A Scalable Distributed Louvain Algorithm for Large-scale Graph Community Detection

Jianping Zeng, Hongfeng Yu
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, Nebraska
Email: {jizeng, yu}@cse.unl.edu

***Abstract*—We present a new distributed community detection algorithm for large graphs based on the Louvain method. We exploit a distributed delegate partitioning to ensure the workload and communication balancing among processors. In addition, we design a new heuristic strategy to carefully coordinate the community constitution in a distributed environment, and ensure the convergence of the distributed clustering algorithm. Our intensive experimental study has demonstrated the scalability and the correctness of our algorithm with various large-scale real-world and synthetic graph datasets using up to 32,768 processors.**

***Index Terms*—large graph, community detection, graph clustering, parallel and distributed processing, scalability, accuracy.**

I. Introduction

Community detection, also named *graph clustering*, aims to identify sets of vertices in a graph that have dense intra-connections, but sparse inter-connections [1]. Community detection algorithms have been widely used to retrieve information or patterns of graphs in numerous applications and scientific domains [2]–[6].

However, it remains a challenging and open research problem to parallelize a community detection algorithm and make it scalable to tackle real-world large graphs. This is mainly because graphs generated from many domains are referred as to *scale-free* graphs, where the vertex degree distribution of such a graph asymptotically follows a power law distribution [7]. High-degree vertices, also called *hubs*, create significant challenges in the development of scalable distributed community detection algorithms. First, for community detection, the workload associated with a vertex is typically proportional to the vertex degree. Thus, it is difficult to effectively partition and distribute hubs to balance workload and communication among processors. Second, it is non-trivial to efficiently synchronize the community states of vertices (particularly, hubs) among processors. Without such information, a processor cannot accurately compute its local communities, which can further impair the correctness of final aggregated results.

In this paper, we present a new distributed modularity-based algorithm to boost the scalability and the correctness of community detection, and make the following three main contributions:

First, we extend a graph partitioning and distribution method with vertex delegates used in distributed graph traversal algorithms [8] for distributed community detection. Through a careful duplication of hubs among processors, our method can ensure each processor to process a similar number of edges, and balance workload and communication among processors.

Second, we design a novel distributed Louvain algorithm based on our new partitioning and distribution method. Our algorithm can maximize modularity gain using a greedy strategy. In addition, for low-degree vertices, the algorithm swaps their update community information among processors through a modified minimum label heuristic. Our design can avoid the inter-community vertex bouncing problem that causes non-convergence in community detection.

Third, we conduct an intensive experimental study to demonstrate the effectiveness and the correctness of our distributed algorithm. We not only evaluate the quality and the scalability of community detection, but also experiment the communication cost for large graphs, which is not fully investigated in existing research work. Our experiments show that our approach can process large graphs (such as UK-2007, one of the large real-world graph datasets) in a scalable and correct manner. To the best of our knowledge, this result is clearly superior to the previous distributed modularity-based community detection algorithms.

II. Related Work

Modularity is one of commonly used measurements for community quality. Since the inception of sequential modularity-based algorithms [9], [10], several parallel algorithms based on shared-memory platforms have been proposed. Riedy et al. [11] presented an approach that performed multiple pairwise community merges in parallel. Xie et al. [12] presented a multi-threaded method that however can only process a graph in the order of ten million edges. Bhowmick et al. [13] proposed an OpenMP implementation that adopted a lock mechanism and showed a limited scalability with 16 threads and graphs with 10 thousand vertices. Lu et al. [14] implemented a shared-memory parallel Louvain algorithm using 32 threads. In general, the scalability of these methods is limited by the capacity of shared-memory platforms.

There is comparably limited work of distributed community detection algorithms based on modularity. Cheong et al. [15]

2168-9253/18/$31.00 ©2018 IEEE
DOI 10.1109/CLUSTER.2018.00044

presented a GPU-based Louvain algorithm that, however, ignored the connected vertices residing in different sub-graphs and caused an accuracy loss. Zeng et al. [16], [17] designed a new graph partitioning method suitable for a parallel Louvain algorithm, but their method needed a considerable preprocessing time and a global reduction cost. Que et al. [18] implemented a distributed Louvain algorithm with a nearly balanced computational load among threads, but the inter-processor communication was less balanced.

III. Preliminaries

In a *graph* $G = (V, E)$, V is the set of vertices (or nodes) and E is the set of edges (or links). The weight of an edge between two vertices, u and v, is denoted as $w_{u,v}$, which is 1 in a undirected unweighted graph. The community detection problem is to find overlapping or non-overlapping vertices sets, named communities, which contain dense intra-connected edges but sparse inter-connected edges. Without loss of generality, we only focus on non-overlapping community detection on undirected graphs in this work. However, our approach can be easily extended to directed graphs [15]. The non-overlapping community set C of a graph $G = (V, E)$ can be represented as:

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

A. Modularity-based Community Detection

Modularity Q is a measurement to quantify the quality of communities detected in a graph, which can be formulated as:

$$Q = \sum_{c \in C} \left(\frac{\sum_{in}^{c}}{2m} - \left(\frac{\sum_{tot}^{c}}{2m}\right)^2 \right), \tag{2}$$

where m is the sum of all edge weights in the graph, $\sum_{in}^{c}$ is the sum of all internal edge weights in a community c, calculated as $\sum_{in}^{c} = \sum w_{u,v} (u \in c \wedge v \in c)$, and $\sum_{tot}^{c}$ is the sum of all edge weights, calculated as $\sum_{tot}^{c} = \sum w_{u,v} (u \in c \vee v \in c)$. Newman et al. [19] used a null model against the original graph, where a high modularity score means the expected degree of each vertex matches the degree of the vertex in the original graph. To be simple, the intuition of Equation 2 is that if the modularity value is high, there are many edges inside communities but only a few between communities, indicating a high quality of community detection.

Modularity gain, δQ, is the gain in modularity obtained by moving an isolated vertex u into a community $c \in C$ [10], which can be computed by:

$$\delta Q_{u \to c} = \left[\frac{\sum_{in}^{c} + w_{u \to c}}{2m} - \left(\frac{\sum_{tot}^{c} + w(u)}{2m}\right)^2\right] - \left[\frac{\sum_{in}^{c}}{2m} - \left(\frac{\sum_{tot}^{c}}{2m}\right)^2 - \left(\frac{w(u)}{2m}\right)^2\right] \tag{3}$$

where $w_{u \to c}$ is the total weight of edges connecting a vertex u and a community c, and $w(u)$ is the weighted degree of u. After reduction, Equation 3 can be formulated as:

$$\delta Q_{u \to c} = \frac{1}{2m} \left(w_{u \to c} - \frac{\sum_{tot}^{c} \cdot w(u)}{m} \right). \tag{4}$$

Clauset et al. [9] introduced an agglomerative community detection algorithm that merged the vertices achieving the global maximum modularity values. Afterwards, Blondel et al. [10] proposed the Louvain algorithm that is a heuristic algorithm and can achieve better results with a lower time complexity. Thus, it has been widely used in practice [20], [21]. The Louvain algorithm greedily maximizes the modularity gain $\delta Q_{u \to c}$ when moving an isolated vertex u into a community c, which is based on Equation 4. This process continues until there is no more vertex movement to increase the modularity. 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.

