A Distributed Infomap Algorithm for Scalable and High-Quality Community Detection

Jianping Zeng
University of Nebraska-Lincoln
jizeng@cse.unl.edu

Hongfeng Yu
University of Nebraska-Lincoln
yu@cse.unl.edu

ABSTRACT

Community detection is essential to various graph analysis applications. Infomap is a graph clustering algorithm capable of achieving high-quality communities. However, it remains a very challenging problem to effectively apply Infomap on large graphs. By analyzing communication and workload patterns of Infomap and leveraging a distributed delegate partitioning and distribution method, we develop a new heuristic strategy to carefully coordinate the community constitution from the vertices of a graph in a distributed environment, and achieve the convergence of the distributed clustering algorithm. We have implemented our optimized algorithm using MPI (Message Passing Interface), which can be easily employed or extended to massively distributed computing systems. We analyze the correctness of our algorithm, and conduct an intensive experimental study to investigate the communication and computation cost of our distributed algorithm, which has not shown in previous work. The results demonstrate the scalability and the correctness of our distributed Infomap algorithm with large-scale real-world datasets.

CCS CONCEPTS

• **Mathematics of computing → Graph algorithms**; • **Theory of computation → Distributed algorithms**;

KEYWORDS

Community detection; Infomap; scalability; accuracy; large graphs

ACM Reference Format:
Jianping Zeng and Hongfeng Yu. 2018. A Distributed Infomap Algorithm for Scalable and High-Quality Community Detection. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3225058.3225137

1 INTRODUCTION

Finding community structures in graphs is a fundamental operation in various domain applications. Examples include identifying communities in social networks [13] and research collaboration networks [12], unsupervised learning [19], and optimizing graph traversal [2]. Many *community detection* (also named *graph clustering*) algorithms have been developed to classify vertices of a graph into different sets, where vertices have dense intra-connections within a set, but sparse inter-connections between sets [11].

Infomap [22] is a community detection algorithm capable of achieving high-quality communities [5]. However, the sequential Infomap algorithm is less capable to process large graphs in a scalable manner compared with other community detection algorithms, such as the Louvain algorithm [7]. Researchers make great efforts to parallelize the Infomap algorithm to tackle large graphs. For example, Bae *et al.* proposed different parallel methods for accelerating Infomap [4, 5]. However, they only showed a comparably limited scalability with up to 128 parallel processing units. Besides, the running time was also considerably long for large graphs, reported as near 2500 seconds for UK-2007 [8] (a graph with about 3.78 billion edges) using 128 cores. Such a performance degradation is mainly due to *hubs* (i.e., *high-degree vertices*) that commonly exist in large scale-free graphs generated from real-world applications. The existence of hubs makes it challenging to balance workload and communication among processors when tackling community detection in a distributed environment. This scalability issue has not been successfully addressed in the existing work.

In this paper, we present a new scalable and high-quality Infomap algorithm in a distributed environment, where the computation and communication costs associated with the hubs are effectively balanced. Meanwhile, our new algorithm can ensure the high quality of the clustering results. More specifically, we make the following four main contributions:

First, we investigate and address the challenging workload imbalance problem associated with large scale-free graphs by exploiting a graph partitioning method with vertex delegates used in distributed graph traversal algorithms [20]. Through a careful duplication of hubs among processors, our method can ensure each processor to handle a similar number of edges, and balance workload and communication among processors, which is neglected in previous work [3, 5]. Therefore, our approach can achieve optimal performance in distributed community detection.

Second, we design a novel distributed Infomap algorithm. We adopt a synchronized strategy in our distributed algorithm, and carefully swap the updated community information of hubs and low-degree vertices, which can effectively make the community information on each processor consistent. Moreover, we use a unique heuristic strategy to avoid the vertex bouncing problem that causes non-convergence in community detection. Therefore, our approach can ensure the accuracy of distributed community detection.

Third, instead of using the open source graph processing framework (e.g., GraphLab [14] and PowerGraph [14]), we investigate

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225137

the workload model of the Infomap algorithm in a distributed environment and provide a distributed implementation directly based on MPI. According to the recent work [23], the GraphLab based implementations do not show a strong scalability with very large real-world datasets. Moreover, these implementations cannot be extended to large-scale machines. Our implementation can effectively overcome these drawbacks.

Our extensive experiments show that our distributed algorithm is effective and correct. We also show the communication cost and the detailed time breakdown of different components in our algorithm, which were mostly ignored in the previous work. Our algorithm outperforms by around 6 times the recent work based on GraphLab. Our results have scaled up to 4,096 processors for distributed Infomap, and clearly showed an improved scalability over the previous state-of-the-art.

2 BACKGROUND AND MOTIVATION

Community detection has been extensively studied [11]. To efficiently and effectively tackle large graphs, one common strategy is to leverage multiple processing units to conduct community detection in a parallel or distributed fashion. In this section, we first review the related work on parallel and distributed community detection algorithms, and show the needs to parallelize the Infomap algorithm. Then, we revisit the mechanism of Infomap, and discuss the fundamental research challenges in designing a scalable distributed Infomap algorithm.

2.1 Related Work

Among different community detection algorithms, the modularity-based algorithms (particularly, the Louvain algorithm [7]) have been successfully parallelized in different ways. Bhowmick *et al.* [6] proposed an OpenMP implementation of the Louvain algorithm that adopted a lock mechanism. Staudt *et al.* [24, 25] proposed a parallelization approach by assembling the Louvain algorithm and the label propagation algorithm. Lu *et al.* [17] took advantage of the coloring algorithm [15, 16] as a preprocessing step for a parallel Louvain algorithm. Cheong *et al.* [10] presented a GPU-based Louvain algorithm based on the Divide-and-Conquer strategy. Naim *et al.* [18] presented a highly efficient Louvain algorithm on a single GPU, where each thread is responsible for one edge to reduce uneven workload caused by nodes of highly varying degrees. Wickramaarachchi *et al.* [26] proposed an MPI implementation of distributed Louvain algorithm for graphs with about 10 million edges. Zeng *et al.* [29, 30] presented a parallel modularity-based algorithm and showed its scalability with over 10,000 cores.

The sequential Infomap algorithm [22] can achieve high-quality communities based on random walks through a graph [5], but only can process comparably small graph sizes. The parallelization efforts for Infomap only achieved marginal scalability. Bae *et al.* [3] proposed RelaxMap, a parallelization of Infomap in a shared-memory multi-core environment, which can achieve a high-quality output similar to the sequential Infomap algorithm. However, this shared-memory implementation does not show its capability of processing a very large graph. Later, Bae *et al.* proposed a prioritization method [4] to reduce the total running time. GossipMap [5] is a distributed algorithm of Infomap based on an open source graph processing framework GraphLab [14]. However, its scalability was shown only on 128 cores, and its running time was notably high for the large real-world datasets. Thus, only very limited scalability has been demonstrated by the existing shared-memory and distributed parallel Infomap algorithms. The existing methods have not fully solved the fundamental challenges (Section 2.3) in designing scalable distributed Infomap algorithms. In this paper, we investigate these challenges, and develop an optimized scalable Infomap algorithm.

