Effectively Unified Optimization for Large-scale Graph Community Detection

Jianping Zeng, Hongfeng Yu
Department of Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, NE, USA

Abstract—In this paper, we present a unified graph clustering framework based on an asynchronous approach. We study the similarities among the Louvain algorithm and the Infomap algorithm. Based on their common features, we build an end-to-end optimized distributed framework for implementing both algorithms. By extending the existing asynchronous distributed framework for large-scale graphs traversal, we ensure both workload and communication balanced. Our extensive experiments show that our framework is correct and effective with different large real-world and synthetic datasets using up to 32,768 processors for the Louvain algorithm and 16,384 processors for the Infomap algorithm. The quality and the scalability of our framework are superior to the existing work.

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

I. INTRODUCTION

Community detection, or graph clustering, is the problem of clustering nodes (or vertices) of a graph into different communities or modules. Although there is no rigorous definition of community structure, generally an individual community has dense intra-connections but sparse inter-connections among them. Various algorithms have been proposed based on different quality measurements of detected communities. Among these algorithms, the Louvain algorithm [1] and the Infomap algorithm [2] are two representative methods of agglomerative community detection. The Louvain algorithm is based on the modularity metric, while the Infomap algorithm adopts the map equation. Both algorithms use a greedy approach that optimizes the measurements by iteratively moving nodes between communities. Once a sufficiently stable solution is obtained, the communities are merged to form a new graph on which the progress is repeated. Compared with the Infomap algorithm, the Louvain algorithm is relatively fast, but the quality of detected results is less accurate [3].

It remains an open and challenging problem to develop a scalable distributed community detection algorithm. This is largely because real-world large graphs are typically scale-free graphs, where the vertex degree distribution of such a graph follows a power law distribution. The existence of high-degree vertices (or named as hubs) makes it difficult to evenly partition and distribution such a graph among processors. The processors assigned with hubs are often associated with high workload and communication overheads. In addition, it is non-trivial to effectively synchronize the hub information among processors, which can impair the accuracy of final community detection results.

To address this issue, in our previous distributed Louvain and Infomap algorithms [4], [5], we exploited delegate partitioning to reduce workload and communication imbalance to boost scalability, where delegates are duplicated hubs among processors. However, we still found that our results were not entirely consistent with the sequential algorithms, and we only showed the scalability of our distributed Infomap algorithm with up to 4,096 processors. We find that the synchronization strategy applied with the delegate partitioning tends to always swap information after each vertex (including a delegate) calculating a new movement using out-of-date information, which incurs a loss of accuracy. Besides, this strategy synchronizes both ghost vertices and delegates, and can thereby incur a high communication cost.

In this paper, we present new strategies to optimize graph clustering algorithms based on asynchronous visitor queue, where the updated information of a vertex can be sent immediately. Our approach can boost both the accuracy and the scalability. Based on this approach, we develop an optimized framework unifying the Louvain and Infomap algorithms with balanced workload and communication cost. We have conducted extensive experiments using different large-scale real-world and synthetic graph datasets. With the support of our new framework, we have demonstrated the effectiveness of our optimized Louvain algorithm with up to 32,768 processors and Infomap algorithm with up to 16,384 processors, which conveys significant improvements over the previous work.

II. RELATED WORK

Based on their architectures, the parallelized Louvain algorithms can be divided into shared memory methods, distributed memory methods, and GPU methods. For shared memory Louvain methods, researchers mainly focus on parallelizing computation. Staudt et al. [6] proposed an ensemble method with a parallel graph labeling approach. Lu et al. [7] adopted a graph coloring strategy as a preprocessing step for their parallel Louvain algorithm. Designing a distributed Louvain algorithm needs to consider both parallelizing computation and communication. Zeng et al. [8], [9] designed a unique graph partitioning approach to balance workload. Their algorithm used up to 16,384 cores and more than 1 billion edges of graphs. Que et al. [10] presented a distributed Louvain algorithm on Blue Gene/Q supercomputer with 8,192 nodes.
Zeng et al. [4] presented a Louvain algorithm by exploiting a
distributed delegate partitioning, and showed the correctness
and the scalability of their algorithm using up to 32,768
processors. In addition, GPU becomes a powerful tool for
implementing efficient parallel Louvain algorithms [11], [12].