B. Research Challenges

The sequential Louvain algorithm generally takes a long time (e.g., several hours) for a large-scale graph (e.g., one with billions of edges) using a single processor. In order to accelerate the processing of a large-scale graph, a common strategy is to partition and distribute a graph among multiple processors, and then conduct computation in a distributed and parallel fashion. However, a scale-free graph follows 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 can cause significant workload and communication imbalance, and inhibit the overall performance and scalability of distributed graph algorithms using conventional graph partitioning methods.

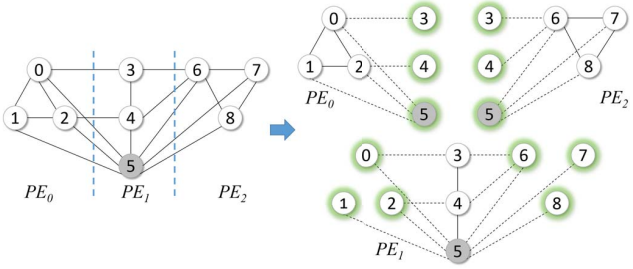


Fig. 1. Partitioning a graph among three processors.

We illustrate these problems using a simple graph in Figure 1 (left), where v_5 is a high-degree vertex. The original graph is partitioned on three processors, denoted as PE_0, PE_1, and PE_2. A vertex without a green circle is denoted as a *local vertex* belonging to a processor, while a vertex around by a green circle is a *ghost vertex* that resides on the other processor. As shown in the right image of Figure 1, PE_1, assigned with v_5, has more edges and thereby has more workload than the others. In addition, generally, ghost vertices are used for inter-processor communication in distributed community detection. On PE_1, there are more edges connecting the local vertices and the ghost vertices residing on PE_0 and PE_2, indicating more communications between PE_1 and the others and leading to the communication imbalance problem.

Although different distributed Louvain algorithms have been designed to avoid these problems [15], [16], [18], they suffer from various limitations such as accuracy loss, preprocessing overhead, or imbalanced inter-processor communication, as

discussed in Section II. We note that these distributed Louvain algorithms use 1D partitioning, which prefers to put the adjacency list of a vertex into a single partition.

We aim to address the problems associated with hub vertices to enable scalable distributed community detection for large scale-free graphs. We are inspired by the recent work conducted by Pearce et al. [8]. To optimize parallel traversal of scale-free graphs, Pearce et al. proposed a parallel partitioning method that duplicates high-degree vertices and re-distributes their associated edges among processors. These duplicated hubs are named *delegates*. Using this method, they can ensure that each processor contains a similar number of edges, and meanwhile achieve balanced communication among processors.

At first glance, delegate partitioning appears a viable method leading to a straightforward parallelization solution for the Louvain algorithm. However, there remain two challenges:

- It is challenging to evaluate the modularity gain δQ for each vertex when updating its community membership in a distributed environment. In our case, due to the involvement of hub delegates on a processor, it is non-trivial to make sure each delegate vertex has a synchronized and accurate community state and avoid a high communication cost across processors.
- For the sequential Louvain algorithm, it is trivial to maintain (or improve) the convergence property, because it always increases the modularity when moving a vertex to a community using the greedy policy. However, in a distributed environment, a ghost vertex may be shared by multiple processors, which can incur the bouncing problem of the ghost vertex from different communities on different processors, and thereby impair the convergence of the distributed parallel algorithm.

These two issues can decimate the scalability and the accuracy of distributed community detection for handling the ever-increasing volume of graph data.

We develop a new approach to address these challenges. We will first introduce the framework of our distributed Louvain algorithm (Section IV-A), and then lay out the key components of the framework, including a distributed delegate partitioning strategy to balance workload and communication among processors (Section IV-B), an enhanced heuristic strategy to assure the convergence of parallel clustering (Section IV-C), our scalable local clustering on each processor (Section IV-D), and distributed graph merging (Section IV-E).

IV. Our Method

A. Distributed Louvain Algorithm Framework

Algorithm 1 shows the framework of our distributed Louvain algorithm that consists of four stages:

The first stage *distributed delegate partitioning* corresponds to Line 1, where the algorithm uses a delegate partitioning to duplicate hubs among processors to make sure each processor have a similar number of edges.

The second stage is referred as *parallel local clustering with delegates*, corresponding to Lines 2 to 7. In this stage, the subgraph on each 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 each delegate have consistent community movement information and modularity gain, the algorithm broadcasts the information of delegates that achieve the maximum modularity gain. Although this is a collective operation involving all processors, its cost is marginal because of a limited number of hubs. In Section V-C, we will show the communication time. After the information communication from Lines 4 to 5, the algorithm updates local community information, such as $\sum_{tot}^{c}$ and $\sum_{in}^{c}$. This process continues until there is no more community changing for each vertex.

The third stage *distributed graph merging*, corresponding to Line 8, merges the communities into a new graph. As the new graph is several order smaller than the original graph, we then apply 1D partitioning on the new 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 local clustering without delegates*. The algorithm stops when there is no more improvement of modularity.

Algorithm 1 Distributed Louvain Algorithm

Require:
$G = (V, E)$: undirected graph, where V is vertex set and E is the edge set;
p: processor number.
Ensure:
C: resulting community;
Q: resulting modularity.
1: Distributed Delegate Partitioning(G, p);
2: **repeat**
3: Parallel local clustering with delegates
4: Broadcast delegates achieving the highest modularity gain
5: Swap ghost vertex community states
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: Parallel local clustering without delegates
12: Swap ghost vertex 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 modularity

B. Distributed Delegate Partitioning

We extend the delegate partitioning proposed by Pearce et al. [8] as a preprocessing step in our parallel community detection algorithm for large scale-free graphs. The basic idea of

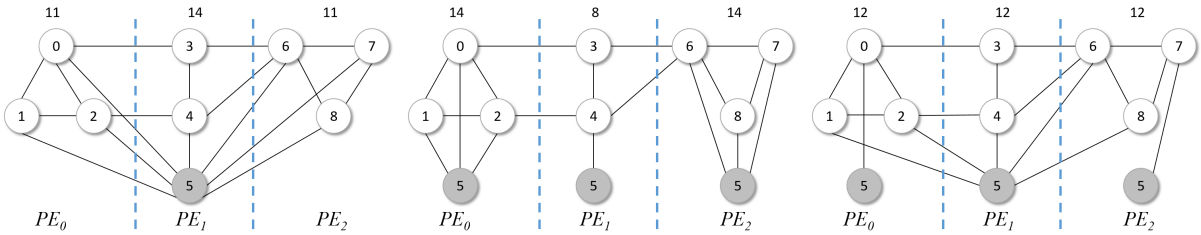


Fig. 2. (a): A graph is partitioned on three processors using 1D partitioning, where v_5 is a high-degree vertex, and PE_1 has higher workload and communication overhead. (b): We duplicate v_5 on all processors and the edge numbers on the processors are 14, 8, and 14, respectively. (c): We reassign the edges whose source vertex is v_5, and balance the final load partition, where the edge number on each processor is 12.

delegate partitioning is that vertices with degrees greater than a threshold are duplicated and distributed on all processors, while a basic 1D partitioning is applied to low-degree vertices. 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. Therefore, a similar number of edges on each processor can be assured.