2.2 Infomap Algorithm

In a *graph* $G = (V, E)$, V is the set of vertices and E is the set of edges. The weight of an edge between two vertices, u and v, is denoted as $w_{u,v}$, which is 1 in an undirected unweighted graph. The community detection problem in Infomap is to find vertices sets, named communities (or modules), which contain high intra-module information flow but low inter-module flow. We denote the non-overlapping community set C of a graph G as:

$$\cup c_i = V, \forall c_i \in C; c_i \cap c_j = \emptyset, \forall c_i, c_j \in C \tag{1}$$

For simplicity, we only consider undirected unweighted graphs. The original Infomap work [22] shows that the Infomap algorithm can be applied on both undirected and directed graphs. Therefore, our work can be easily extended to directed graphs.

The *map equation* [22] is the objective function of the Infomap algorithm. It is based on the information flow, and is used to find a compressed representation of a set of random walks through a graph. Its insight is that a succinct representation of a graph walk can be expressed over clusters rather than individual vertices, and the clustering that produces the shortest representation also produces the community detection result of the highest quality in practice. Infomap uses *minimum description length* (*MDL*) to measure the quality of detected community results, where a shorter MDL means a more compressed community structure.

The definition of map equation is shown in Equation 2:

$$L(M) = q_{\curvearrowright} H(\boldsymbol{Q}) + \sum_{m \in M} p^m_{\circlearrowright} H(\boldsymbol{\mathcal{P}}^m), \tag{2}$$

where M is the set of modules, $q_{\curvearrowright}$ is the sum of the *exit probability* of each module in the graph, $H(\boldsymbol{Q})$ is the average code length of movements between the modules, $p_{\circlearrowright}$ is the *stay probability* for the random walks in a module m, which is equal to the sum of the *exit probability* and the *visit probability* of the random walks, and $H(\boldsymbol{\mathcal{P}}^m)$ is the average code length of a module codebook for m. $L(M)$ represents the lower bound on the code length of detected community structure M.

The map equation can be expanded as in Equation 3:

$$\begin{aligned} L(M) = \; & (\sum_{m \in M} q_m) log(\sum_{m \in M} q_m) \\ & -2 \sum_{m \in M} q_m log(q_m) - \sum_{\alpha \in V} p_\alpha log(p_\alpha) \\ & + \sum_{m \in M} (q_m + \sum_{\alpha \in m} p_\alpha) log(q_m + \sum_{\alpha \in m} p_\alpha), \end{aligned} \tag{3}$$

where $q_m$is the exit probability of a module m, p_α is the visit probability of a vertex α during the random walk, and V is the set of vertices in the graph. For an undirected graph, p_α corresponds to the relative weight w_α that is computed as the total weight of the links connected to the vertex α divided by twice the total weight

of all links in the graph (self-connected edges excluded). The visit probability of a module m, described as p_m, is the relative weight of m, calculated as $\sum_{\alpha \in m} p_\alpha$. The *exit probability* of m, q_m, is defined by the relative weight of links exiting the module q. $\sum_{m \in M} q_m$ is the total relative weight of the links between modules.

Algorithm 1 shows the sequential Infomap algorithm conducted in an iterative fashion. In each iteration, the algorithm greedily minimizes the MDL change δL ($\delta L < 0$) (i.e., maximizes the MDL decrease) to minimize the MDL when moving an isolated vertex u into a module m, which is based on Equation 3. The algorithm continues this process until there is no more vertex movement that can make the MDL increase. Then, the algorithm treats each community as one vertex to form a new graph and continues the above process. The algorithm stops when communities become stable (Line 31).

In Algorithm 1, G^k is the graph in the kth iteration and M_u^k is the community of a vertex u in G^k. The output of the algorithm is a module set M of each vertex and the MDL L of the detected community. There are three main phases in the algorithm:

- Phase 1: The visit probability of each vertex u is computed in terms of the total relative weight of the links connected to u, described as Line 3.
- Phase 2: The algorithm follows a hierarchical agglomerative clustering strategy where initially each vertex is regarded as a unique module, described as Lines 7 to 11. The algorithm initializes the visit probability of each module using the visit probability of each vertex (Line 9), and initializes the exit probability of each module using the links connected to each vertex (Line 10). In Lines 15 to 23, the algorithm calculates each MDL gain by moving a vertex u from its original module M_u^k to each neighbor module, and determines the movement of u to the module $M_u'^k$ to achieve the minimum MDL change δL. If there is no more vertex movement, the algorithm exits the loop of estimating δL of each vertex. In Line 25, the algorithm calculates the new MDL length using the updated $p^{M_u^k}$ and $q^{M_u^k}$.
- Phase 3: The algorithm merges the current communities into a new graph in Lines 27 to 29. In the new graph G^{k+1}, each vertex in the vertex set V^{k+1} represents a community c within the current communities C^k, and each edge in the edge set E^{k+1} represents all the edges connecting the communities M_u^k and M_v^k. The algorithm continues until the communities become stable (i.e., the MDL change is less than a predefined threshold θ) or the iteration number reaches a user-specified maximum number.

As we can see, Algorithm 1 mainly contains two levels of iterations: The inner iteration is from Lines 15 to 23 to calculate the minimal MDL. The outer iteration is from Lines 5 to 31, containing four steps, community initialization (Lines 7-11), minimal MDL calculation (Lines 15-23), MDL update (Line 25), and community merging (Lines 27-29). Each outer iteration can have multiple inner iterations.

Algorithm 1 Sequential Infomap Algorithm

Require:
$G^0 = (V^0, E^0)$: initial undirected graph, where V^0 is vertex set and E^0 is the edge set;
M^0: initial community of G^0;
θ: quality improvement threshold;
$max_{iteration}$: maximum iteration number.
Ensure:
M: resulting module;
L: resulting MDL;
δL: change of MDL.

1: $k = 0$ // k indicates the iteration number
2: **for all** $u \in V^0$ **do**
3: $p_u = degree(u)/|E|$
4: **end for**
5: **repeat**
6: // Initialize communities
7: **for all** $u \in V^k$ **do**
8: $M_u^k = \{u\}$
9: $p^{M_u^k} = \sum_{\alpha \in M_u^k} p_\alpha$
10: $q^{M_u^k} = \sum w_{u,v}, (u,v) \in E^k, u \in C_u^k$ and $v \notin C_u^k$
11: **end for**
12: Compute $L = L(M)$ using Equation 3
13: Randomize the order of vertices
14: // Calculate the change of MDL δL
15: **repeat**
16: **for all** $u \in V^k$ **do**
17: **if** $M_u'^k = argmin(\delta L_{M_u^k \to M_u'^k}) < 0$ **then**
18: $M_u^k = M_u^k - \{u\}$; $M_u'^k = M_u'^k \cup \{u\}$
19: $p^{M_u^k} = \sum_{\alpha \in M_u^k} p_\alpha - p_u$;$p^{M_u'^k} = \sum_{\alpha \in M_u'^k} p_\alpha + p_u$
20: update $q^{M_u^k}$; update $q^{M_u'^k}$
21: **end if**
22: **end for**
23: **until** No more vertex movement
24: // Calculate updated MDL
25: Compute $L_{new} = L(M)$ using Equation 3
26: // Merge communities into a new graph
27: $V^{k+1} \leftarrow C^k$
28: $E^{k+1} \leftarrow e(C_u^k, C_v^k)$
29: $G^{k+1} = (V^{k+1}, E^{k+1})$
30: $k = k + 1$
31: **until** $k \le max_{iteration}$ **and** $L - L_{new} < \theta$

2.3 Research Challenges

Algorithm 1 is computationally intensive. For example, it can take a long time (e.g., several hours) for a large-scale graph (e.g., one with billions of edges) using a single processor. A common strategy for acceleration is to first partition and distribute the original graph among multiple processors, and then conduct computation in a distributed fashion.