There are limited approaches on parallel Infomap algo-

rithms. Bae et al. [13] proposed a shared-memory parallel
Infomap algorithm. In order to process much larger datasets,
Bae et al. [3] presented a distributed Infomap algorithm named
GossipMap. Its scalability was shown on 128 cores. Zeng et
al. [5] developed a distributed Infomap algorithm with a new
heuristic strategy to achieve the convergence of Infomap. Their
results have scaled up to 4,096 processors.

III. PRELIMINARIES

A. Graphs and Community Detection

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 an undirected unweighted graph. The community detection
problem is to find overlapping or non-overlapping vertices sets,
named communities (or modules), which contain high intra-
community flows but low inter-community flows. In this work,
we only focus on non-overlapping community detection on
undirected unweighted graphs. The existing work [2] shows
that an undirected graph can be easily transferred to a directed
graph. Therefore, our work can be easily extended to directed
graphs. The non-overlapping community set $C$ of a graph $G = (V, E)$
can be represented as:

$$
C = \{c_1, c_2, \ldots, c_n\}, \forall c_i \in C, c_i \cap c_j = \emptyset, \forall c_i, c_j \in C
$$

B. Louvain Algorithm

The Louvain algorithm uses modularity, $Q$, to measure the
quality of communities detected in a graph, which can be formulated as :

$$
Q = \sum_{c \in C} \left( \frac{\sum_{u,v \in c} w_{u,v}}{\sum_{u,v} w_{u,v}} - \frac{(\sum_{u \in c} w_u)^2}{\sum_{u,v} w_{u,v}} \right),
$$

(2)

where $m$ is the sum of all edge weights in the graph, $\sum_{c}$
is the sum of all internal edge weights in a community $c$,
calculated as $\sum_{c} = \sum w_{u,v}(u \in c \land v \in c)$, and $\sum_{c}$
is the sum of all edge weights, calculated as $\sum_{c} = \sum w_{u,v}(u \in
c \lor v \in c)$. 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, $\delta Q$, is the gain in modularity obtained
by moving an isolated vertex $u$ into a community $c \in C$ [1],
which can be computed by:

$$
\delta Q_{u \rightarrow c} = \frac{1}{m} \left( w_{u \rightarrow c} - \sum_{u \in c} w(u) \right),
$$

(3)

where $w_{u \rightarrow c}$ is the total weight of edges connecting a vertex
$u$ and a community $c$, and $w(u)$ is the weighted degree of $u$.
The Louvain algorithm iterates multiple stages for computing a
hierarchical clustering of the vertices in $G$. Each stage consists
of two phases. In this phase (named vertex movement), each
vertex is considered in turn and moved to a community with
the maximum modularity gain given by Equation 3. The vertex
will remain in its current community if no positive modularity
gain can be achieved. This indeed employs a greedy strategy to
compute graph clustering that optimizes the modularity given
by Equation 2. The algorithm continues iterations until no
further gain can be obtained or if the gain falls below some
predefined threshold. In the second phase, the communities
generated in the first phase are represented as a new graph with
vertices of each community merged into a single new vertex. If
there are multiple edges between different communities, these
coaalesce into one edge between the new vertices. The weight
of a new edge is the sum of the weights of all the edges merged
into it. The new aggregated graph is then iteratively input to
the next stage of the algorithm. This whole process continues
until the graph is stable (no modularity change).

C. Infomap Algorithm

The Infomap algorithm is a flow-based information-
theoretic method to assign vertices to communities or modules
for community detection. The algorithm is enlightened by
a duality between the problem of compressing a dataset and
the problem of detecting and extracting significant patterns or
structures within the dataset. The general description of duality
in statistics is known as minimum description length (MDL).
The Infomap algorithm aims to find the structures within a
graph that are significant with respect to how information
flow through the graph. In general, the sequential Infomap
algorithm optimizes MDL to the shortest code length. The
map equation [2] is the objective function of the Infomap
algorithm, which is based on the information flow. The map
equation finds a compressed representation of a set of random
walks through a graph. The map equation can be expressed as
in Equation 4:

$$
L(M) = \left( \sum_{m \in M} q_m \log q_m - \sum_{\alpha \in V} p_\alpha \log p_\alpha \right) + \sum_{m \in M} (q_m + \sum_{\alpha \in m} p_\alpha) \log (q_m + \sum_{\alpha \in m} p_\alpha),
$$

(4)

where $M$ is the set of communities (or modules), $q_m$ is the exit
probability of a community $m$, $p_\alpha$ is the visit probability of
a vertex $\alpha$ during a random walk, and $V$ is the set of vertices
in the graph. $L(M)$ represents the lower bound on the code
length of detected community structure $M$ based on Shannon’s
information theory [14], which is MDL.

