2015 IEEE 29th International Parallel and Distributed Processing Symposium

Scalable Community Detection with the Louvain Algorithm

Xinyu Que Fabio Checconi Fabrizio Petrini John A. Gunnels

IBM Research
Yorktown Heights, NY 10598
{xque,fchecco,fpetrin,gunnels}@us.ibm.com

***Abstract*—In this paper we present and evaluate a parallel community detection algorithm derived from the state-of-the-art Louvain modularity maximization method. Our algorithm adopts a novel graph mapping and data representation, and relies on an efficient communication runtime, specifically designed for fine-grained applications executed on large-scale supercomputers. We have been able to parallelize graphs with up to 138 billion edges on 8,192 Blue Gene/Q nodes and 1,024 P7-IH nodes. Leveraging the convergence properties of our algorithm and the efficient implementation, we can analyze communities of large-scale graphs in just a few seconds. To the best of our knowledge, this is the first parallel implementation of the Louvain algorithm that scales to these large data and processor configurations.**

I. Introduction

Community detection is an important problem that spans many research areas, such as health care, social networks, systems biology, power grid optimization, etc. [1] It is not surprising that it has been so extensively investigated over the last few years. Community detection algorithms attempt to identify the modules and, possibly, their hierarchical organization, in a graph. While there is no rigorous mathematical definition of community structure, the modularity metric proposed by Girvan and Newman [2] is one of the most used and best known functions to quantify community structure in a graph. From an empirical point of view, a high modularity value typically indicates a good quality partition.

The techniques used for modularity maximization can be categorized into four main classes: greedy, simulated annealing, extremal, and spectral optimization. Greedy optimization applies different approaches to merge vertices into communities for higher modularity, which normally generates high quality communities [3]. Simulated annealing adopts a probabilistic procedure for global optimization on modularity [4].Extremal optimization is a heuristic search procedure [5]. Spectral optimization uses the eigenvalues and eigenvectors of a special matrix for modularity optimization [6]. These methods normally lead to poor results when applied to large graphs with many communities.

Achieving strong scalability together with high-quality community detection is still a challenging, open research problem. Real-world applications often generate massive-scale graphs and require efficient processing. Unfortunately, most of the aforementioned algorithms are sequential, and only a handful of researchers have explored parallel algorithms for community detection on the Cray XMT and Intel-based systems [7]–[11]. These shared memory designs can only handle small to medium-sized graphs. Recent contributions include MapReduce and Message Passing Interface (MPI) parallelizations [12], [13]. However, the fine-grained communication and irregular access pattern to memory and interconnect limit their overall scalability and performance.

The goal of this work is to improve the body of existing results by designing a highly scalable parallel Louvain algorithm for distributed memory systems that can analyze graphs with hundreds of billions of edges.

A. The Louvain Algorithm

Blondel et al. [3] introduced a popular greedy algorithm for community detection–the *Louvain algorithm* [3], for the general case of weighted graphs. This algorithm has been widely utilized in many application domains [14]–[16] thanks to its rapid convergence properties, high modularity and hierarchical partitioning. The Louvain algorithm starts by putting all vertices of a graph in distinct communities, one per vertex. It then sequentially sweeps over all vertices in the *inner loop*. For each vertex i, the algorithm performs two calculations: (1) compute the modularity gain ΔQ when putting vertex i in the community of any neighbor j; (2) pick the neighbor j that yields the largest gain in ΔQ and join the corresponding community. This loop continues until no movement yields a gain. At the end of this phase, the Louvain algorithm obtains the first level partitions. In the second step, these partitions become supervertices and the algorithm reconstructs the graph by calculating the weight of edges between all supervertices. Two supervertices are connected if there is at least one edge between vertices of the corresponding partitions, in which case the weight of the edge between the two supervertices is the sum of the weights from all edges between their corresponding partitions at the lower level. These two steps of the algorithm are then repeated, yielding new hierarchical levels and supergraphs. The algorithm stops when communities become stable. The Louvain algorithm typically converges very quickly, and it can identify communities in just a few iterations.

B. Contributions

The paper provides three main contributions. The first contribution is the parallelization of the Louvain algorithm, which, in its original formulation, is constrained by serial dependence at every level between vertices and iterations. The parallel algorithm preserves and, in a few cases, slightly improves the convergence properties as well as the overall modularity and the quality of the detected communities. A key aspect of the proposed algorithm is a global strategy to orchestrate the migration of vertices to communities using a dynamic threshold that is exponentially decayed during each iteration of the inner loop.