However, it is non-trivial to achieve scalable community detection for large graphs generated from real-world applications. These graphs typically are scale-free graphs and follow the power-law degree distribution, where the majority of vertices have small degrees, while only a few vertices (or hubs) have extremely high degrees. The existence of hubs makes it challenging to achieve balanced workload and communication among processors. In Algorithm 1, we can

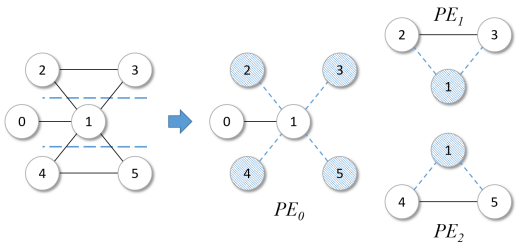

Figure 1: 1D partitioning of a graph on three processors, where a shaded vertex is a ghost vertex residing on a remote processor.

easily see that the complexity of the most intensive computational components, community initialization (Lines 7-11) and iterative MDL calculation (Lines 16-22), is proportional to the vertex number. The existing distributed graph clustering algorithms [10, 21, 29, 30] often employed simple 1D partitioning strategies, which prefer to put the entire adjacency list of a vertex into a single partition. If a processor is assigned with hubs, the workload and communication of this processor can be significantly higher than the other processors, which can inhibit the overall performance and scalability of distributed graph algorithms. Figure 1 shows an example to illustrate this problem. A simple graph, where the vertex 1 is a hub, is divided among three processors, PE_0, PE_1, and PE_2. A shaded vertex is a ghost vertex residing on a remote processor. The subgraph assigned to PE_0 contains the vertex 1 and has more edges, thus incurring more workload on PE_0. In addition, inter-processor communication is generally conducted through ghost vertices. There are more edges connecting the local vertices and the ghost vertices on PE_0, thereby incurring more communication between PE_0 and the other processors.

Moreover, it is very challenging to obtain accurate results in a distributed environment as each processor only has partial graph information. For example, in Lines 15 to 23 of Algorithm 1, it determines the movement of each vertex u to achieve the minimum δL among the neighbor modules of u. However, the neighbors of u may be located on different remote processors. Bae *et al.* [5] showed some relatively simple methods on how to move vertices in a distributed environment, such as assigning each vertex by a majority vote among its local neighbors (no remote information required), or moving vertices to the modules with maximum aggregate network flow. However, these methods only use local information of each processor and cannot match the output quality of the sequential Infomap algorithm. Alternatively, the information of boundary vertices (e.g., the community IDs of the boundary vertices) can be exchanged among the processors. For example, in Figure 1, in each iteration PE_0 sends the community information of the vertex 1 to PE_2, no matter whether the vertex 1 is in the same community with the vertices 4 and 5 or not. In this case, the vertex 3 on PE_1 still only has the partial community information of the vertex 1 and is not aware of its community information on PE_2. Therefore, we cannot achieve the quality of the sequential Infomap algorithm either. It is non-trivial to determine how information can be appropriately swapped among processors and how much information would suffice to achieve accurate results for a distributed Infomap algorithm.

3 OUR METHOD

3.1 Overview

We aim to address all these challenges analyzed in Section 2.3 in our distributed Infomap algorithm design. We first need to model the computation and communication costs of each processor in a distributed environment. Zeng *et al.* [29, 30] proposed an efficient model to estimate the workload of distributed Louvain algorithm using the edge number on each processor. Similar with the Infomap algorithm, in the Louvain algorithm each vertex needs to check all of its neighbors. Inspired by this work, we employ a graph partition strategy that assures each processor has a similar number of edges.

For designing scalable distributed community detection for large scale-free graphs, we also need to carefully consider hub vertices, as they can easily incur imbalanced workload and communication among processors. In a recent work conducted by Pearce *et al.* [20], they used a delegate partitioning method to optimize parallel traversal for scale-free graphs, which can make each processor have not only a similar number of edges but also balanced communication. Inspired by this work, we exploit the delegate partitioning method that duplicates high-degree vertices (named *delegates*), and then re-distributes their associated edges among processors.

We design our distributed Infomap algorithm by following the general logic of the sequential Infomap algorithm: After the graph partitioning step, the distributed algorithm conducts local clustering on each processor and then swaps necessary module information (e.g., exit probability and visit probability) among processors. If there is no more vertex movement, the algorithm merges the communities into a new graph and repeats the clustering on the new graph. The algorithm stops if there is no more MDL change.

Due to the involvement of delegates, the information exchange and synchronization become challenging with massive processors. The way to efficiently and effectively swap information among processors not only affects the accuracy but also the scalability of the algorithm. In addition, in a distributed environment, the convergence property of the Infomap algorithm is also an important factor that we should consider.

In this section, we will show our solution to address all these issues. We will first introduce the framework of our distributed Infomap algorithm (Section 3.2), and then lay out the key components of the framework, including preprocessing (Section 3.3), parallel local clustering and information swapping (Section 3.4), and distributed graph merging (Section 3.5).

3.2 Distributed Infomap Algorithm Framework

Algorithm 2 shows the framework of our distributed Infomap algorithm that consists of four stages:

The first stage corresponds to Line 1 that is the *preprocessing* stage. In this stage, we partition the graph with delegates, calculate the visit probability of each vertex, and calculate the exit probability of each link as described in Section 3.3. For an undirected graph, we transform it into a directed graph. After delegate partitioning, each processor can have a similar number of edges, and the subgraph on each processor consists of both high-degree duplicates and low-degree vertices.

The second stage is referred as *parallel clustering with delegates*, corresponding to Lines 2 to 7. In this stage, the subgraph on each

Algorithm 2 Parallel Infomap Algorithm on Each Processor

Require:
$G = (V, E)$: undirected graph, where V is vertex set and E is the edge set;
p : processor number.
Ensure:
M: resulting module set;
L: resulting MDL;
δL: change of MDL.

```
1:  Preprocessing
2:  repeat
3:      Local clustering with duplicates
4:      Broadcast delegate states with the minimum δL
5:      Swap community information
6:      Update community information on each processor
7:  until No vertex community state changing
8:  Merge communities into a new graph, and partition the new
    graph using 1D partitioning
9:  repeat
10:     repeat
11:         Local clustering
12:         Swap community states
13:         Update community information on each processor
14:     until No vertex movement
15:     Merge communities into a new graph
16: until No improvement of MDL
```

processor consists of low-degree vertices and duplicated hubs. The algorithm calculates the best community movement for each vertex as Line 3. In order to make sure that each delegate has consistent community movement information and δL, the algorithm broadcasts the information of delegates that achieve the largest decrease of MDL. Although this is a collective operation involving all processors, its cost is marginal because of a limited number of delegates. After the information communication between Lines 4 and 5, the algorithm updates local community information, such as the exit probability and the visit probability of modules. This process continues until there is no more community changing for each vertex.