For an input 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 detect high-degree vertices based on a threshold d_{high}, and duplicate them on all processors. Accordingly, the edge set E is partitioned into two subsets: E_{high} (whose source vertex degrees are greater than or equal to d_{high}) 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 vertex set on each processor includes the duplicated high-degree vertices, as well as the low-degree vertices partitioned by the traditional 1D partitioning.

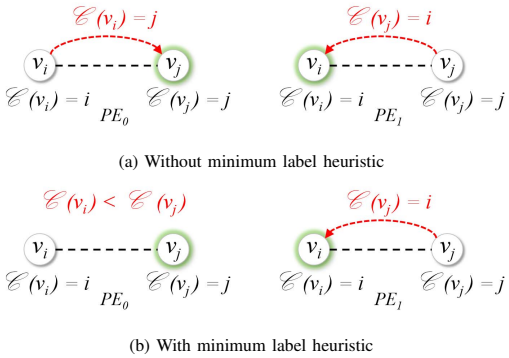
Second, we define a round-robin 1D partitioning, where we partition the edges in E_{low} according to their source vertex partitioning mapping, and partition the edges in E_{high} according to their target vertex partitioning.

Third, 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 $\frac{|E|}{p}$. However, this may not be gained through the first two 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 number of edges is less than $\frac{|E|}{p}$. Pearce et al. [8] categorized the processors into the master and the workers, and differentiated the delegates among them. In our work, we do not differentiate the delegates among the processors.

Figure 2 shows an example of this delegate partitioning, where a graph G is partitioned on three processors.

C. Heuristic for Convergence

In the original sequential Louvain algorithm, a vertex can always move to a community according to the greedy strategy. However, this case may not be always held in a distributed environment, as each processor concurrently updates its local modularity gain based on the latest snapshot information of its local subgraph. There can exist some edges across two processors (i.e., the two endpoints of such an edge belonging to different processors). In this case, swapping the community information of these two vertices can delay the convergence of the parallel algorithm.



(a) Without minimum label heuristic

(b) With minimum label heuristic

Fig. 3. Examples without and with the minimum label heuristic strategy using two vertices and two processors. (a): The vertices v_i and v_j are supposed to be in the same community. However, after swapping community information, they are still in the different communities. (b): Applying the minimum label strategy, a vertex is only allowed to move into the community with a smaller community ID. In this way, v_i and v_j can have a consistent community.

We show a simple example of this problem in Figure 3(a), where two vertices v_i and v_j are the endpoints of an edge. They are located on two different processors, PE_0 and PE_1. On PE_0, the vertex v_i is the local vertex and the vertex v_j is the ghost vertex, and vice versa on PE_1. On each processor, a vertex with a green circle denotes a ghost vertex.

Initially, a vertex is in its own community of size one and the community ID is the vertex ID, i.e., $\mathscr{C}(v_i) = i$ and $\mathscr{C}(v_j) = j$, where the function $\mathscr{C}$ denotes the community ID of a vertex or a community. After the local modularity gain calculation using Equation 4, we can easily see that both vertices move to each other's community on their local processors to increase the local modularity gain, i.e., $\mathscr{C}(v_i) = j$ on PE_0 and $\mathscr{C}(v_j) = i$ on PE_1, as shown on the red dash arrows in Figure 3(a). This, however, cannot gain any modularity for the global graph. This

phenomenon can incur the vertex bouncing problem between two different communities, and thus inhabit the convergence of the algorithm.

Lu et al. [14] proposed a minimum label heuristic to address this problem for shared-memory architectures. Given two candidate communities with the same modularity gain, this strategy only allows a vertex to be moved into a community with the smaller community ID, and thus prevents the bouncing problem. In Lu et al.'s work, this strategy is applied on each edge of a graph, where edges are concurrently processed by multiple threads. In our case, we only need to consider the minimum label heuristic for those edges connecting local vertices and ghost vertices. Figure 3(b) shows the community states of both vertices by applying the minimum label strategy. We assume that $\mathscr{C}(v_i) < \mathscr{C}(v_j)$ (i.e., $i < j$). Thus, the vertex v_j moves to the community of v_i (i.e., $\mathscr{C}(v_j) = i$) on PE_1 while v_i remains in its own community $\mathscr{C}(v_i)$ on PE_0.

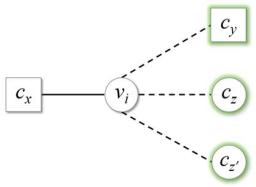


Fig. 4. A complex case of moving a local vertex v_i to one of the communities c_x, c_y, c_z, and $c_{z'}$. The circles highlighted in green, c_z and $c_{z'}$, are the singleton communities, and each only contains one ghost vertex. The square c_x is the local community on the same processor as v_i, and can contain one or more vertices. The square highlighted in green c_y is the community belonging to other processor and can contain one or more vertices.

However, this heuristic is not sufficient for more complex cases where a vertex connects to multiple communities that have different structures in a distributed environment. Consider the case in Figure 4, where we would like to move a local vertex v_i to one of the communities c_x, c_y, c_z, and $c_{z'}$. c_z and $c_{z'}$ are the singleton communities, and each only contains one ghost vertex. c_x is the local community on the same processor as v_i, and can contain one or more vertices. c_y is the community belonging to other processor, and can contain one or more vertices. In this example, we assume that $\delta Q_{v_i \to c_x} = \delta Q_{v_i \to c_y} = \delta Q_{v_i \to c_z} = \delta Q_{v_i \to c_{z'}}$, and c_z has the minimum community label. According to the minimum label heuristic, the vertex v_i should be moved to the community c_z. In their work, Lu et al. [14] mentioned that this may only delay the convergence, but will not affect consistency with the sequential algorithm, because in a shared memory architecture this community update can be easily communicated among the processing threads or cores in the shared memory. However, in a distributed environment, because each processor concurrently updates their local communities in their own memory, the community updated information cannot be broadcasted immediately during the process of local clustering. As the community c_z only contains one vertex, it can be moved to other communities on its own processor, and this information cannot be instantly sent to the other processors. Therefore, the modularity gain of moving vertex v_i to the community c_z can lead to an inefficient or wrong movement, due to the unawareness of the update of c_z on its own processor. This impairs not only the convergence but also the consistent results compared to the sequential Louvain algorithm.