The second contribution is a novel implementation strategy to store and process dynamic graphs, based on two distinct hash tables. The first hash table stores the incoming edges of each vertex, while the second hash table stores the outgoing ones. Using this approach the graph can be dynamically rewritten from scratch during each iteration of the outer loop, grouping vertices and edges in a super-graph with little overhead. We also provide a carefully optimized hierarchical edge hashing scheme. We have identified a high-quality and computationally inexpensive Fibonacci hashing function [17] to load balance the data structures and the occupancy of the hash bins. We believe that this implementation strategy can be generalized to a larger class of graph algorithms, in order to efficiently store and update dynamically changing graphs, where edges are grouped and the topology of the graph changes very frequently.

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.59

The third contribution is an extensive experimental evaluation. We validate the proposed algorithm on a wide variety of real-world social graphs, using two massively parallel machines, Power7-IH [18] (P7-IH) and Blue Gene/Q. More specifically, we analyze (1) the quality of the community detection using a collection of social graphs, obtained from Wikipedia, Amazon, DBLP, YouTube and others; (2) the properties of the hash function and the trade-offs between the size of the hash-table and hash collisions; and (3) the scalability of the algorithm with synthetic R-MAT graphs with up to 8 billion vertices/138 billion edges that are mapped on 8,192 nodes (524,288 threads) of Blue Gene/Q and with BTER graphs with up to 4 billion vertices/138 billion edges on 1,024 (32,768 threads) P7-IH nodes. To the best of our knowledge, these graphs are more than an order of magnitude (32x) larger than the largest results in the literature [7]. With the largest graph on Blue Gene/Q (a scale R-MAT 33 on 8,192 nodes), the inner loop takes only three seconds, and the overall graph and the supervertices can be reconstructed in less than two seconds, providing real-time processing capabilities.

Finally, perhaps a less measurable but equally important contribution, our study provides evidence that distributed-memory machines can efficiently analyze unbalanced, fine-grained and dynamic graph algorithms –a result that just a few years ago was not considered within reach, opening the path for a number of future results in parallel graph algorithmic design.

II. Background

A. Problem Definition

A weighted graph G can be represented as a 2-tuple (V,E), where V denotes a set of vertices, E a set of edges. When $u,v \in V$, an edge $e(u,v) \in E$ has weight $w_{u,v}$. The goal of community detection is to partition a graph into a set of disjoint communities, C, as shown in Equations 1 and 2, which is similar to, but not identical [19], the graph partition problem [20].

$$\cup c_i = V,\ \forall c_i \in C \tag{1}$$
$$c_i \cap c_j = \emptyset,\ \forall c_i, c_j \in C \tag{2}$$

When a graph is partitioned into communities, vertices in the same community are then densely connected while those in different communities are only sparsely connected. The quality of the partition is often measured by the *modularity*, Q [21],

$$Q = \sum_{c \in C} \left[\frac{\Sigma_{in}^c}{2m} - \frac{(\Sigma_{tot}^c)^2}{4m^2} \right], \tag{3}$$

where Σ_{in}^c is the sum of the weights from all internal edges of community c, calculated as $\sum w_{u,v}, \forall u,v \in c$ and $e(u,v) \in E$, Σ_{tot}^c is the sum of the weights from edges incident to any vertex in community c, calculated as $\sum w_{u,v}, \forall u \in c$ or $v \in c$ and $e(u,v) \in E$, and $m = \sum_{e(u,v) \in E} w_{u,v}$ is the normalization factor obtained by summing the edge weights across the entire graph.

B. Sequential Louvain Algorithm

Modularity has been widely used to compare the quality of community detection [22]. However, modularity optimization is an NP-complete problem [23].

$$\begin{aligned} \Delta Q_{u \to c} &= \left[\frac{\Sigma_{in}^c + w_{u \to c}}{2m} - \left(\frac{\Sigma_{tot}^c + w(u)}{2m} \right)^2 \right] \\ &\quad - \left[\frac{\Sigma_{in}^c}{2m} - \left(\frac{\Sigma_{tot}^c}{2m} \right)^2 - \left(\frac{w(u)}{2m} \right)^2 \right] \\ &= \frac{w_{u \to c}}{2m} - \frac{\Sigma_{tot}^c * w(u)}{2m^2} \end{aligned} \tag{4}$$

The Louvain algorithm greedily maximizes the modularity gain $\Delta Q_{u \to c}$ when moving an isolated vertex u into community c, which can be calculated by Equation 4, where Σ_{in}^c, Σ_{tot}^c, and m are defined in Equation 3, $w(u)$ is the sum of the weights of the edges incident to vertex u, and $w_{u \to c} = \sum_{v \in c} w_{u,v}$ is the sum of the weights of the edges from vertex u to vertices in community c.