The third stage, corresponding to Line 8, merges the communities into a new graph, and applies a normal 1D partition for the newly merged graph. The reason for us to use the normal 1D partition is that after the step of graph merging, the size of the new graph is several orders of magnitude less than the original graph.

The fourth stage, corresponding to Lines 10 to 14, processes the subgraphs in a way similar to Lines 2 to 7, except there are no delegated vertices in the subgraphs. Thus, this stage is referred as *parallel clustering without delegates*. The algorithm stops when there is no more improvement of modularity.

3.3 Preprocessing

In our preprocessing stage, we first use delegate partitioning [20] to achieve balanced workload and communication cost among processors. The basic idea is that vertices with degrees greater than a threshold are duplicated and distributed on all processors, while a

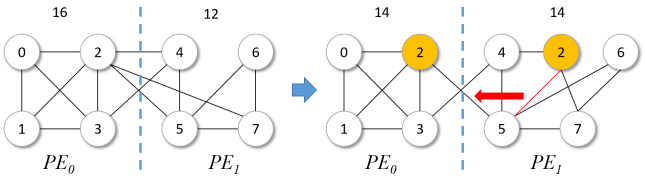

Figure 2: Delegate partitioning among two processors.

basic 1D partitioning is applied to low-degree vertices. The high-degree vertices or hubs are referred as *delegates*. Ideally, after partitioning, an outgoing edge whose source vertex is high-degree will be stored in the partition containing the edge's target vertex. In this way, the delegate and the target vertex will co-locate in the same partition. Thus, each processor can have a similar number of edges.

According to the map equation in Section 2.2, in the partitioning phase, we need to initialize the visit probability of each vertex and the exit probability of each link. For an input undirected graph $G = (V, E)$ with a vertex set V and an edge set E, the delegate partitioning can be concluded as the following steps on p processors:

First, we calculate the degree $degree(\alpha)$ of each vertex α in a distributed manner. The visit probability p_α of each vertex α as $degree(\alpha)/|E|$, which is also its exit probability.

Second, we detect high-degree vertices and duplicate them on all processors. The high-degree vertices are identified based on a threshold d_{high}. Accordingly, the edge set E is partitioned into two subsets: E_{high} (whose source vertexes are high-degree) and E_{low} (whose source vertex degrees are less than d_{high}). The delegates of high-degree vertices are created on all processors. After this step, the local vertices on each processor include the duplicated high-degree vertices, and the low-degree vertices that are partitioned by the traditional 1D partitioning.

Third, we define a round-robin 1D partitioning, where we partition the edges in E_{low} according to their source vertex partitioning mapping, and partition the edges in E_{high} according to their target vertex partitioning.

Fourth, we correct possible partition imbalances. Ideally, the number of edges locally assigned to each processor (i.e., E_{low} and E_{high}) should be close to $\frac{|E|}{p}$ for p processors. However, this may not be gained through the previous steps. In order to achieve this goal, we reassign an edge in E_{high} to any partition because its source vertex is duplicated on all processors. In particular, we reassign these edges to those processors whose numbers of edges are less than $\frac{|E|}{p}$. In the original sequential Infomap algorithm, it treats each undirected edge e_{ab} between two vertices a and b as a directed edge. The direction is defined by the order of vertex ID, that is if $a < b$, the edge e_{ab} is the outlink for the vertex a; otherwise, the edge e_{ab} is the inlink for a. For each link, we also need to calculate its exit probability, which is defined as $\frac{1}{|E|}$. Compared with the original work [20], we do not differentiate the delegates among the master and worker processors.

Figure 2 shows an example of the delegate partitioning result. Originally, all vertices are evenly distributed on 2 processors, as shown in the left image. Because each vertex needs to calculate δL for all its neighbor vertices, the workload of each vertex is proportional to its edge number. If we add together the edge number of

List 1 Message Interface

```
1:  struct {
2:      // module ID
3:      uint64_t modID;
4:      // sum of visit probability of the module
5:      double sumPr;
6:      // sum of exit probability of the module
7:      double exitPr;
8:      // vertex number in this module
9:      int numMembers;
10:     // whether this local module has been sent or not
11:     bool isSent;
12: } Module_Info;
```

each vertex on each processor, we can clearly see that the workload is imbalance among the processors, where PE_0 has 16 edges, and PE_1 has 12 edges. If we choose the degree threshold as 5, then the vertex 2 is the delegate and is duplicated. In order to correct possible partition imbalances, we move the edges connecting the delegates and the low-degree vertices. For example, we move the edge (2,5) on PE_1 to PE_0, as shown in the right image. Therefore, each processor has 14 edges, and the final partition is balanced.

3.4 Parallel Local Clustering and Information Swapping

After delegate partitioning, each processor gains a subgraph $G_s = (V_s, E_s)$, where V_s is the vertex set and E_s is the edge set of the subgraph G_s. As stated in Section 3.2, in the stage of parallel clustering with delegates (Lines 2 to 7 in Algorithm 2), V_s is divided into V_{low} and V_{high}, where V_{low} is the low-degree vertices and V_{high} is the global high-degree vertices (i.e., hubs). In the stage of parallel clustering without delegates (Lines 10 to 14 in Algorithm 2), we do not differentiate the high-degree vertices and the low-degree vertices.

The parallel clustering algorithm runs on each subgraph following similar steps as the sequential Infomap algorithm:

- Each processor calculates δL for each vertex in its subgraph. Each processor broadcasts the high-degree vertices who achieve the minimum local δL, and swaps the community IDs of its boundary vertices with its neighbor processors. To decide the movement of a vertex, each processor first checks whether the vertex should be moved to a boundary community. If so, we use the minimum label strategy to avoid the vertex bouncing problem [17]. Otherwise, the vertex can be moved to any community.
- Each processor updates its local module information, calculates its local minimal MDL value, and then uses the `Allreduce` function to obtain the global minimal MDL value from the other processors.
- Each processor swaps its community information with its neighbor processors for calculating δL in the next iteration.
- If there is no more vertex movement or there is no more MDL optimization, the original graph is merged into a new graph (this step will be described in Section 3.5).

Algorithm 3 Parallel Information Swapping on Each Processor

```
1:  //Prepare information swapping
2:  V_high_min ← the hubs with the global minimal MDL
3:  for all u ∈ V_high_min do
4:      if Module_Info(u) NOT sent then
5:          Construct Module_Info(u) with isSent as false
6:      else
7:          Set Module_Info(u) with isSent as true
8:      end if
9:  end for
10: V_low ← the low-degree vertices
11: for all u ∈ V_low do
12:     if u is a ghost vertex on other processors then
13:         if Module_Info(u) NOT sent then
14:             Construct Module_Info(u) with isSent as false
15:         else
16:             Set Module_Info(u) with isSent as true
17:         end if
18:     end if
19: end for
20: Swap module information with neighbor processors
21: //Update module information
22: for all m in received module information do
23:     if m.modID NOT exist then
24:         Build a new module according to m
25:     else
26:         if m.isSent == false then
27:             Add the information of m to the existing module
28:         else
29:             Continue
30:         end if
31:     end if
32: end for
```

- For the newly merged graph, where high-degree vertices and low-degree vertices are not differentiated, the above steps are applied until there is no more MDL change.