The core of the Infomap method also iteratively builds a
hierarchical clustering through multiple stages. Each stage
has two phases. Similar to the first phase of the Louvain
algorithm, each vertex is moved to the neighboring community
that results in the largest decrease of the map equation. If no
movement results in a decrease of the map equation, the vertex
stays in its original community. This procedure is repeated
until no move generates a decrease of the map equation. In
the second phase, the algorithm builds the graph using a
similar method as the Louvain algorithm. The whole process
is repeated until the map equation cannot be reduced further.
D. Sequential Community Detection Framework

Through the basic analysis of the Louvain and Infomap algorithms, we can find that as agglomerative graph clustering they both follow two similar phases, which are vertex movement and graph rebuilding. We generalize both algorithms in Algorithm 1. In Algorithm 1, we treat each vertex as one community initially as Line 1. In Line 4, we calculate the initial measurement $L$ of the graph. In our case, this can be modularity for Louvain, or $MDL$ for Infomap. From Line 5 to Line 10, the algorithm optimizes the communities using the greedy strategy. For each vertex, the algorithm calculates its best movement by modularity gain $\delta Q$ for Louvain, or $\delta MDL$ for Infomap. The vertex will move to the best community that can achieve maximum optimization. The algorithm will repeat this process until the change of $L$ is less than a predefined threshold $\gamma$. The communities will be merged into a new graph, where each vertex represents one community and each edge represents all the edges connecting different communities. The algorithm will output the detected communities as the final output.

E. Distributed Community Detection Framework

In our previous work, we have developed distributed Louvain [4], [8], [9] and Infomap [5] algorithms by carefully investigating the communication and workload patterns of Louvain and Infomap. Our algorithms have achieved more accurate community results and more scalable performance compared to the existing approaches. We demonstrated the effectiveness of our distributed Louvain algorithm with up to 32,768 processors, and our distributed Infomap algorithm with up to 4,096 processors, which are clearly superior to the previous work.

Although different measurements of communities are used in the Louvain and Infomap algorithms, our existing distributed Louvain and Infomap algorithms were designed based on Algorithm 1 and shared considerable similarities. The generalized distributed algorithm can be expressed in Algorithm 2 that consists of four stages:

In the first stage (Line 1), the algorithm uses a delegate partitioning and distribution strategy to divide the input graph among processors. In particular, hubs are duplicated as delegates among processors to ensure that each processor has a similar number of edges. In the second stage (Lines 2 to 7), the algorithm first detects the best community movement of a vertex as Line 3. It then broadcasts the information of delegates that achieve the maximum $\delta L$. This can ensure each delegate to have consistent community movement information and $\delta L$. After the communication from Lines 4 and 5, local community information is updated in Line 6. This process continues until there is no more community change for each vertex. In the third stage (Line 8), the algorithm forms a new graph from the communities. The new graph is several order smaller than the original graph, and thus is partitioned using a simple 1D partitioning [15]. In the fourth stage (Lines 10 to 14), the algorithm processes the subgraphs in a way similar to the second stage, except there are no delegated vertices in the subgraphs. The algorithm stops when there is no improvement of $L$.

IV. OUR APPROACH

A. Rationale

Although we have achieved superior performance in distributed community detection using Algorithm 2, there remains a fundamental bottleneck in Lines 4 and 5 that uses a synchronization strategy to make the community information consistent among the processors, and has two limitations.

First, after finishing local clustering at Line 3, each processor needs to be synchronized before exchanging the community information. This incurs a performance loss.

Second, this can incur the vertex bouncing problem [7] when processors concurrently exchange the latest snapshot information of their local subgraphs. It is due to ghost vertices that may be shared by multiple processors in a distributed environment, which can be illustrated using a simple example.