Algorithm 1: Sequential Louvain Algorithm

```
   Input: G=(V,E): graph representation.
   Output: C: community sets at each level;
           Q: modularity at each level.
   Var: ĉ: vertex u's best candidate community set.
1  Loop outer
2      C ← {{u}}, ∀u∈V ;
3      Σ_in^c ← Σw_{u,v}, e(u,v)∈E, u∈c and v∈c ;
4      Σ_tot^c ← Σw_{u,v}, e(u,v)∈E, u∈c or v∈c ;
5      // Phase 1.
6      Loop inner
7          for u∈V and u∈c do
8              // Find the best community for vertex u.
9              ĉ ← argmax_{∀c', ∃e(u,v)∈E, v∈c'} ΔQ_{u→c'} ;
10             if ΔQ_{u→ĉ} > 0 then
11                 // Update Σ_tot and Σ_in.
12                 Σ_tot^ĉ ← Σ_tot^ĉ + w(u) ; Σ_in^ĉ ← Σ_in^ĉ + w_{u→ĉ} ;
13                 Σ_tot^c ← Σ_tot^c − w(u) ; Σ_in^c ← Σ_in^c − w_{u→c} ;
14                 // Update the community information.
15                 ĉ ← ĉ ∪ {u} ; c ← c − {u} ;
16         if No vertex moves to a new community then
17             exit inner Loop;
18     // Calculate community set and modularity.
19     Q ← 0 ;
20     for c∈C do
21         Q ← Q + Σ_in^c/2m − (Σ_tot^c/2m)^2 ;
22     C' ← {c}, ∀c∈C ; print C' and Q ;
23     // Phase 2: Rebuild Graph.
24     V' ← C' ;
25     E' ← {e(c,c')}, ∃e(u,v)∈E, u∈c, v∈c' ;
26     w_{c,c'} ← Σw_{u,v}, ∀e(u,v)∈E, u∈c, v∈c' ;
27     if No community changes then
28         exit outer Loop;
29     V ← V' ; E ← E' ;
```

The algorithm tries to identify communities with the maximum modularity Q, as shown in Algorithm 1. For each vertex u, the algorithm considers each neighbor v of u and evaluates the modularity gain (ΔQ) which could take place by removing u from its community c and by placing it in the neighbor's community c'. The vertex u is placed in the community $\hat{c}$ that maximizes ΔQ. When there is no positive modularity gain, u remains in the original community (Lines 7–15). All vertices are processed until no further improvement can be achieved, obtaining the community information (C') and modularity Q (Lines 19–22). The second phase of the algorithm builds a new graph whose vertices are the communities found in the previous iteration. The new set of vertices V' consists of the latest communities, and the weights of the edges between the new vertices are given by the sum of the weight of the edges between vertices in the corresponding two communities [24]. The edges between vertices of the same community lead to self-loops for this community in the new graph (Lines 24–26). The algorithm repeats these two phases for the next level until the communities become stable.

III. Challenges in Designing a Parallel Louvain Algorithm

The most computationally intensive parts of the Louvain algorithm are the evaluation of the modularity gain for all vertices (Lines 7–15 in Algorithm 1) and the construction of the next-level supergraph based on the new community structure (Lines 24–26 in Algorithm 1). During the modularity gain (ΔQ) computation and the supergraph reconstruction, the change of the community membership for each vertex is applied immediately. Therefore, the original algorithm is permeated by serial dependencies at every level, from vertex to vertex and iteration to iteration. The parallelization of the Louvain algorithm on a distributed memory system cannot be achieved by simply partitioning all vertices and distributing the modularity gain (ΔQ) calculation across compute nodes because the change of community membership at one vertex is not available to the other vertices.

The first challenge is the evaluation of the modularity gain ΔQ for all vertices with updated community membership. Each vertex computes the modularity gain for joining its neighbors' communities, gathering the required information of communities c', such that $w_{u\to c'}$. This resulting collective operation requires a significant amount of communication and synchronization, and limits the degree of parallelism.

The second challenge is to maintain (or improve, if possible) the convergence properties. The sequential algorithm always increases the modularity when moving a vertex to a community, thanks to the greedy policy. However, this is not always true in a parallel algorithm: when vertices compute their ΔQ in parallel, they can only see a limited snapshot of their neighbors' pre-existing community membership. The convergence property of the original greedy policy is no longer preserved. Vertices may end up exchanging obsolete community membership with little gain in modularity, resulting in the infinite movement of vertices, without reaching convergence.

A solution to these challenges requires not only an in-depth examination of the inherent computation and communication balance in the algorithm, but also a novel heuristic that can dynamically control the vertices' movement to different community membership, purge obsolete or non-contributing vertices, and eventually converge with high-modularity communities. Section IV details our parallel algorithm and how we address these challenges.