To determine the movement of the vertex v_i in this more complex (and also more general) situation where v_i connects to multiple communities, we enhance the simple heuristic strategy. For a simple situation where only one of the δQ values is maximum for a vertex, we can move it to the community with the maximum δQ value without using any heuristic. For a more complex situation where there are several communities who have the same δQ for a vertex, we develop a new heuristic to make sure the vertex can move to a local community. The following cases are considered in our heuristic with the example in Figure 4:

- When the δQ values of all the communities are equal for a vertex, we move the vertex to a local community: If $\delta Q_{v_i \to c_x} = \delta Q_{v_i \to c_y} = \delta Q_{v_i \to c_z} = \delta Q_{v_i \to c_{z'}}$, then $\mathscr{C}(v_i) = \mathscr{C}(c_x)$.
- When the δQ values of the ghost communities are equal, but larger than the δQ values of the local communities for a vertex, we move the vertex to the ghost community with more than one vertex: If $\delta Q_{v_i \to c_y} = \delta Q_{v_i \to c_z} = \delta Q_{v_i \to c_{z'}} > \delta Q_{v_i \to c_x}$, then $\mathscr{C}(v_i) = \mathscr{C}(c_y)$.
- When the δQ values of the singleton ghost communities are equal, but larger than the δQ values of the other communities for a vertex, we move the vertex to the singleton ghost community with the minimal label: If $\delta Q_{v_i \to c_z} = \delta Q_{v_i \to c_{z'}} > \delta Q_{v_i \to c_x}, \delta Q_{v_i \to c_y}$, then $\mathscr{C}(v_i) = min(\mathscr{C}(c_z), \mathscr{C}(c_{z'}))$.

In this enhanced strategy, we prefer to first move v_i to local communities, because the community information can be updated immediately on the local processor. Otherwise, we attempt to move v_i to a ghost community with multiple vertices. Although the ghost community can be updated on its own processor, likely the vertices of this ghost community cannot be entirely changed. Finally, if only moving vertex v_i to singleton ghost communities can archive the maximum modularity gain, the minimum label strategy is applied. In this way, we can overcome the disadvantage of the simple minimum label strategy and achieve the convergence of our parallel algorithm and the consistency with the result of sequential algorithm, which will be evaluated and demonstrated in Section V-A.

D. Parallel Local Clustering

Algorithm 2 shows the details of our parallel local clustering of a subgraph $G_s = (V_s, E_s)$ on a processor PE_i, where V_s is the vertex set and E_s is the edge set of the subgraph G_s. For simplicity, we only show the algorithm with delegated vertices. For the case without delegates, the only modification is not to classify the vertex set V_s into V_{low} and V_{high}, where V_{low} is the set of low-degree vertices and V_{high} is the set of global high-degree vertices (i.e., hubs).

The algorithm first initializes each vertex as a unique community, described as Lines 2 to 7. Then, the algorithm calculates the modularity gain for each vertex $u \in V_{low} \cup V_{high}$, and

Algorithm 2 Parallel Local Clustering

Require:
$G_s = (V_s, E_s)$: undirected subgraph, where V_s is vertex set and E_s is the edge set;
$V_s = V_{low} \cup V_{high}$: subgraph vertex set, where V_{low} is low-degree vertices and V_{high} is global high-degree vertices;
C_s^0: initial community of G_s^0;
θ: modularity gain threshold;
PE_i: local processor.

Ensure:
C_{PE_i}: local resulting community;
Q_{PE_i}: local resulting modularity;
Q: global resulting modularity.

1: $k = 0$ // k indicates the inner iteration number
2: **for all** $u \in V_s^k$ **do**
3: Set $C_u^k = u$
4: Set $m_u = 0$
5: Set $\sum_{in}^{C_u^k} = w_{u,u}, (u,u) \in E^k$
6: Set $\sum_{tot}^{C_u^k} = w_{u,v}, (u,v) \in E^k$
7: **end for**
8: **repeat**
9: **for all** $u \in V_{low} \cup V_{high}$ **do**
10: **if** $Cu^{k'} = argmax(\delta Q_{C_u^k \to C_u^{k'}}) > m_u$ **then**
11: $\mathscr{C}(u) = min(\mathscr{C}(C_u^{k'}), \mathscr{C}(C_u^k))$
12: **end if**
13: **end for**
14: Synchronize V_{high} states
15: Synchronize ghost vertex community states
16: **for all** $u \in V_{low} \cup V_{high}$ **do**
17: $\sum_{tot}^{C_u^k} = \sum_{tot}^{C_u^k} - w(u)$; $\sum_{in}^{C_u^k} = \sum_{in}^{C_u^k} - w_{u \to C_u^k}$
18: $\sum_{tot}^{C_u^{k'}} = \sum_{tot}^{C_u^{k'}} + w(u)$; $\sum_{in}^{C_u^{k'}} = \sum_{in}^{C_u^{k'}} + w_{u \to C_u^{k'}}$
19: **end for**
20: //Calculate partial modularity
21: $Q_{PE_i} = 0$
22: **for all** $c \in C_{PE_i}$ **do**
23: $Q_{PE_i} = Q_{PE_i} + \frac{\sum_{in}^{C_u^k}}{2m} - (\frac{\sum_{tot}^{C_u^k}}{2m})^2$
24: **end for**
25: $Q = Allreduce(Q_{PE_i})$
26: $k = k + 1$
27: **until** No modularity improvement

Algorithm 3 Distributed Graph Merging

Require:
$G_s = (V_s, E_s)$: undirected subgraph, where V_s is vertex set and E_s is the edge set;
$V_s = V_{low} \cup V_{high}$: the vertex set of subgraph, where V_{low} is low-degree vertices and V_{high} is global high-degree vertices;
PE_i: local processor.

Ensure:
$G_s^{new} = (V_s^{new}, E_s^{new})$: undirected subgraph, where V_s^{new} is vertex set containing all vertices belonged to PE_i and E_s^{new} is the edge set whose source vertex is in V_s^{new}

1: **for all** $u \in V_s^k$ **do**
2: Send u to PE_j (u belonged to PE_j)
3: **end for**
4: **for all** $(u,v) \in E_s^k$ **do**
5: Send (u,v) and $w_{u,v}$ to PE_j (u belonged to PE_j)
6: **end for**
7: **for all** u received **do**
8: Insert u into V_s^{new}
9: **end for**
10: **for all** (u,v) and $w_{u,v}$ received **do**
11: Insert (u,v) and $w_{u,v}$ into E_s^{new}
12: **end for**

updates the state information of the vertex u according to our modified minimum label heuristic, as described from Lines 9 to 13. From Lines 14 to 19, the algorithm communicates the community information of delegated vertices and ghost vertices across the processors, and then updates the community information on the local processor. From Lines 21 to 24, the algorithm calculates a partial modularity according to the community information on the local processor. Through the MPI *Allreduce* operation in Line 25, each processor acquires a global modularity value.