Although the collective operation (i.e., `Allreduce`) has been used, its cost is marginal because of a limited number of hubs. In Section 4, we evaluate the performance results to verify the workload balancing and the scalability of our algorithm.

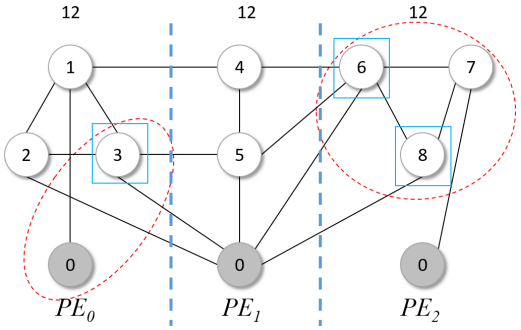

Figure 3: Example of information swap.

In our distributed algorithm, a well-designed information swapping strategy is needed. As illustrated in Figure 3, the vertices 0 and 3 on PE_0 are in the same community, while the vertices 6, 7 and 8 on PE_2 are in the same community. A naive information swapping strategy is to let each processor only send the community information of boundary vertices. For example, PE_0 only sends the information of the vertex 3 and PE_2 only sends the information of the vertices 6 and 8. In this case, after information swapping, the vertex 5 cannot have the whole neighbor community information, such that the vertex 5 does not know that the vertices 0 and 3 are in the same community on PE_0, and the vertex 7 is in the same community with the vertices 6 and 8 on PE_2. This inconsistency on local community information incurs inaccurate results in the following iterations, leading to a severe problem for detecting final global communities.

To address this challenging issue, in our information swapping, we consider swapping the whole community information of each boundary vertex. However, this can incur a problem that the same community information can be sent multiple times. Considering Figure 3, when we send the community information of the vertices 6 and 8 from PE_2 to PE_1, because they are in the same community, their community information will be received twice on PE_1. In order to overcome this disadvantage, we design a message interface $Module\_Info$ as List 1 for swapping the whole local community information of boundary vertices. As we can see, the message interface $Module\_Info$ contains not only the module information (e.g., ID, the sum of visit probability, the sum of exit probability, etc.), but also the information on whether this module information has been sent or not.

Based on this message interface, we develop the information swapping algorithm as Algorithm 3. Each processor first prepares its local module information that needs to be swapped with the neighbor processors (Lines 2 to 19). Specifically, each processor only needs to consider two types of vertices: the hubs that have the global minimal MDL value, and the low-degree vertices that are the ghost vertices on other processors. For each of these two types of vertices, each processor constructs the message interface $Module\_Info$. If the module information of a vertex has been sent, its *isSent* attribute is set as $true$, otherwise $false$. Then, the processors swap the module information (Line 20). After receiving the information from its neighbor processors, each processor updates its local module information (Lines 22 to 32). For newly received module information, a processor builds a new module accordingly. For the existing module information, if it has not been sent before, it is added to the existing module; otherwise, it is skipped. Through this means, we can effectively eliminate the possible duplication of module information, while ensuring that the whole community information of each boundary vertex can be synchronized to the relevant processors.

3.5 Distributed Graph Merging

This step is relatively simple and intuitive. On each processor, the algorithm merges the local communities into a new graph, where a community becomes a vertex with the same community ID in the newly merged graph. Then, the processor sends the information of the new vertices and their adjacent edges to their assigned processors according to the 1D partitioning.

4 EXPERIMENTS

We evaluate the quality and the scalability of our distributed Infomap algorithm using different datasets. We also assess the convergence of our distributed algorithm. Finally, we show a detailed performance evaluation, including the analysis of the partitioning, the execution time breakdown, and the scalability.

Table 1: Datasets.

Name	Description	#Vertices	#Edges
Friendster [8]	An on-line gaming network	65.61M	1.81B
UK-2007 [8]	Web crawl of the .uk domain in 2007	105.9M	3.78B
UK-2005 [8]	Web crawl of the .uk domain in 2005	39.46M	936.4M
WebBase-2001 [9]	A crawl graph by WebBase	118.14M	1.01B
ND-Web [1]	A web network of University of Notre Dame	0.33M	1.50M
LiveJournal [9]	A virtual-community social site	5.20M	76.94M
YouTube [28]	YouTube friendship network	11.34M	29.87M
DBLP [28]	A co-authorship network from DBLP	0.31M	1.04M
Amazon [28]	Frequently co-purchased products from Amazon	0.33M	0.92M

As shown in Table 1, our experiment has used the datasets at different scales. These include small-scale graphs (the vertex number is less than 1 million, and the edge number is around 1 million; e.g., Amazon, DBLP, and ND-Web), medium-scale graphs (the vertex number is between 1 million and 10 million, and the edge number is in the order of 10 million; e.g., LiveJournal and YouTube), and large-scale graphs (the edge number is around 1 billion; e.g., UK-2005, WebBase-2001, Friendster, and UK-2007). As far as we know, the UK-2007 is one of the largest real-world undirected datasets that are publicly available.

We implemented our algorithm using MPI and C++. Our experiments are performed on *Titan*, a supercomputer at the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory. Each compute node contains a 16-core 2.2GHz AMD Opteron processor and 32GB memory. We set the threshold d_{high} as the processor number to determine high-degree vertices (i.e., hubs).

4.1 Community Quality Analysis

It is non-trivial to achieve the convergence of a distributed Infomap algorithm. To this end, we first examine the convergence of our algorithm, and compare the quality of community detection between our distributed algorithm and the sequential algorithm.

Due to the page limit, in Figure 4, we compare the convergence of our distributed algorithm to the sequential algorithm using the Amazon, DBLP, NDWeb, and YouTube datasets. As shown in each plot of Figure 4, our distributed Infomap algorithm can achieve a converged MDL close to the sequential algorithm. Similar results have been also obtained on the other datasets. This shows that our solution can address the challenging convergence problem when involving delegates in distributed community detection.

We also compare the merging rate of the sequential Infomap algorithm and our distributed algorithm in Figure 5. The merging rate is the merged vertex number of each outer iteration compared to the original graph vertex number. As shown in Figure 5, our algorithm conveys a similar convergence pattern as the sequential

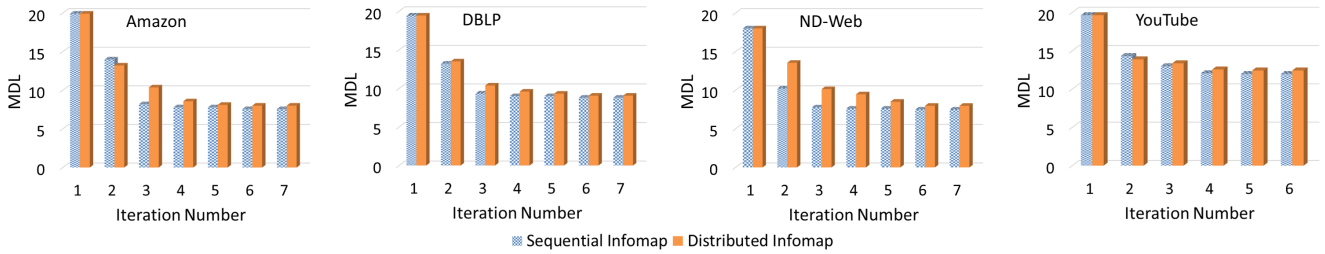

Figure 4: Comparison of MDL between the sequential algorithm and our distributed algorithm.