---

**Algorithm 1** Sequential Community Detection Framework

**Require:**
- $G = (V, E)$: undirected graph, where $V$ is vertex set and $E$ is edge set;
- $\gamma$: per-iteration quality improvement threshold.

**Ensure:**
- $M$: resulting community or modules;
- $L$: resulting measurement.
1: $M = \{(v_i), \{v_i \in V\} \}$
2: $L = L(M)$
3: repeat
4: $L_{prev} = L$
5: randomize the order of vertices
6: for all $u \in V$ do
7: $m_{new} = bestNewCommunity(M, u_i)$
8: Move $u_i$ to $m_{new}$ community, and update $M$ and $L$
9: end for
10: until $L_{prev} - L < \gamma$
11: return $M$

**Algorithm 2** Distributed Community Detection Framework

**Require:**
- $G = (V, E)$: undirected graph, where $V$ is vertex set and $E$ is edge set;
- $p$: processor number.

**Ensure:**
- $M$: resulting communities or modules;
- $L$: resulting measurement;
- $\delta L$: change of $L$.
1: Distributed Delegate Partitioning($G, p$)
2: repeat
3: Parallel local clustering with delegates
4: Broadcast delegates achieving the highest $\delta L$
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: Parallel local clustering without delegates
11: Swap ghost vertex community states
12: Update community information on each processor
13: until No vertex movement
14: Merge communities into a new graph
15: until No improvement of $L$
16: until No improvement of $L$
17: return $M$
in Figure 1, 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 \( v_j \) 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., \( C(v_i) = i \) and \( C(v_j) = j \), where the function \( C \) denotes the community ID of a vertex or a community. After calculating the improvement of local measurement \( L \), both vertices move to each other’s community on their local processors to increase the local measurement gain, i.e., \( C(v_i) = j \) on \( PE_0 \) and \( C(v_j) = i \) on \( PE_1 \), as shown on the blue dash arrows in Figure 1. However, this cannot achieve any global improvement of \( L \). This phenomenon can incur the vertex information bouncing between two different communities, and thus inhibit the convergence of the algorithm, which has not been fundamentally addressed in the existing approaches [4], [5], [7].

These two limitations are rooted in the synchronization strategy. We can easily see that in the sequential framework, a vertex can always find the best community that archives the maximum optimization using the instantaneous information of the global graph (Line 7 in Algorithm 1), and then move to a community according to the received information and make \( v_i \) and \( v_j \) in the same community. This is also held if \( C(v_j) \) is first changed on \( PE_1 \). In this case, the asynchronous communication can send the latest local update to a processor to other related processors. Therefore, the local clustering at each processor (Line 3 in Algorithm 2) can always use the instantaneous global information, and generate the results similar to the sequential algorithm.

B. Highly Asynchronous Visitor Queue Graph

There are a few existing approaches for asynchronous communication. Among them, HavoqGT (Highly Asynchronous Visitor Queue Graph Toolkit) [16] facilities distributed graph traversal using an asynchronous visitor abstraction [17]. The framework is based on delegate partitioning. For each delegated vertex, one of its delegates will be chosen as a master and the others will be controllers. The visitor abstraction allows us to define vertex-centric procedures that execute on vertex, and offers the ability to pass visitor state to other vertices. The visitor procedures and the state to be defined in the visitor are summarized in Table I. In the original HavoqGT graph traversal algorithm, when an algorithm starts, an initial set of visitors are pushed into the queue and the framework’s driver invokes the traversal. The asynchronous traversal completes when all visitors have completed. We leverage HavoqGT to optimize distributed community detection algorithms.

C. Asynchronous Vertex Movement

According to Algorithm 1, when we move one vertex into one community, we can obtain a delta value \( \delta v_{c \rightarrow c'} \). For the Louvain algorithm, \( \delta v_{c \rightarrow c'} = \delta Q_{v_{c \rightarrow c'}} \). For the Infomap algorithm, \( \delta v_{c \rightarrow c'} = \delta MDL_{v_{c \rightarrow c'}} \). As both algorithms adopt a greedy strategy, after moving all vertices, new modularity \( Q' \) and \( MDL' \) can be written as:

\[
Q' = Q - \sum_{v \in V} \delta v_{c \rightarrow c'}; MDL' = MDL + \sum_{v \in V} \delta v_{c \rightarrow c'}. \tag{5}
\]

As we described previously, only when the vertex achieving the maximum modularity gain (the value is larger than 0) or the largest decrease of map equation (the value is smaller than 0), we move the vertex to the corresponding community.