From Lines 9 and 16 in Algorithm 2, we can clearly see that the execution time of parallel local clustering is largely proportional to the size of $V_{low} \cup V_{high}$. Our distributed delegate partitioning (Section IV-B) ensures that each processor is assigned a similar number of low-degree vertices and high-degree vertices. Therefore, our parallel local clustering can achieve well balanced workload among processors. Although the synchronization operation has been used, its cost is marginal because of a limited number of hubs. In Section V, we will show the detailed performance evaluation results to demonstrate the workload balancing and the scalability of our algorithm.

E. 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 new merged graph. Then, the processor sends the information of the new vertices and their adjacent edges to their belonging processors according to the 1D partitioning. Algorithm 3 shows the steps for constructing the merged graph. From Lines 1 to 6, the algorithm sends the local vertices and their adjacent edges to the related processors. From Lines 7 to 12, each processor receives the vertices and edges, and builds the subgraphs.

V. EXPERIMENTS

We show the experimental results of our distributed Louvain algorithm. We first evaluate the community detection results, including the convergence of our algorithm and the quality of communities detected. Then, we show the performance of

TABLE I
DATASETS.

Name	Description	#Vertices	#Edges
Amazon [22]	Frequently co-purchased products from Amazon	0.34M	0.93M
DBLP [22]	A co-authorship network from DBLP	0.32M	1.05M
ND-Web [23]	A web network of University of Notre Dame	0.33M	1.50M
YouTube [22]	YouTube friendship network	1.13M	2.99M
LiveJournal [24]	A virtual-community social site	3.99M	34.68M
UK-2005 [25]	Web crawl of the .uk domain in 2005	39.36M	936.36M
WebBase-2001 [24]	A crawl graph by WebBase	118.14M	1.01B
Friendster [25]	An on-line gaming network	65.61M	1.81B
UK-2007 [25]	Web crawl of the .uk domain in 2007	105.9M	3.78B
LFR [26]	A synthetic graph with built-in community structure	0.1M	1.6M
R-MAT [27]	A R-MAT graph satisfying Graph 500 specification	2^{SCALE}	$2^{SCALE+4}$
BA [28]	A synthetic scale-free graph based on Barabasi-Albert model	2^{SCALE}	$2^{SCALE+4}$

our algorithm, including the analysis of the partitioning, the execution time breakdown, and the scalability.

Table I summarizes the datasets used in our experiment. In particular, we use three large real-world datasets (i.e., WebBase-2001, Friendster, and UK-2007) that all contain more than 1 billion edges. In addition, there are more than 3 billion edges in UK-2007. Besides the real-world graphs, we also use R-MAT [27] and BA (Barabasi-Albert) [28] to generate large synthetic datasets. We implemented our algorithm using MPI and C++, and employed Titan, a supercomputer operated by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, in our experiment. We set the threshold d_{high} as the number of processors to determine high-degree vertices (i.e., hubs).

A. Community Result Analysis

We first compare the convergence of our parallel algorithm to the sequential algorithm for all the datasets. Figure 5 shows the selected results with six datasets, and similar results have been obtained on the other datasets. We observe that our parallel Louvain algorithm with our enhanced heuristic can achieve a converged modularity close to the sequential Louvain algorithm. However, with the simple minimum label heuristic proposed by Lu et al. [14], the modularity values of the final results are significantly lower than the sequential algorithm and our enhanced heuristic. For example, for the DBLP dataset, the modularity of the simple heuristic is 0.57, while the ones of the sequential algorithm and our enhanced heuristic are 0.80 and 0.82, respectively. For the Amazon dataset, the modularity values are 0.53, 0.89, and 0.93 for the simple heuristic, the sequential algorithm, and our solution, respectively. This shows that our solution has successfully addressed the challenging convergence problem when involving delegates in parallel community detection. Our heuristic can effectively suppress the vertex bouncing issue, which is non-trivial when the communities of multiple vertices (in particular, ghost vertices) are determined concurrently across processors.

We also note that sometimes our parallel algorithm needs one more iteration than the original sequential algorithm. We suspect that in each iteration, our parallel algorithm may compute a slightly smaller modularity gain value compared to the sequential algorithm. Such a possible smaller difference may be accumulated to increase the iteration number of our algorithm to be converged. We will investigate this phenomenon in our future work.

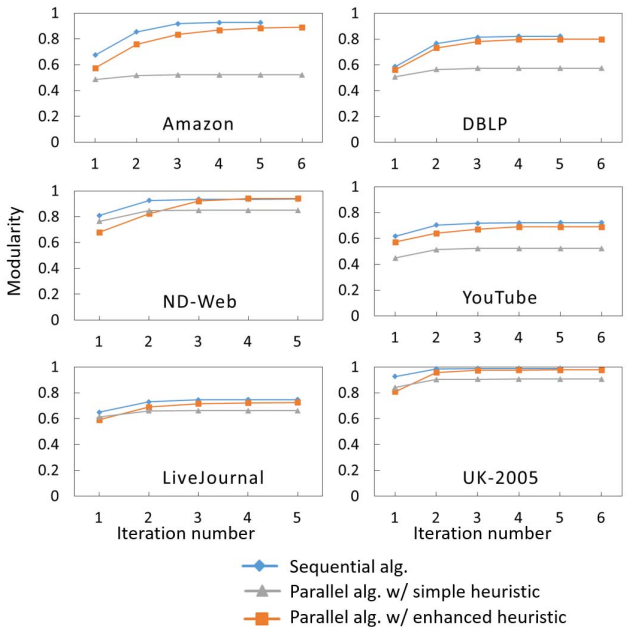


Fig. 5. Comparison of modularity convergence between the sequential Louvain algorithm, our parallel Louvain algorithm using the simple minimum label heuristic, and our parallel Louvain algorithm using the enhanced heuristic.

Apart from modularity, 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, Normalized Van Dongen metric (NVD), Rand Index (RI), Adjusted Rand Index (ARI) and Jaccard Index (JI). Among all these measurements, except NVD, a high value corresponds to a high quality [29]. Table II shows the results for two datasets. For the most important measurement NMI, their values are all above 0.80, indicating a high quality of community detection [29].

TABLE II
QUALITY MEASUREMENTS.

Dataset	NMI	F-measure	NVD	RI	ARI	JI
ND-Web	0.8021	0.8111	0.2640	0.9688	0.6039	0.6651
Amazon	0.8455	0.8075	0.1678	0.9733	0.6887	0.8432

B. Workload and Communication Analysis

Most existing distributed Louvain algorithms employ the simple 1D partitioning strategy, and thus suffer from accuracy loss [15], preprocessing overhead [16], or imbalanced inter-processor communication [18]. One unique feature of our parallel Louvain algorithm is the usage of delegate partitioning strategy that facilitates us to address the issues incurred in the existing work. We investigate clustering workload and communication costs affected by the 1D partitioning and our delegate partitioning.