IV. A Parallel Louvain Algorithm for Distributed Memory Systems

The proposed parallel version of the Louvain algorithm adopts a novel hash-based graph mapping using two distinct tables for incoming and outgoing edges, a heuristic that controls the fraction of vertices moved during each iteration, a hash functions selected and tuned for this algorithm, and a highly optimized communication run-time support to handle the fine-grained communication generated by the state updates. Together, these techniques are integrated into our parallel Louvain algorithm, which we have implemented and tested on P7-IH and BlueGene/Q.

A. Hash-Based Data Organization

The major algorithmic challenge faced by the parallel implementation is how to store a dynamically changing graph, and how to resolve the contributions that each vertex needs to provide to its neighbors. More specifically, a vertex may be connected to other vertices that are in the same community, whose weights need to be summed to determine the proper change in modularity. In order to quickly resolve this at run-time we have two possible solutions: we can either sort all the contributions for each vertex or we can hash them. While the sorting algorithm would require a complete shuffling of data during each iteration of the inner loop, the hash-based approach can effectively deal with the dynamic nature of the graph. An innovative aspect of our proposed algorithm is to use two distinct hash tables to represent the (directed) graph: one to store the *incoming edges* of each vertex, that is not changed during the inner loop and keeps track of the original graph structure, and another hash table for the *outgoing edges*, which is changed during each iteration. This data layout can also support the outer loop, remapping the whole graph simply deleting the content of the input table and replacing it with the specular image of the output table of the previous phase. We believe that this novel approach is very promising to attack a larger class of dynamic graph problems, and its applicability is not limited to the Louvain algorithm.

We linearly split the vertices and their edge lists among the compute nodes using a 1D decomposition. Each node is assigned a set of vertices according to a simple modulo function. The same node is responsible for all the information related to the vertices, edges, vertex and community info and other data.

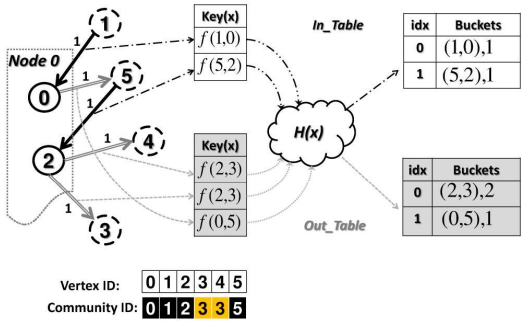


Fig. 1. Edge Hashing Scheme

The edges belonging to the vertices on one compute node are managed by two distinct hash tables, namely *In_Table* and *Out_Table*. Figure 1 shows the hash mechanism for two representative vertices 0 and 2 on *Node 0* along with all the neighbor vertices $(1,3,4,5)$. Both hash tables are hashed on edges. The *In_Table* manages the in-edges (the edges with destination as vertex 0 or 2 originating from other nodes) and the *Out_Table* manages the out-edges (the edges with source as vertex 0 or 2 connected to communities). The input key (x) is a function of a tuple (t_1,t_2) as defined in Equation 5 ($\ll$ is bitwise left shift operator and $|$ is bitwise OR operator), which is different for the two hash tables. For *In_Table* the tuple contains the source vertex u and the destination vertex i (i is the vertex 0 or 2 in Figure 1) of the edge being hashed ((u,i), $\forall e(u,i) \in E$). For *Out_Table*, the tuple contains the source vertex i and the destination vertex v's community c for the corresponding edge ((i,c), $\forall v \in c, e(i,v) \in E$).

$$f(t_1,t_2) = (t_1 \ll 16)|t_2 \tag{5}$$

In addition, for the *In_Table*, the bucket is a triple $((u,i),w_{u,i})$, which includes the weight of the corresponding edges $(w_{u,i})$. The bucket in the *Out_Table* is a triple $((i,c),w_{i\to c})$, where $w_{i\to c}$ (as defined in Equation 4) refers to the sum of weights of all the edges from the vertex i to a specific community c because all these edges are hashed to the same bucket in the *Out_Table*. In our experiments we have utilized a Fibonacci hash [17] as the primary hash function, which is shown in Equation 6, where M is the size of the hash table, W is $2^{64}-1$, and ϕ is the *golden ratio* [25].

$$H(x) = \lfloor \frac{M}{W} \cdot ((\phi^{-1} \cdot W \cdot x) mod W) \rfloor \tag{6}$$

B. A Novel Convergence Heuristic

We have examined the convergence behavior of the sequential Louvain algorithm through extensive simulation analysis. In particular, we have traced the migration patterns of the vertices using the LFR benchmark [26]. LFR is designed to generate graphs to test community detection algorithms and mimic the properties of real-world, large-scale complex graphs, through tunable parameters. During the initial part of the algorithmic design, we have observed an inverse exponential relationship between the movement of the vertices and the number of iterations in the inner loop. By using statistical regression to quantify such relationship, we have derived a dynamical threshold ε,

$$\varepsilon = p_1 * e^{\frac{1}{p_2 * iter}} \quad (7)$$

which identifies the fraction of the vertices that need to be updated during each iteration of the inner loop. The threshold decreases exponentially with the number of iterations as shown by Equation 7, where *iter* is the number of iterations of the current inner loop, and p_1 and p_2 are the parameters obtained through regression training and analysis.