In the asynchronous framework, when a vertex moves, it will affect its neighbors’ movements. Therefore, when we send a vertex movement message to other processors, we should send not only the community index of the vertex, but also the delta information of the community that is affected by this vertex movement. This delta information can be implemented using visitor. As illustrated as List 1 and List 2, we can easily record the community change information when moving vertex. The information includes:

- **Movement information:** record the vertex moving from source community to destination community.
- **Community delta information:** record the community information change when moving the vertex. For the Louvain algorithm, the information is \( \text{deltaTot} \) and \( \text{deltaQ} \), which are the changed total degree of one community and the modularity gain of that movement. For the Infomap algorithm, similar information can be described from Line 6 to Line 13 in List 2.

<table>
<thead>
<tr>
<th>Pre-Visit</th>
<th>Evaluation of whether the visitation should proceed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visitor</td>
<td>Main visitor procedure</td>
</tr>
<tr>
<td>Beast Delegates</td>
<td>Controller broadcasts the current visitor to all delegates</td>
</tr>
<tr>
<td>Make Visitor</td>
<td>Change the vertex into visitor</td>
</tr>
</tbody>
</table>

**Table I**

HavoqGT Visitor Abstraction

![Figure 1](image-url)
Algorithm 3 Unified Optimized Community Detection Framework

Require: 
\[ G = (V, E) : \] undirected graph, where \( V \) is vertex set and \( E \) is the edge set; 
\( p : \) processor number.

Ensure: 
\[ M: \] resulting communities or modules; 
\( L: \) resulting measurement; 
\( \delta L: \) change of \( L. \)

1: repeat 
2: Delegate partitioning 
3: repeat 
4: Local clustering with duplicates using asynchronous vertex movement 
5: until No vertex community state changing 
6: Merge communities into a new graph 
7: until No improvement of measurement 

Algorithm 4 Modified HavoqGT Primitives

1: function pre_visit() 
2: if firstSelfVisit then 
3: return true 
4: else 
5: priority = delegated?'1:0 
6: if caller_priority > priority or (caller_priority == priority and caller.ID < this.ID) then 
7: this.want_count += 1; 
8: return false 
9: end if 
10: end if 
11: 12: function visit(graph, queue) 
13: if vertex is ghost vertex of others or vertex is delegate then 
14: queue.insert(make_visitor(vertex)) 
15: end if 
16: 17: function process_pending_queue(visitor) 
18: repeat 
19: if visitor is delegate and visitor belongs to current PE then 
20: find visitor with the best visitor.optimum 
21: beast_delegates(visitor) 
22: update local community information 
23: else 
24: update local community information 
25: end if 
26: queue.popup() 
27: until queue is empty 

V. EXPERIMENTAL RESULTS

We have evaluated both the quality and scalability of our framework with the existing sequential and distributed algorithms using the synchronization strategy, specifically, the previous distributed Infomap [5] and Louvain [4] algorithms. Table II shows the datasets at different scales used in our experiments\(^1\). Each of the three large real-world datasets (i.e., WebBase-2001, Friendster, and UK-2007) contains more than 1 billion edges. Besides the real-world datasets, we also use R-MAT [18] and BA (Barabasi-Albert) [19] to generate large synthetic datasets. Our graph clustering framework is implemented by MPI and C++. Our experiments have been performed on Titan, a supercomputer at the Oak Ridge Leadership Computing Facility. At the time of our experiments, 

\(^1\)Given the page limit, we only show the results of some datasets for each test. We have gained the same observation for the other datasets in each test.
Algorithm 5 Asynchronous Vertex Movement

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;

\( C^0_s \): initial community of \( G^0_s \);

\( \theta \): modularity gain threshold;