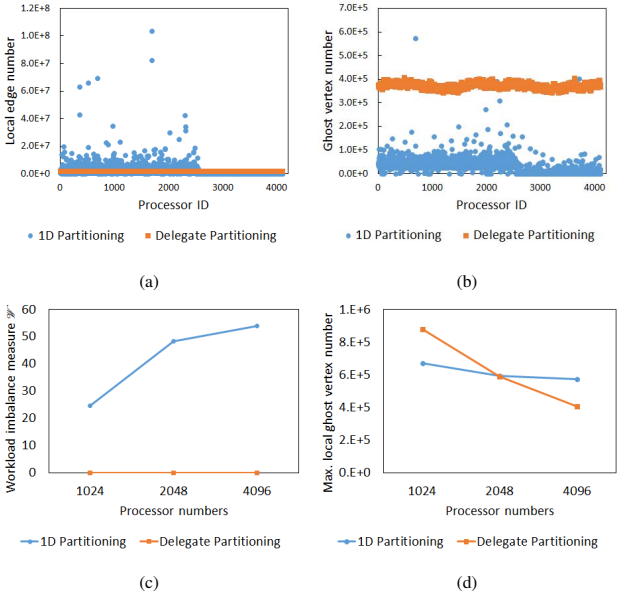


Fig. 6. Comparison of workload and communication costs using the 1D partitioning and our delegate partitioning with the UK-2007 dataset: (a) the workload of each of 4096 processors; (b) the local ghost vertex number of each of 4096 processors; (c) the workload imbalance on 1024, 2048 and 4096 processors; (d) the maximum local ghost vertex number among 1024, 2048 and 4096 processors.

Figure 6 shows the results using the large real-world dataset UK-2007. We first investigate clustering workload affected by different partitioning methods. Figure 6(a) shows the local edge number on each of 4096 processors, which is related to the local clustering running time and can be used to estimate workload of clustering on each processor. We can clearly see that delegate partitioning can make each processor have similar number of edges. With 1D partitioning, the maximum local edge number can be two orders larger than that in delegate partitioning.

We also explore communication costs affected by the 1D partitioning and our delegate partitioning. Figure 6(b) compares the local ghost vertex number of each of 4096 processors using these two partitioning methods. Communication cost correlates with ghost vertex number in the parallel algorithm. With 1D partitioning, there are several processor containing nearly 1 million ghost vertices, while in delegate partitioning each processor has similar ghost vertex number that is about 0.4 million. This indicates that a balanced communication can be achieved using delegate partitioning. Through the analysis of delegate partitioning, we note that the algorithm distributes the edges connecting hubs and low-degree vertices evenly on different processors. That is why in delegate partitioning the ghost vertex number on each processor can be larger than that on some processor using the 1D partitioning. However, this will not make the communication as a performance bottleneck. We will show this in Section V-C.

We further compare these two partitioning methods with an increasing number of processors. For each processor number,

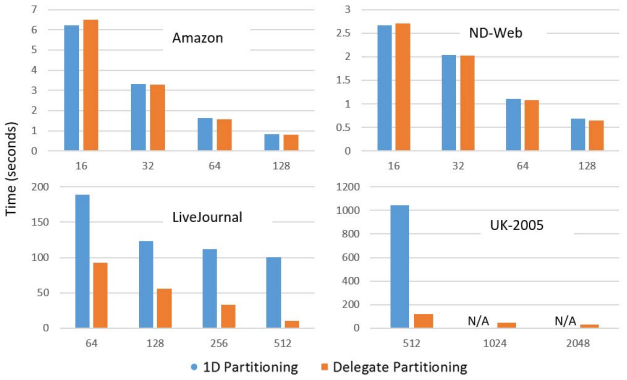


Fig. 7. Running time comparison between our approach and an existing distributed Louvain algorithm with the 1D partitioning [15].

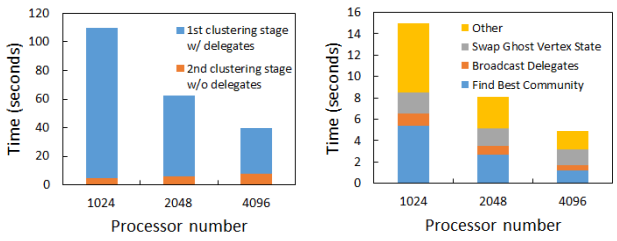
we define a measure of workload imbalance $\mathscr{W}$ as

$$\mathscr{W} = \frac{|E_{max}|}{|E_{avg}|} - 1, \tag{5}$$

where $|E_{max}|$ denotes the maximum local edge number among processors, and $|E_{avg}|$ denotes the average local edge number among processors. Thus, $\mathscr{W}$ expresses the folds that the maximum workload is more than the average workload. Figure 6(c) shows that the results of workload imbalance measure $\mathscr{W}$. We can see that the use of the 1D partitioning can increase the workload imbalance among processors when using more processors, while the workload imbalance is close to zero with our delegate partitioning. This shows that our delegate partitioning can achieve a more scalable performance than the 1D partitioning, which will be verified in Section V-D.

Figure 6(d) shows that the maximum local ghost vertex number among processors with different numbers of processors. With an increasing number of processors, the maximum local ghost vertex number using the 1D partitioning is reduced slightly. This indicates that the communication cost can remain the same with more processors. Our delegate partitioning can effectively reduce the maximum local ghost vertex number when increasing the processor number. Although the maximum local ghost vertex of our method can be higher than that of the 1D partition with a fewer number of processors, our method can reduce the ghost vertices in a scalable manner, leading to more scalable communication performance.

Through this analysis, we observe that the delegate partitioning can significantly balance workload and communication, and thereby fundamentally ensure that our parallel algorithm can outperform the distributed Louvain algorithms simply based on the 1D partitioning [15], [16], [18]. Among these existing work, we implemented an MPI version of Cheong's work [15] using the 1D partitioning. In Figure 7, we compare the total running time between our approach and the distribution implementation of Cheong's work using the 1D partitioning. It clearly shows the advantage of the delegate partitioning. In order to compare how much this workload balance can affect the total running time, we compare the running time between our method and the distributed Louvain



(a) (b)

Fig. 8. (a): Running times of the first and second clustering stages for UK-2007 on 1024, 2048 and 4096 processors. (b): Time breakdown of one inner clustering iteration using delegates for UK-2007 on 1024, 2048 and 4096 processors.

algorithm with 1D partitioning. As shown in Figure 7, we can find that for smaller datasets (e.g., the Amazon and ND-Web datasets), even though there is some workload difference, the total running time is similar. However, with the size of dataset increasing, the running time of distributed Louvain using the 1D partitioning also increase. In particular, on UK-2005, the simple distributed algorithm with the 1D partitioning cannot complete when using 1024 processors or more. This is because one processor has much more workload than others and needs more time for local clustering and swapping ghosts.