Figure 2 compares the regression analysis and the dynamical threshold ε with different community structure graphs

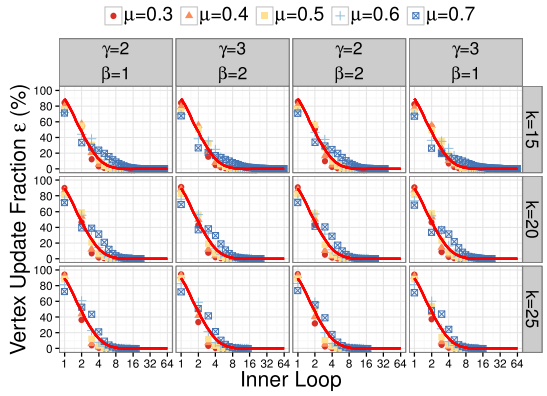


Fig. 2. Simulation Analysis Comparison

generated by LFR benchmark, where we only show the first outer loop due to space limitations. The x-axis is the inner loop and y-axis is the vertex update fraction ε. The generated graphs vary in average degree k, power-law distribution γ, community size distribution β, and the fraction of edges between vertices belonging to different communities μ, which carry different community structure with modularity from 0.2 to 0.8 (we refer to the LFR paper for further details [26]). Points with the same color corresponds to 100 experiments with the LFR benchmark and the red line is the regression analysis obtained with ε. Different community graphs require a different number of iterations to converge and the regression analysis ε is able to capture the execution behavior of different community structures. We translate ε to the modularity gain threshold $\Delta\hat{Q}$ by sorting the $\Delta Q_u(u \in V)$ for all the vertices and selecting the top ε vertices to update the community membership. This purges the vertices that provide limited modularity gains.

Once $\Delta\hat{Q}$ is known, the vertices can update the community information in parallel. We evaluate the impact of varying $\Delta\hat{Q}$ in section V-B.

C. Parallel Louvain Algorithm

The pseudocode for the parallel Louvain algorithm is shown in Algorithm 2. The algorithm starts by initializing all the data structures. At the very beginning, the *In_Table* contains all the in-edges information of the vertices owned by each node/process and the *Out_Table* is empty. The vertices form independent communities, which are managed by the owner process of the vertices. The algorithm starts with STATE PROPAGATION(), which globally propagates the community state information. Once all the messages have been delivered and hashed in place, *Out_Table* is initialized. We then invoke REFINE(), which corresponds to the inner loop of the sequential algorithm and orchestrates the vertex migration to different communities, based on the modularity gain. We reconstruct the graph with (GRAPH RECONSTRUCTION()) and prepare for the next round, which corresponds to the outer loop. The algorithm halts when there is no further modularity improvement.

Algorithm 2: Parallel Louvain Algorithm.

Input: p: process;
$G=(V,E)$: graph representation;
V_p: vertices owned by process p.
Output: C: community sets at each level;
Q: modularity at each level.
Var: C_p: community set owned by process p;
In_Table_p: In_Table owned by process p;
Out_Table_p: Out_Table owned by process p.

```
1  In_Table_p ← ((v,u),w_{v,u}), ∀u∈V_p, ∀e(v,u)∈E ;
2  Out_Table_p ← ∅ ;
3  C_p ← {{u}}, ∀u∈V_p ;
4  Loop outer
5      STATE PROPAGATION() ;
6      // Refine vertices' community set.
7      REFINE() ;
8      print C_p and Q ;
9      // Reconstruct the Graph.
10     GRAPH RECONSTRUCTION() ;
11     if No improvement on the modularity then
12         exit outer Loop;
```

1) Community State Propagation

The implementation of the Louvain Algorithm relies on a highly-optimized messaging layer specifically designed to support graph algorithms and fine-grained communication patterns. Due to space limitations we refer to [27]–[29] for more information.

