 0.0421$.

			PUR		PUD	
Input	|V|	|E|	τ	time(s)	τ	time(s)
HIHI	20M	475M	0.0421	0.8337	0.0421	4.7423
HILO	20M	475M	0.0421	0.9103	0.0421	3.6336
LOHI	20M	475M	0.0421	0.9058	0.0421	5.3491
LOLO	20M	475M	0.0421	2.9041	0.0421	7.8571

frontier are false influential seed vertices. These vertices are most likely on the border of community structure instead of the center of a community.

B. Performance

To understand how seeding parameter τ affects the performance, we perform a similar hyper-parameter study on DDOLP using PULL only — PUR and PUD, with two seeding strategies. τ ranges from $[1, 0.8, 0.6, 0.4, 0.2, 0.042]$, where $\tau = 0.042$ is the seeding parameter we predicted for best performance on graphs having 20 million vertices. The experiments are performed on four groups of 20 million vertices graph.

Fig. 3 reveals that the execution time drops as τ drops. Table III summarizes the best execution time of PUR and PUD for the largest official dataset. PUR runs less than one second with the largest dataset (20M vertices) provided. Fig. 4 shows the scalability of DDOLP using PULL only (PUR) and DDOLP using Direction-optimizing (DOR) under random seeding strategy up to 64 nodes.

Based on the hyper-parameter study results of the quality of the solution (Fig. 2) and the performance (Fig. 3), we observe that the best quality of the solution can be attained when $\tau = 1$

978-1-7281-5020-8/19/$31.00 ©2019 IEEE

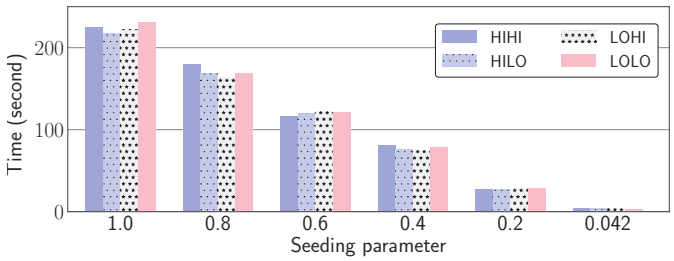

Fig. 3: Execution time results on four graphs with 20 million vertices with PULL only DDOLP under random seeding strategy (PUR) when τ decreases from $[1, 0.8, 0.6, 0.4, 0.2, 0.042]$. The performance is achieved with 8 nodes and 44 threads per node. The execution time drops when τ decreases. The same trend is observed with PUD.

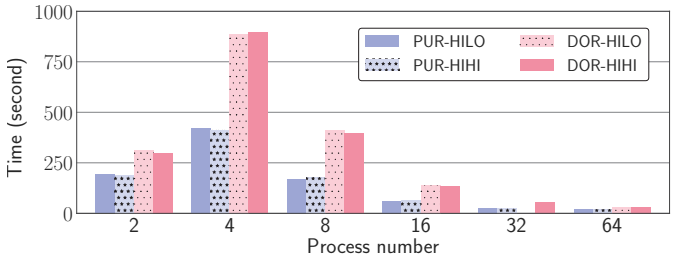

Fig. 4: Strong scaling performance of PUR on HILO 20M and HIHI 20M graphs when $\tau = 0.8$, and strong scaling performance of DOR on HILO 20M and HIHI 20M graph when $\tau = 0.042$. The performance is achieved with 44 threads per node.

but have the longest execution time. On the other hand, the best performance is achieved when $\tau = 1/d$ but results in the worst quality of the solution. Hence, DDOLP provides the flexibility to control the tradeoff between the performance and the quality of the solution by adjusting the combination of the seeding parameter τ, the seeding strategies and the switch threshold ω.

V. Summary and Future Work

We presented a new Distributed Direction-optimizing Label Propagation algorithm (DDOLP) for community detection and its implementation using active messages. We evaluated the quality of the solution and the performance with DDOLP using the 2019 Graph Challenge dataset with ground truth partitions. With hyper-parameter study results, We demonstrated our algorithm provided a tradeoff between the quality of the solution and the performance.

DDOLP can be extended to community detection on dynamics graphs. In addition, formulating better seeding strategies to improve the quality of the solution is also of interest.

Acknowledgments

The research is in part supported by the NSF award IIS 1553528 and by the U.S. DOE ExaGraph project, a collaborative effort of U.S. DOE SC and NNSA at DOE PNNL, and by the Defense Advanced Research Projects Agency's (DARPA) Hierarchical Identify Verify Exploit Program (HIVE) at DOE PNNL. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

References

[1] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, pp. 75–174, Feb. 2010.

[2] 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, "Streaming graph challenge: Stochastic block partition," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–12, Sep. 2017.

[3] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Phys. Rev. E*, vol. 76, p. 036106, Sept. 2007.

[4] V. D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[5] I. X. Y. Leung, P. Hui, P. Liò, and J. Crowcroft, "Towards real-time community detection in large networks," *Phys. Rev. E*, vol. 79, p. 066107, June 2009.

[6] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Am++: A generalized active message framework," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pp. 401–410, ACM, 2010.

[7] L. Šubelj, "Label propagation for clustering," *arXiv*, 2017.

[8] R. Hosseini and R. Azmi, "Memory-based label propagation algorithm for community detection in social networks," in *2015 The International Symposium on Artificial Intelligence and Signal Processing (AISP)*, pp. 256–260, March 2015.

[9] A. M. Fiscarelli, M. R. Brust, G. Danoy, and P. Bouvry, "A memory-based label propagation algorithm for community detection," in *Complex Networks and Their Applications VII*, Studies in Computational Intelligence, pp. 171–182, Springer International Publishing, 2019.

[10] Y. Xing, F. Meng, Y. Zhou, M. Zhu, M. Shi, and G. Sun, "A node influence based label propagation algorithm for community detection in networks," *The Scientific World Journal*, vol. 2014, 2014.

[11] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[12] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "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, pp. 93–104, ACM, 2017.

[13] Y. Zhao, S. Li, and F. Jin, "Identification of influential nodes in social networks with community structure based on label propagation," *Neurocomputing*, vol. 210, pp. 34–44, 2016.

[14] J. Scripps, P. Tan, and A. Esfahanian, "Exploration of link structure and community-based node roles in network analysis," in *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pp. 649–654, Oct 2007.

[15] I. M. Kloumann and J. M. Kleinberg, "Community membership identification from small seed sets," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pp. 1366–1375, ACM, 2014.

[16] C. E. Leiserson and T. B. Schardl, "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*, SPAA '10, pp. 303–314, ACM, 2010.

[17] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *Parallel Computing*, vol. 47, pp. 19 – 37, 2015.

978-1-7281-5020-8/19/$31.00 ©2019 IEEE