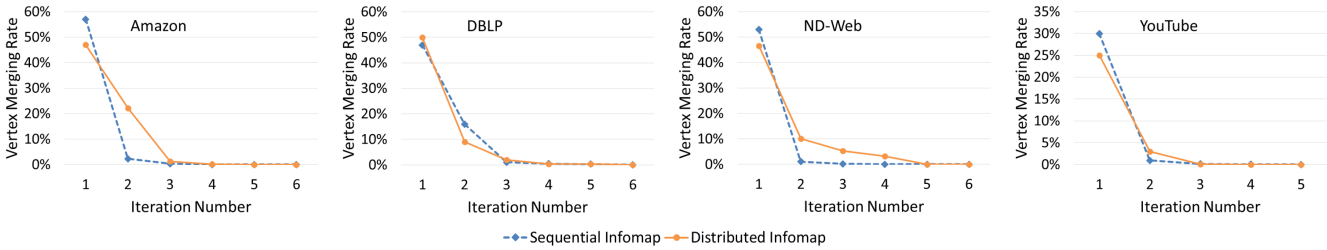

Figure 5: Comparison of vertex merging rate between the sequential algorithm and our distributed algorithm.

algorithm. We notice that as our algorithm using delegates in clustering, after the first iteration, the merging rate is usually around 50%. This means, after clustering with vertex delegates (Lines 2 to 7 in Algorithm 2), the original graph can be efficiently merged into a graph that is several orders of magnitude smaller. Therefore, we do not need the delegate partitioning in the second stage clustering (Lines 10 to 14 in Algorithm 2).

Apart from MDL, we have examined other quality measurements to make sure that our algorithm can achieve high-quality results of community detection. The measurements include Normalized Mutual Information (NMI), F-measure, and Jaccard Index (JI). For all these measurements, a high value corresponds to a high quality [27]. Table 2 shows the results for four datasets. All these values are around 0.80, which means our distributed algorithm can achieve similar results as the sequential algorithm.

Table 2: Quality Measurements.

Dataset	NMI	F-measure	JI
DBLP	0.79	0.80	0.78
Amazon	0.82	0.81	0.80

4.2 Workload and Communication Balance Analysis

There is very limited existing work on partitioning strategies of distributed Infomap algorithms. In other distributed graph clustering research, the 1D partitioning is a common strategy for distributing the original graph datasets [10, 21, 29, 30]. Thus, we compare our delegate partitioning with the 1D partitioning on different datasets.

We first investigate the effect of different partitioning methods on the clustering workload balance. In order to compare the workload balance between the 1D partitioning and our delegate partitioning, we count the edge number of each subgraph on a processor. This is because, for a distributed Infomap algorithm, each vertex needs to calculate δL for all its neighbor vertices, and the total workload is proportional to the total edge number on this processor. In Figure 6, we examine the workload on each processor between the 1D partitioning and our delegate partitioning. For all the large real-world datasets, our delegate partitioning assigns each processor a similar number of edges. We find that on all these large datasets, the difference can be significant between these two methods. For example, when using the 1D partitioning on UK-2005, the maximum workload can be $O(10^7)$ edges for a processor, while the maximum workload on each processor is similar and less than 10^6 when the delegate partitioning is used. For WebBase-2001, Friendster, and UK-2007, if using the 1D partitioning, the minimum workload can be only hundreds of edges while the maximum workload can be up to $O(10^8)$. We also find that although UK-2005 has fewer edges than WebBase-2001, the maximum workload of UK-2005 is $O(10^7)$ and the maximum workload of WebBase-2001 is $O(10^6)$. This is because the vertices in UK-2005 are more densely connected than those in WebBase-2001. This means that high-degree vertices are more prone to incur imbalanced workload using the 1D partitioning.

We also explore the effect of the 1D partitioning and our delegate partitioning on the communication cost. Because the information swapping is through boundary vertices in our distributed algorithm, the communication cost correlates with the ghost vertex number. By comparing the ghost vertices number between two partitioning strategies, we can easily infer the communication cost. In Figure 7, we compare the communication cost of two strategies on large datasets. As shown in Figure 7, the difference of ghost vertex number can be extremely large for the 1D partitioning. For a distributed algorithm, the communication cost is mostly determined by the slowest part. Thus, with the 1D partitioning, there can be an extremely high number of ghost vertices on certain processors, which

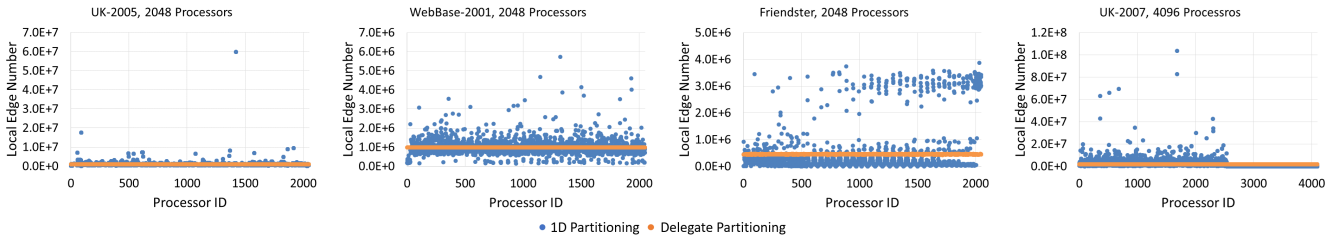

Figure 6: Comparison of workload balance between the 1D partitioning and the delegate partitioning.

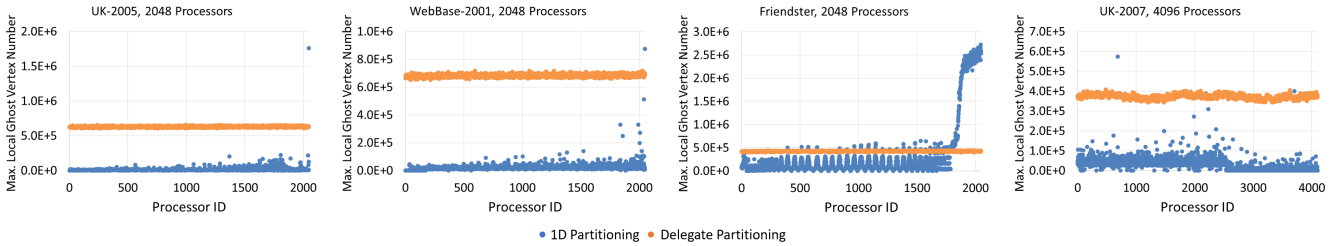

Figure 7: Comparison of communication balance between the 1D partitioning and the delegate partitioning.

can cause intense communication and impair the total performance of the distributed algorithm. Meanwhile, we can clearly see that our delegate partitioning can effectively balance the local ghost vertex number among processors for each dataset. Thus, our delegate partitioning can significantly reduce the communication cost.