The state propagation procedure is shown in Algorithm 3. All processes sequentially scan their *In_Table* and send messages to the destination processes that own the corresponding vertices (v in the algorithm) (Lines 4–5). In the meantime, the processes also receive messages from others and insert/update hashed edges to the *Out_Table* (Lines 7–11)

Algorithm 3: Community State Propagation.

```
1  function STATE PROPAGATION
2  begin
3      // Scan In_Table and send messages.
4      for ((v,u),w) ∈ In_Table_p do
5          send ((v,c),w) to process p'(v∈V_{p'}, u∈c) ;
6      // Update Out_Table.
7      for ((u,c),w) received do
8          if ∃((u,c),w') ∈ Out_Table_p then
9              w' ← w' + w ;
10         else
11             place the triple with linear probing ;
```

2) Refine

Algorithm 4 describes the REFINE() routine. It starts with initializing $\hat{c_u}$ and m_u. The *Out_Table* contains all the

Algorithm 4: Refine.

```
   Var: m_u: vertex u's maximum modularity gain;
        ĉ_u: vertex u's best candidate community set.
1  function REFINE
2  begin
3      Loop inner
4          m_u ← 0, ĉ_u ← c, ∀u∈V_p, u∈c ;
5          // FIND BEST COMMUNITY.
6          for ((u,c'),w) ∈ Out_Table_p, u∈c do
7              if ΔQ_{u→c'} ≥ m_u then
8                  ĉ_u ← c' ;
9                  m_u ← ΔQ_{u→c'} ;
10         // Obtain updated threshold.
11         Compute threshold ΔQ̂ from Equation 7 ;
12         // UPDATE COMMUNITY INFORMATION.
13         ∀u∈V_p  if m_u ≥ ΔQ̂ then
14             Σ_tot^{ĉ_u} ← Σ_tot^{ĉ_u} + w(u) ;  Σ_tot^c ← Σ_tot^c − w(u) ;
15             ĉ_u ← ĉ_u ∪ {u} ;  c ← c − {u} ;
16         STATE PROPAGATION() ;
17         // Calculate Σ_in.
18         for ((u,c),w) ∈ Out_Table_p do
19             if (u∈c) then
20                 Σ_in^c ← Σ_in^c + w ;
21         // Calculate community set and modularity.
22         Q_p ← 0 ;
23         for c∈C_p do
24             Q_p ← Q_p + Σ_in^c/2m − (Σ_tot^c/2m)^2 ;
25         Q ← Allreduce(Q_p) ;
26         if No improvement on the modularity then
27             exit inner Loop;
28         Reset Out_Table_p and Σ_in^c ;
29         V_p ← C_p ;
```

neighbors' communities and the accumulated weights to the communities from the vertices owned by that process. Each process starts scanning the hashed edges (u,c') and then calculates the modularity gain of putting vertex u to community c'. It updates the corresponding $\hat{c_u}$ and m_u if needed, as shown by Lines 6–9. After all the *Out_Table*s have been examined, each vertex has the information on the best community to join.

We then build a histogram based on m_u and calculate the update threshold $\Delta\hat{Q}$ according to Equation 7. Lines 13–15 update the community information and the corresponding Σ_{tot} based on $\Delta\hat{Q}$, and we compute the new modularity (Lines 16–25). For this purpose, we need the latest Σ_{in}^c, which is the sum of the weights of all edges inside community c. We first perform a community state propagation to refresh the latest community information and weights (Line 16). Then we calculate the Σ_{in} for each community (Lines 18–20). Finally, we calculate the modularity (Lines 23–25). The procedure terminates when no further vertex movement can increase the modularity.

3) Graph Reconstruction

The reconstruction of the super graph can be easily achieved through an all-to-all exchange of messages between *Out_Table* and *In_Table*.

Algorithm 5 reconstructs a graph whose vertices are now the communities found during the REFINE() phase. To do so, the weights of the edges between the new vertices are given by the sum of the weight of the edges between vertices in the corresponding two communities. Edges between vertices of the same community lead to self-loops in the new graph. Lines 4–5 scan the *Out_Table* and send the aggregated edges information to the owner process of the vertex (community).

Algorithm 5: Graph Reconstruction.

```
1  function GRAPH RECONSTRUCTION
2  begin
3      Reset In_Table_p ;
4      for ((u,c),w) ∈ Out_Table_p do
5          send ((c',c),w) to process p'(c∈C_{p'}, u∈c') ;
6      // Reconstruct In_Table.
7      for ((v,u),w) received do
8          if ∃((v,u),w') ∈ In_Table_p then
9              w' ← w' + w ;
10         else
11             place the triple with linear probing ;
12     Reset Out_Table_p ;
```

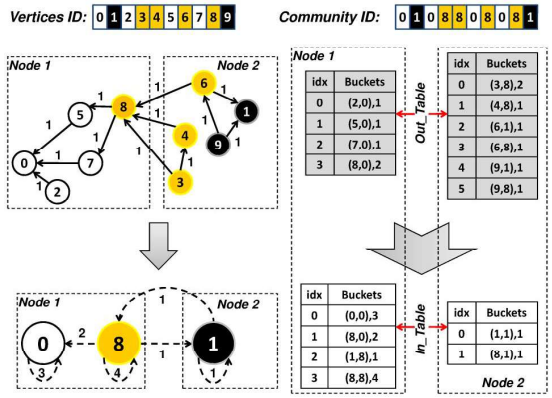


Fig. 3. Graph Reconstruction Steps

Upon receiving the messages, lines 7–11 finally prepare the *In_Table* as the input graph for the next round of execution.