C. Execution Time Breakdown

We examine the main performance components of our algorithm using large real-world datasets. Figure 8(a) shows the total clustering time of our algorithm using the UK-2007 dataset, including the times of the two clustering stages (i.e., the first parallel local clustering stage with delegates, and the second parallel local clustering stage without delegates). For simplicity, the graph merging time is included in the clustering time. There is only one step in the first parallel local clustering stage with delegates, while there can be multiple steps in graph clustering and merging in the second parallel local clustering stage without delegates. As we can see, the first clustering stage uses most of the running time. After this stage, the original graph is merged into a new graph that can be several order smaller than the original graph. Therefore, the running time of the section clustering stage becomes much shorter.

As we previously stated, in the first clustering stage with delegates, there can be multiple iterations to calculate modularity gain of each vertex and swap community information. Therefore, in Figure 8(b), we show one iteration of clustering with delegates on each processor. For each iteration, our clustering contains four parts, which are *Find Best Community*, *Broadcast Delegates*, *Swap Ghost Vertex State*, and Other. In *Find Best Community*, the algorithm calculates the modularity gain for each vertex. After this step, each processor uses *Broadcast Delegates* to broadcast the community information of those vertex delegates with the maximum modularity gain, and uses *Swap Ghost Vertex State* to send the community info of vertices who are the ghost vertices on other processors. As we can see, the collective operation for all delegated vertices is just a small portion of each iteration. The *Other* part mainly

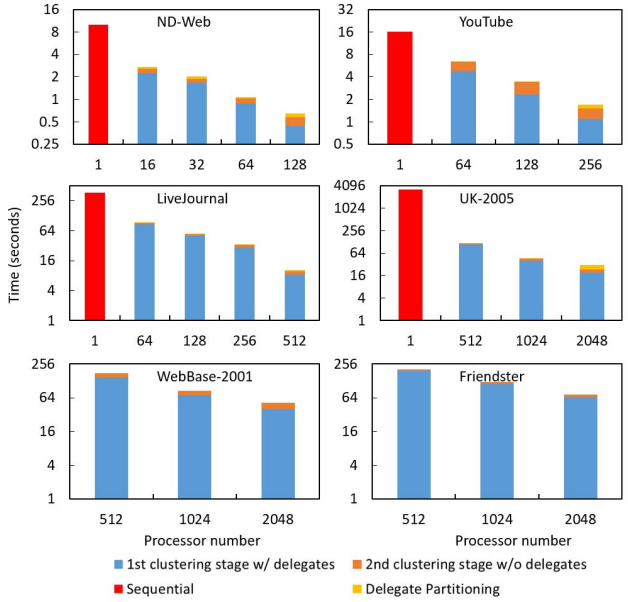


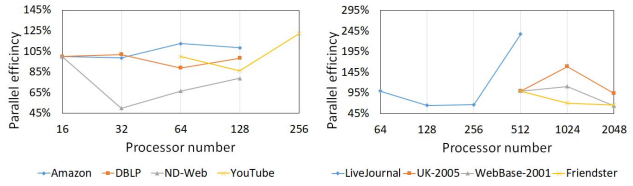
Fig. 9. Scalability study of our algorithm using different datasets and different processor numbers. Our total clustering time contains the times of the first clustering stage with delegates and the second clustering stage without delegates. Note that each plot is in a logarithmic scale. The results clearly show the scalable performance of our parallel clustering algorithm with the different sizes of datasets.

updates the information of communities, such as $\sum_{in}$ and $\sum_{tot}$, which are responsible for computing the temporal partial modularity of each subgraph and using the MPI *Allreduce* to calculate the global modularity.

Unsurprisingly, in Figure 8(b), we can notice that the time of *Find Best Community*, *Broadcast Delegates* and *Other* are reduced with the increasing number of processor number. The time of *Find Best Community* is related with the workload on each processor. With delegates partitioning, we can evenly partition the workload among processors. With the increasing number of processors, the number of high-degree vertices is decreasing, and thus the time of *Broadcast Delegates* can also be decreased. While in *Other* part, our algorithm mainly updates local community information for calculating temporal partial modularity, which is related with the number of local communities. For *Swap Ghost Vertex State*, 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 numbers of ghost vertex on the different number of processors are still the same order (Figure 6(d)) and all these ghost vertices need to be swapped in each iteration.

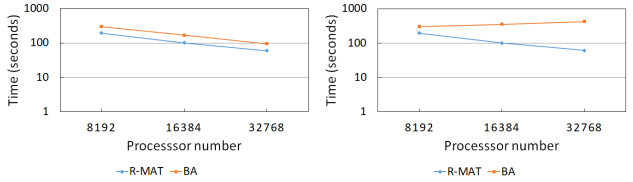
D. Scalability Analysis

We examine the parallel performance of our algorithm on the real-world datasets of different sizes. Besides the small real-world datasets, such as ND-Web and YouTube, we also show the performance on large real-world datasets, including LiveJournal (medium size), UK-2005 (large size), WebBase-



(a) (b)

Fig. 10. Parallel efficiency of our algorithm on (a) the Amazon, DBLP, ND-Web, and YouTube datasets; (b) the LiveJournal, UK-2005, WebBase-2001, and Friendster datasets.



(a) (b)

Fig. 11. (a): Strong scaling of the clustering time of our algorithm with up to 32768 processors using the R-MAT and BA with the global vertex size of 2^{30}. (b): Weak scaling of the clustering time of our algorithm with up to 32768 processors using the R-MAT and BA with the global vertex sizes of 2^{29}, 2^{30}, and 2^{31} on each processor from left to right.

2001 (large size), Friendster (large size), and UK-2007 (very large size). As our algorithm first clusters subgraphs with delegates and then clusters merged graphs without delegates, the running time of our algorithm is mainly the total time of these two clustering stages. Figure 9 shows the running times of our parallel algorithm on the real-world datasets. Our parallel clustering algorithm achieves a scalable performance with the different sizes of datasets.

Figure 9 also shows the sequential time on all datasets except WebBase-2001 and Friendster, because the sequential time on these two datasets are too long. We can easily see the unsustainable running time of the sequential Louvain algorithm with the increasing graph size. We also see that the delegate partitioning time is almost negligible for all 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:

$$\tau = \frac{p_1 T(p_1)}{p_2 T(p_2)}, \tag{6}$$

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(a) and (b) show the parallel efficiency of our algorithm for the small real-world datasets and the large real-world datasets. We find that in most cases, our parallel algorithm can achieve more than 65% parallel efficiency and in some cases it can even achieve more than 100% efficiency. We investigated our parallel clustering on the LiveJournal dataset, and found that on 512 processors our parallel clustering has a fewer (around 50% less) clustering iteration number than that on 256 processors. This explains that the parallel efficiency is more than 100% in this case. Similar observations were also obtained for the WebBase-2001 and UK-2005 datasets on 512 and 1024 processors, where the parallel efficiency is over 100%. In addition, Figure 9 shows the first clustering stage with delegates is the dominant part and its running time 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.