4.3 Execution Time Breakdown

We examine the main performance components of our algorithm using large real-world datasets. We break down the key components in our first clustering stage with delegates.

In the first clustering stage, there can be multiple iterations for calculating δL of each vertex and swap community information. We show one iteration running time on different datasets in Figure 8. There are four key components in our profiling results, which are *Find Best Module*, *Broadcast Delegates*, *Swap Boundary Information* and *Other*. In *Find Best Module*, the algorithm calculates δL for each vertex and moves the vertex to the neighbor community with the minimum δL. After this step, each processor uses *Broadcast Delegates* to broadcast the community information using the message structure in List 1, and uses *Swap Ghost Vertex State* to send the boundary community information. The *Other* part mainly updates the information of communities, such as the visit probability and the exit probability. Moreover, it creates new modules and updates existing module information.

As we stated previously, the collective operation (broadcasting hubs information) for all delegated vertices only needs a small portion of time in each iteration.

In Figure 8, we also notice that the time of *Find Best Module*, *Broadcast Delegates* and *Other* are reduced with the increasing number of processors. For *Find Best Module*, it is much related to the workload on each processor. With our delegate partitioning, we can evenly partition the workload among processors. The number of high-degree vertices decreases with the increasing number of processors, and thus the time of *Broadcast Delegates* can also be decreased. While in the *Other* part, our algorithm mainly updates local community information, which is related to the number of local communities. For *Swap Boundary Information*, we find its execution time is relatively stable, and does not change significantly with the processor number. The reason is that when the graph is partitioned among more processors, the number of ghost vertices on the different numbers of processors can still be the same order and all these ghost vertices need to be swapped in each iteration.

4.4 Scalability Analysis

We examine the scalability of our algorithm on large real-world datasets. As we stated in Section 3.2, our algorithm first clusters subgraphs with delegates and then clusters merged graphs without delegates, the running time of our algorithm should be the sum of these two stages running times. As we discussed previously, our method can achieve balanced computation workload and communication workload, which can assure a better scalability. Figure 9 shows the running times of our distributed algorithm on four large real-world datasets: UK-2005, WebBase-2001, Friendster, and UK-2007. We can clearly see that the total running time for each dataset is nearly inversely proportional to the processor number, which indicates that our clustering algorithm can achieve a scalable performance on large real-world datasets.

In order to quantify the scalability of our algorithm, we measure the parallel efficiency, more specifically, the relative parallel efficiency τ that is defined as:$\tau = \frac{p_1 T(p_1)}{p_2 T(p_2)}$, where p_1 and p_2 are the processor numbers, and $T(p_1)$ and $T(p_2)$ are their corresponding running times. Figure 10 shows the parallel efficiency of our algorithm for the small real-world datasets and the large real-world datasets. For the baseline of each dataset, we use the running time on a minimal number of processors that can suitably handle the data size. Specifically, we use the running times on 16 processors

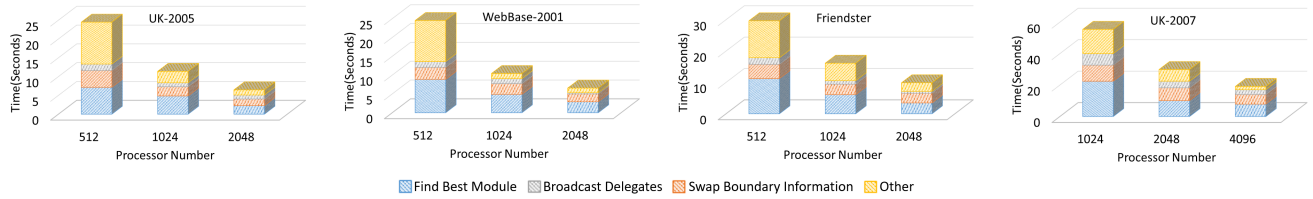

Figure 8: Time breakdown on real-world datasets.

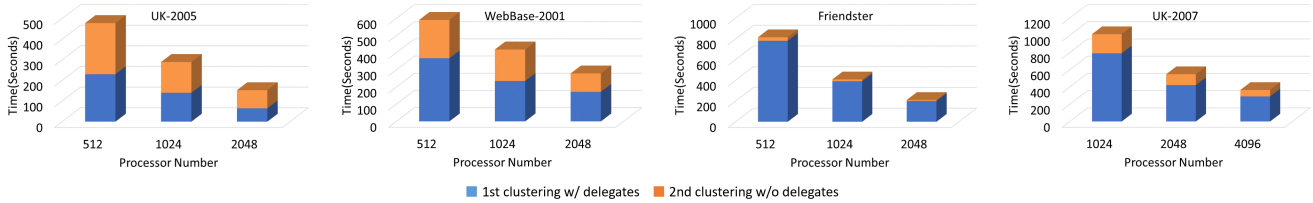

Figure 9: Scalability study of our algorithm using different datasets and different processor numbers. Our total clustering time contains the times of the first clustering stage with delegates and the second clustering stage without delegates.

for Amazon, DBLP, and ND-Web, 64 processors for YouTube, 256 processors for UK-2005, Webbase-2001, and Friendster, and 1024 processors for UK-2007. Figure 10 (top) shows that our parallel algorithm can achieve at least around 65% parallel efficiency for most small- and medium-scale graphs, and achieve nearly 100% parallel efficiency on 64 cores for the Amazon dataset. Figure 10 (bottom) shows that our algorithm can achieve at least 70% parallel efficiency for most large graphs. Through our analysis in Figure 9, we know that the first clustering stage with delegates can be the dominant part of the total running time. For those whose parallel efficiency is around 100% (e.g., the Friendster dataset), the running time of the first clustering stage with delegates is effectively reduced with the increasing processor number. This states that through duplicating high-degree vertices, our approach can effectively evenly distribute the workload of graph clustering among all processors.

Table 3: Speedup of our algorithm on different datasets.

Dataset	ND-Web	LiveJournal	WebBase-2001	UK-2007
Speedup	1.08 ×	3.05 ×	3.18 ×	6.02 ×

We also compared the running time of our distributed algorithm and the previous state-of-the-art Bae *et al.*'s work [4, 5]. We use the fastest time in Bae *et al.*'s work to calculate the speedup of our approach as shown in Table 3. We can see that for a small dataset (e.g., ND-Web) the speedup is not significant. However, with the size of graph increasing, our speedup is enlarged noticeably. Especially on UK-2007, our approach archives about a 6× speedup.