\( P/E \): 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 \( u \in V_{low} \) do
3: \( C^{k}_u = u \)
4: \( m_u = 0 \)
5: \( \sum_{v \in u} C^k_v = 0 \)
6: \( \sum_{v \in u} C^k_v = 0 \)
7: end for
8: repeat
9: for all \( u \in V_{low} \cup V_{high} \) do
10: if \( C^{k+1}_u = \arg \max (Q(C^{k+1}_u \rightarrow C^{k}_u)) > m_u \) then
11: \( C^{k+1}(u) = \min (C(C^{k+1}_u), C^{k}(u)) \)
12: end if
13: end for
14: visit()
15: process_pending_queue()
16: for all \( u \in V_{low} \cup V_{high} \) do
17: \( \sum_{v \in u} C^{k+1}_v = \sum_{v \in u} C^k_v - w(u), v \) \( \sum_{v \in u} C^{k+1}_v = \sum_{v \in u} C^k_v - w(u) \rightarrow C^k_v \)
18: \( \sum_{v \in u} C^{k+1}_v = \sum_{v \in u} C^k_v + w(u), v \) \( \sum_{v \in u} C^{k+1}_v = \sum_{v \in u} C^k_v + w(u) \rightarrow C^k_v \)
19: end for
20: //Calculate partial modularity
21: \( Q_{PE_i} = 0 \)
22: for all \( v \in C_{PE_i} \) do
23: \( Q_{PE_i} = Q_{PE_i} + \sum_{v \in u} C^{k+1}_u - \left( \frac{C^k_u}{2m} \right)^2 \)
24: end for
25: \( Q = Allreduce(Q_{PE_i}) \)
26: \( k = k + 1 \)
27: until No modularity improvement

Each compute node has contained a 16-core 2.2GHz AMD Opteron processor and 32GB memory.

A. Community Quality Analysis

We compare the quality among the existing sequential algorithms, the distributed algorithms using the synchronization strategy [4], [5], and our new distributed algorithms using the asynchronous strategy. We first compare the vertex merging rate among the algorithms. The merging rate is the merged vertex number of each iteration compared to the original graph vertex number. As shown in Figure 2 (Infomap) and Figure 3 (Louvain), compared with the existing algorithms using the synchronization strategy, the convergence of our new algorithms is more similar to the sequential algorithms’.

Figure 4 compares each iteration’s MDL among the Infomap algorithms. Through an asynchronous visitor queue, our new distributed algorithm can achieve a similar quality as the sequential version. Figure 5 compares each iteration’s modularity among the Louvain algorithms. Our new algorithm can also achieve similar results as the sequential algorithm. The modularity of the existing distributed Louvian algorithm is generally lower than the sequential algorithm in each iteration. Moreover, through the asynchronous approach, our new distributed algorithm can converge with the same number of iterations as the sequential algorithm, while the existing distributed algorithm using the synchronization strategy often needs more iterations to coverage.

We have also examined other quality measurements including Normalized Mutual Information (NMI) and F-measure, where a high value corresponds to high quality [9]. Table III shows the results for the Amazon dataset among the existing distributed Infomap [5] and Louvain [4] algorithms using the synchronization strategy and our new algorithms. We can find

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>NMI</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon [20]</td>
<td>Frequently co-purchased products from Amazon</td>
<td>0.34M</td>
<td>0.93M</td>
</tr>
<tr>
<td>DBLP [21]</td>
<td>A co-authorship network from DBLP</td>
<td>0.52M</td>
<td>1.09M</td>
</tr>
<tr>
<td>NDWeb [22]</td>
<td>A web network of University of Notre Dame</td>
<td>0.35M</td>
<td>1.08M</td>
</tr>
<tr>
<td>YouTube [23]</td>
<td>Vertically partitioned social site</td>
<td>1.18M</td>
<td>1.99M</td>
</tr>
<tr>
<td>WellBase-2001 [25]</td>
<td>A crawl graph by WellBase</td>
<td>118.14M</td>
<td>1.01B</td>
</tr>
<tr>
<td>UK-2007 [27]</td>
<td>Web crawl of the .uk domain in 2007</td>
<td>105.9M</td>
<td>3.78B</td>
</tr>
<tr>
<td>DBLP-10M [18]</td>
<td>A DBLP graph satisfying Graph 500 specification</td>
<td>2.41B</td>
<td>2.19B</td>
</tr>
<tr>
<td>BA [19]</td>
<td>A synthetic scale-free graph based on Barabasi-Albert model</td>
<td>2.7M</td>
<td>2.2M</td>
</tr>
</tbody>
</table>
that all the values of our new algorithms are over 0.9, and considerably higher than the existing distributed algorithms, which means that our new algorithms can achieve more similar results as the sequential algorithm.