In order to further examine the scalability of our algorithm, we also use R-MAT [27] and BA [28] to generate synthetic datasets. As shown in Figure 11(a), we test the strong scaling of our algorithm, where we set the vertex scale to 30 and the edge scale is 34. For R-MAT, the clustering time was reduced from 194.15 seconds to 60.32 seconds when we increased the processor number for 8192 to 32768. For BA, the clustering time was reduced from 302.16 seconds to 95.45 seconds. We can see that even using these very large synthetic datasets, our distributed clustering algorithm can still achieve around 80% parallel efficiency with up to 32768 processors.

Figure 11(b) shows the weak scaling results. We use R-MAT and BA synthetic graphs, the vertex scale is set to 20 on each computing node, therefore vertex scale is 16 on each processor (each computing node contains 16 processors). From 8192 to 16384 to 32768 processors, the global graph vertex sizes are 2^{29}, 2^{30}, and 2^{31}, respectively. We notice that for R-MAT graphs, our algorithm has a negative slope of weak scaling. We investigated it and inferred that R-MAT graphs are not entirely scale-free, and our algorithm uses less iteration to converge although the graph size is increasing. For BA graphs, our algorithm maintains a nearly horizontal line.

These experimental results showed that our method can achieve balanced computation workload and communication among massive processors using large graphs, which clearly demonstrated an improved scalability over the previous state-of-the-art using the 1D partitioning [15], [16], [18].

VI. Conclusion

In this paper, we present a new and scalable distributed Louvain algorithm. Using the vertex delegates partitioning, our algorithm can achieve balanced workload and communication among massive processors with large graphs. Moreover, we develop a collective operation to synchronize the information on delegated vertices, and design a new heuristic strategy to ensure the convergence of the algorithm. Through intensive experiments, we analyze and compare the correctness, the workload, and the communication of our algorithm to the other methods. Our algorithm clearly shows a superior scalability and accuracy over the existing distributed Louvain algorithms.

In the future, we would like to further accelerate our parallel community detection approach by leveraging the power of graphics processing units (GPUs). In our current CPU-based implementation, the communication cost is comparably smaller than the computational cost of clustering. Although it appears a less difficult task to use GPUs to parallelize local clustering on a single compute node, inter-processor

communication cost can possibly become a major performance bottleneck when the GPU-based clustering time can be significantly reduced. To this end, we plan to improve our graph partitioning method and investigate possible ways to further reduce the communication cost.

ACKNOWLEDGMENT

This research has been sponsored by the National Science Foundation through grants IIS-1423487, IIS-1652846, and ICER-1541043.

REFERENCES

[1] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[2] S. Harenberg, G. Bello, L. Gjeltema, S. Ranshous, J. Harlalka, R. Seay, K. Padmanabhan, and N. Samatova, "Community detection in large-scale networks: a survey and empirical evaluation," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 6, no. 6, pp. 426–439, 2014.

[3] K. D. Devine, E. G. Boman, and G. Karypis, "Partitioning and load balancing for emerging parallel applications and architectures," *Parallel Processing for Scientific Computing*, vol. 20, p. 99, 2006.

[4] A. Y. Ng, M. I. Jordan, Y. Weiss *et al.*, "On spectral clustering: Analysis and an algorithm," *Advances in neural information processing systems*, vol. 2, pp. 849–856, 2002.

[5] Y.-W. Huang, N. Jing, and E. A. Rundensteiner, "Effective graph clustering for path queries in digital map databases," in *Proceedings of the fifth international conference on Information and knowledge management*. ACM, 1996, pp. 215–222.

[6] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, "Rabbit order: Just-in-time parallel reordering for fast graph analysis," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 22–31.

[7] K. Choromański, M. Matuszak, and J. Miękisz, "Scale-free graph with preferential attachment and evolving internal vertex structure," *Journal of Statistical Physics*, vol. 151, no. 6, pp. 1175–1183, 2013.

[8] R. Pearce, M. Gokhale, and N. M. Amato, "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, 2014, pp. 549–559.

[9] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, no. 6, p. 066111, Dec. 2004.

[10] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 10, p. 8, Oct. 2008.

[11] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *Parallel Processing and Applied Mathematics*. Springer, 2012, pp. 286–296.

[12] J. Xie, B. K. Szymanski, and X. Liu, "SLPA: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process," in *2011 IEEE 11th International Conference on Data Mining Workshops*. IEEE, 2011, pp. 344–349.

[13] S. Bhowmick and S. Srinivasan, "A template for parallelizing the Louvain method for modularity maximization," in *Dynamics On and Of Complex Networks, Volume 2*. Springer, 2013, pp. 111–124.

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

[15] C. Y. Cheong, H. P. Huynh, D. Lo, and R. S. M. Goh, "Hierarchical parallel algorithm for modularity-based community detection using GPUs," in *Proceedings of the 19th International Conference on Parallel Processing*, ser. Euro-Par'13, 2013, pp. 775–787.

[16] J. Zeng and H. Yu, "Parallel modularity-based community detection on large-scale graphs," in *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, Sept 2015, pp. 1–10.

[17] ——, "A study of graph partitioning schemes for parallel graph community detection," *Parallel Computing*, vol. 58, pp. 131–139, 2016.

[18] X. Que, F. Checconi, F. Petrini, and J. A. Gunnels, "Scalable community detection with the Louvain algorithm," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 28–37.

[19] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[20] T. Raeder and N. V. Chawla, "Market basket analysis with networks," *Social network analysis and mining*, vol. 1, no. 2, pp. 97–113, 2011.

[21] D. Meunier, R. Lambiotte, A. Fornito, K. D. Ersche, and E. T. Bullmore, "Hierarchical modularity in human brain functional networks," *Hierarchy and dynamics in neural networks*, vol. 1, p. 2, 2010.

[22] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, ser. MDS '12, 2012, pp. 3:1–3:8.

[23] R. Albert, H. Jeong, and A.-L. Barabási, "Internet: Diameter of the world-wide web," *nature*, vol. 401, no. 6749, pp. 130–131, 1999.

[24] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, 2004, pp. 595–601.

[25] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[26] A. Lancichinetti and S. Fortunato, "Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities," *Physical Review E*, vol. 80, no. 1, p. 016118, 2009.

[27] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

[28] B. Machta and J. Machta, "Parallel dynamics and computational complexity of network growth models," *Physical Review E*, vol. 71, no. 2, p. 026704, 2005.

[29] J. Xie, S. Kelley, and B. K. Szymanski, "Overlapping community detection in networks: The state-of-the-art and comparative study," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 43:1–43:35, Aug. 2013.