5 DISCUSSION

Our experimental study clearly shows the scalability of our clustering algorithm. However, we observe the different running times of the first and second clustering stages across different datasets. As shown in Figure 9, the running times of these two stages are similar on UK-2005 and WebBase-2001. While on Friendster and

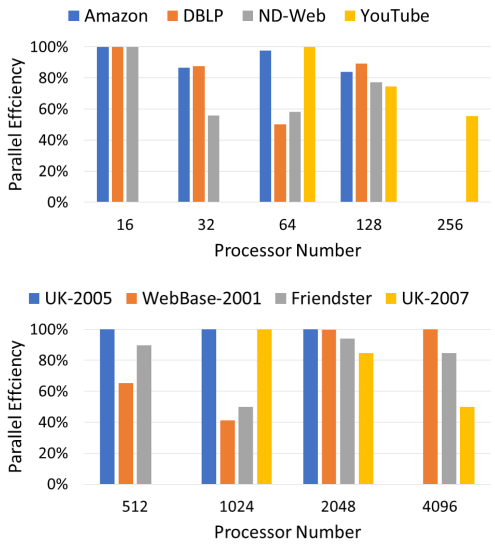

Figure 10: Parallel efficiency of our algorithm with (top) the Amazon, DBLP, ND-Web, and YouTube datasets, and (bottom) the UK-2005, WebBase-2001, Friendster, and UK-2007 datasets.

UK-2007, the running time of the second clustering stage is shorter than the first stage.

We find that this is because the Friendster and UK-2007 datasets can be clustered into a smaller number of clusters in the first stage. Therefore, the running time of the second stage can be much shorter. For the UK-2005 and WebBase-2001 datasets, in the first stage, although their sizes are also reduced significantly, the resulting cluster numbers are comparably larger. Moreover, we find that for UK-2005 and WebBase-2001, the vertex movements reduce δL relatively marginally in each iteration of the second stage. Thus, the

second stage needs more iterations and a longer running time. We will conduct a more detailed investigation on the different running times of the first and second stages on different datasets.

6 CONCLUSION

We present a distributed Infomap algorithm for scalable and high-quality community detection. Our algorithm can gain accurate community detection results and achieve balanced computation workload and communication among massive processors for large graphs, which demonstrate a clear improvement over the previous state-of-the-art. Our algorithm makes Infomap scalable and practical to achieve high-quality communities from large graphs. In the future, we would like to extend our algorithm using heterogenous architectures and exploit hardware accelerations (e.g., GPUs) to further improve the scalability of community detection. Moreover, we will study algorithms to effectively and efficiently visualize community detection results of large graphs. We plan to apply our solution on other very large graphs from different domains.

ACKNOWLEDGMENTS

This research has been sponsored by the National Science Foundation through grants IIS-1423487 and IIS-1652846.

REFERENCES

[1] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 1999. Internet: Diameter of the world-wide web. *nature* 401, 6749 (1999), 130–131.
[2] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit Order: Just-in-time parallel reordering for fast graph analysis. In *Parallel and Distributed Processing Symposium (IPDPS), 2016 IEEE International*. IEEE, 22–31.
[3] Seung-Hee Bae, Daniel Halperin, Jevin West, Martin Rosvall, and Bill Howe. 2013. Scalable flow-based community detection for large-scale network analysis. In *Data Mining Workshops, 2013 IEEE 13th International Conference on*. 303–310.
[4] Seung-Hee Bae, Daniel Halperin, Jevin D West, Martin Rosvall, and Bill Howe. 2017. Scalable and efficient flow-based community detection for large-scale graph analysis. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 11, 3 (2017), 32.
[5] Seung-Hee Bae and Bill Howe. 2015. GossipMap: a distributed community detection algorithm for billion-edge directed graphs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.
[6] Sanjukta Bhowmick and Sriram Srinivasan. 2013. A template for parallelizing the Louvain method for modularity maximization. In *Dynamics On and Of Complex Networks, Volume 2*. Springer, 111–124.
[7] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.
[8] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.
[9] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph framework I: compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. 595–601.
[10] Chun Yew Cheong, Huynh Phung Huynh, David Lo, and Rick Siow Mong Goh. 2013. Hierarchical parallel algorithm for modularity-based community detection using GPUs. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par'13)*. 775–787.
[11] Santo Fortunato. 2010. Community detection in graphs. *Physics Reports* 486, 3-5 (2010), 75–174.
[12] Michelle Girvan and Mark EJ Newman. 2002. Community structure in social and biological networks. *Proceedings of the national academy of sciences* 99, 12 (2002), 7821–7826.
[13] Steve Harenberg, Gonzalo Bello, L Gjeltema, Stephen Ranshous, Jitendra Harlalka, Ramona Seay, Kanchana Padmanabhan, and Nagiza Samatova. 2014. Community detection in large-scale networks: a survey and empirical evaluation. *Wiley Interdisciplinary Reviews: Computational Statistics* 6, 6 (2014), 426–439.
[14] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.
[15] Hao Lu, Mahantesh Halappanavar, Daniel Chavarría-Miranda, Assefaw Gebremedhin, and Ananth Kalyanaraman. 2015. Balanced coloring for parallel computing applications. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 7–16.
[16] Hao Lu, Mahantesh Halappanavar, Daniel Chavarria-Miranda, Assefaw H Gebremedhin, Ajay Panyala, and Ananth Kalyanaraman. 2017. Algorithms for balanced graph colorings with applications in parallel computing. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (2017), 1240–1256.
[17] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel Comput.* 47 (2015), 19–37.
[18] Md Naim, Fredrik Manne, Mahantesh Halappanavar, and Antonino Tumeo. 2017. Community detection on the GPU. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 625–634.
[19] Andrew Y Ng, Michael I Jordan, Yair Weiss, et al. 2002. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems* 2 (2002), 849–856.
[20] Roger Pearce, Maya Gokhale, and Nancy M Amato. 2014. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 549–559.
[21] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A Gunnels. 2015. Scalable community detection with the Louvain algorithm. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 28–37.
[22] Martin Rosvall, Daniel Axelsson, and Carl T Bergstrom. 2009. The map equation. *The European Physical Journal Special Topics* 178, 1 (2009), 13–23.
[23] Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce. 2016. Graph colouring as a challenge problem for dynamic graph processing on distributed systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 30.
[24] Christian L Staudt and Henning Meyerhenke. 2013. Engineering high-performance community detection heuristics for massive graphs. In *Parallel Processing, 2013 42nd International Conference on*. IEEE, 180–189.
[25] Christian L Staudt and Henning Meyerhenke. 2016. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2016), 171–184.
[26] Charith Wickramaarachchi, Marc Frincu, Patrick Small, and Viktor K Prasanna. 2014. Fast parallel algorithm for unfolding of communities in large graphs. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 1–6.
[27] Jierui Xie, Stephen Kelley, and Boleslaw K Szymanski. 2013. Overlapping community detection in networks: The state-of-the-art and comparative study. *Acm computing surveys (csur)* 45, 4 (2013), 43.
[28] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics (MDS '12)*. Article 3, 8 pages.
[29] Jianping Zeng and Hongfeng Yu. 2015. Parallel modularity-based community detection on large-scale graphs. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 1–10.
[30] Jianping Zeng and Hongfeng Yu. 2016. A study of graph partitioning schemes for parallel graph community detection. *Parallel Comput.* 58 (2016), 131–139.