Our results clearly show that our unified framework has improved the community quality of both the Infomap and Louvain algorithms, compared to the previous work. This is mainly because the asynchronous approach can exchange updated information immediately, rather than possibly out-of-date synchronized information.

**B. Scalability Analysis**

In order to quantify the scalability of our algorithm, we measure the parallel efficiency, more specifically, the relative parallel efficiency \( \tau = \frac{p_1 T(p_1)}{p_2 T(p_2)} \), where \( p_1 \) and \( p_2 \) are the processor number, and \( T(p_1) \) and \( T(p_2) \) are their corresponding running time. In Figure 6, we show the relative parallel efficiency of our new distributed community detection framework for the small and 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 time on 16 processors 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. We can find in our unified framework, both the new distributed Louvain and Infomap algorithms can achieve around 75% to 85% relative parallel efficiency, which proves the scalability of our framework on real-world datasets.

We also examine the scalability using the synthetic datasets generated by R-MAT [18] and BA [19], where we set the vertex scale to 30 and the edge scale is 34. Figure 7 shows the results of the existing distributed Louvain algorithm using the synchronization strategy [4], and our new Louvain algorithm. Both algorithms can achieve nearly linear strong scalability. For R-MAT, the time of the existing Louvian algorithm was reduced from 194.15 seconds to 60.32 seconds, and the time of

![Fig. 4. Comparison of MDL among the existing sequential Infomap algorithm, the existing distributed Infomap algorithm using the synchronization strategy [5], and our new Infomap algorithm.](image)

![Fig. 5. Comparison of modularity among the existing sequential Louvain algorithm, the existing distributed Louvain algorithm using the synchronization strategy [4], and our new Louvain algorithm.](image)

![Fig. 6. Relative parallel efficiency of our new distributed Infomap (a) and Louvain (b) algorithms using different small and large real-world datasets.](image)

![Fig. 7. Strong scaling test using the R-MAT and the BA with the global vertex size of 2^{30} for the existing distributed Louvain algorithm using the synchronization strategy [4], and our new Louvain algorithm.](image)
our new Louvain algorithm was reduced from 189.15 seconds to 56.22 seconds, when we increased the processor number from 8,192 to 32,768. For BA, the time of the existing Louvian algorithm was reduced from 302.16 seconds to 95.45 seconds, and the time of our new Louvian algorithm was reduced from 282.16 seconds to 84.25 seconds. Using these very large synthetic datasets with up to 32,768 processors, the relative parallel efficiencies of the existing algorithm and our algorithm are approximately 80% and 84%, respectively. Moreover, the time of our new algorithm is lower as the synchronization stalls are eliminated.

Figure 8 shows the results of our new Infomap algorithm. For R-MAT, the time of our Infomap algorithm was reduced from 334.55 seconds to 120.91 seconds, when we increased the processor number from 4,096 to 16,384. For BA, the time of our algorithm was reduced from 454.33 seconds to 150.11 seconds. Thus, our algorithm can archive around 70% time of our new algorithm was reduced from 8,192 to 32,768. For BA, the time of the existing Louvian algorithm was reduced from 302.16 seconds to 95.45 seconds, and the time of our new Louvian algorithm was reduced from 282.16 seconds to 84.25 seconds. Using these very large synthetic datasets with up to 32,768 processors, the relative parallel efficiencies of the existing algorithm and our algorithm are approximately 80% and 84%, respectively. Moreover, the time of our new algorithm is lower as the synchronization stalls are eliminated.

Figure 8 shows the results of our new Infomap algorithm. The existing distribute Infomap algorithm [5] did not show the results using such large datasets and large numbers of processors.

ACKNOWLEDGMENT

This research has been sponsored in part by the National Science Foundation through grants IIS-1423487, IIS-1652846, and ICER-1541043, and the Department of Energy through the ExaCT Center for Exascale Simulation of Combustion in Turbulence.

REFERENCES