Figure 3 illustrates the idea of graph reconstruction with an example, where a weighted graph with 10 vertices is partitioned across 2 nodes (vertex $0, 2, 5, 7, 8$ on node 1 and vertex 1,3,4,6,9 on node 2). The community information (same color for vertices belonging to the same community) and the latest *Out_Table* are available as well. Each node sends a message to the remote *In_Table* and updates its *In_Table* . For example, node 2 first scans $((3,8),2)$ and sends $((8,8),2)$ to node 1, which will be hashed (inserted/updated) into the third bucket in its *In_Table*. When all the messages have been delivered, the *In_Tables* now represent the full super graph, as shown in the bottom left. In this way we transform the graph relabeling problem into an all-to-all communication with hashing.

V. Experimental Results

In this section we present the experimental analysis of the parallel Louvain algorithm. We begin by introducing the metrics used in the evaluation and the input graphs, then we compare the quality of the communities identified by the parallel implementation to that of the original sequential algorithm. We then analyze the performance of the algorithm, starting from an evaluation of the hashing techniques used, continuing with a breakdown of the execution time, concluding with a scalability study.

The hardware systems used in this experimental evaluation were Zeus, an IBM Power7-IH [18] cluster, and Mira, a 48 rack BlueGene/Q system operated by the Argonne Leadership Computing Facility (ALCF). We have coded our algorithms using the C language and Pthreads.

TABLE I. GRAPHS USED FOR EVALUATION

Category	Size	Name	Description	# Vertices	# Edges	Diameter	References
Real-world Graphs	Small	Amazon	Amazon product co-purchasing network	0.335M	0.925M	44	[30]
		DBLP	DBLP collaboration network	0.317M	1.049M	22	[30]
		ND-Web	University of Notre Dame web-pages network	0.325M	1.497M	46	[31]
		YouTube	YouTube social network	1.135M	2.987M	21	[30]
	Medium	LiveJournal	LiveJournal Social Network	3.997M	34.68M	18	[30]
		Wikipedia	Graph of the English Part of Wikipedia	4.206M	77.66M	6.81	[32], [33]
	Large	UK-2005	Web crawl of English sites in 2005	39.46M	936.4M	23	[32], [33]
		Twitter	The entire Twitter follower links of July 2009	41.7M	1,470M	18	[34]
	Very Large	UK-2007	Web crawl of English sites in 2007	105.90M	3,783.7M	23	[32], [33]
Synthetic Graphs	Small	LFR graphs	Sequential generator with build-in communities	100,000	1.6M	N/A	[26]
	Very Large	R-MAT graphs	Conforming to Graph500 Specifications	2^{SCALE}	$2^{SCALE+4}$	about 13	[35]
		BTER graphs	Block Two-Level Erdős-Rényi model generator	4,295M	138,000M	N/A	[36], [37]

A. Benchmarks and Performance Metrics

In our evaluation, we used both real-world and synthetic social graphs, as listed in Table I.

The real-world graphs have good community structure, which include friendship networks, collaboration networks, and shopping networks, collected from sources such as Amazon, DBLP, YouTube, LiveJournal, Wikipedia, and Twitter, plus three webpage networks available in the literature, ND-Web, UK-2005, and UK-2007. We further classify these graphs according to their size. Small, Medium, and Large graphs fit in the memory of a single node, while Very Large graphs require multiple nodes to be stored and processed.

We also used the synthetic graphs LFR, R-MAT, and BTER. Using LFR in section IV-B, we illustrate the manner in which we fine-tune our convergence heuristic. LFR provides many tunable parameters that allow quick prototyping of graphs with various degrees of community structure. R-MAT is a popular family of scale-free graphs that has been adopted by many benchmarks, but does not have any marked community structure [36]. The community issue is addressed by BTER, which can generate a rich class of graphs with parametric community structure. The last two parallel generators are ideal candidates for algorithmic and scalability evaluation on large node counts.

We use the metrics listed in Table II to analyze the algorithmic behavior, examine the quality of the output, and study the scalability of our parallel algorithm.

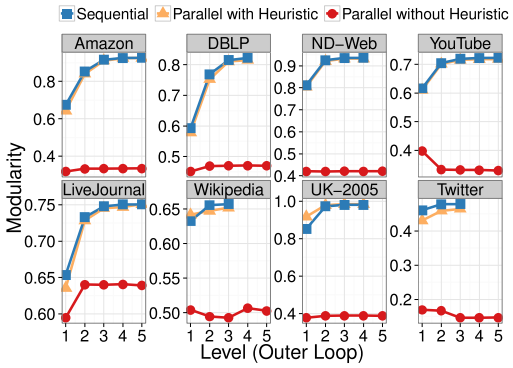
TABLE II. EVALUATION METRICS

	Metrics
Community Detection Quality	*Modularity, Evolution Ratio, Community Size Distribution, F-measure, Rand Index (RI), Normalized Mutual Information (NMI), Normalized Van Dongen metric (NVD), Jaccard Index (JI), Adjusted Rand Index (ARI)*
Hash Function	*Number of Hashed Entries, Average Bin Length, Maximum Bin Length*
Scalability	*TEPS* (Traversed Edges Per Second)

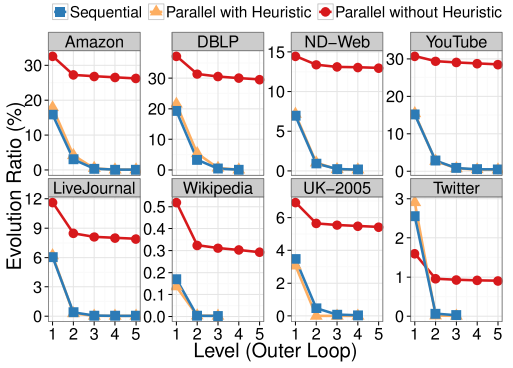
B. Community Detection Analysis

The first objective of the experimental evaluation is to determine the accuracy of the output produced by our algorithm. The type and quality of the detected communities are in general heavily influenced by the order in which vertices are processed and updates are performed. The parallel version also presents additional complexities, due to the fact that the updates may be based on stale information, potentially resulting in oscillating behavior of some vertices between communities, at the risk of greatly slowing or preventing convergence altogether.

Given the approximate nature of our approach, we are not providing a formal proof of convergence for our parallel algorithm. Instead we show how it compares to the original sequential formulation. We consider several metrics: Newman's modularity [21], which characterizes the quality of the overall community selection, the *evolution ratio*, the ratio between the number of vertices of the community graph and the original graph, the distribution of community sizes, and other similarity metrics based on *F-measure*, *RI*, *NMI*, *NVD*, *JI*, and *ARI*.



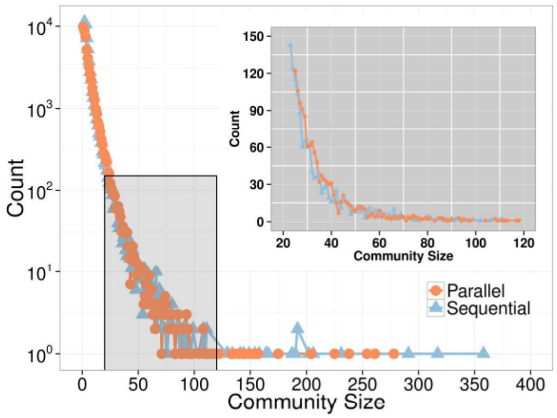
(a) Modularity (higher is better)



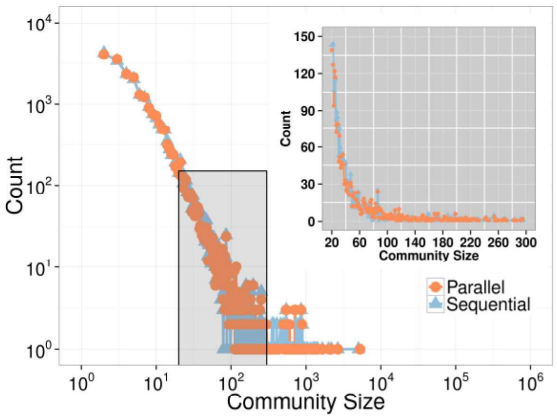
(b) Evolution Ratio (lower is better)

Fig. 4. Convergence and Detection Quality with Social Networks

Figure 4 compares the convergence properties of the sequential algorithm, a naïve parallel algorithm, and the proposed parallel algorithm using the convergence heuristic described in section IV-B. The x-axis shows the outer loop iteration, while the y-axis shows (a) modularity and (b) evolution ratio. We can make several observations. First, the basic parallel version converges very slowly, if at all, as shown in Figure 4 (a), with a very low modularity score. The lack of coordination between



(a) Amazon



(b) ND-Web

Fig. 5. Community Size Distribution with Small Social Graphs

vertex updates creates a chaotic motion of vertices between communities, resulting in very small improvements on the overall modularity across multiple iterations. The parallel version with the proposed heuristic is on par with the original sequential algorithm and, surprisingly, in one case, UK-2005, obtains slightly higher modularity. The first iteration is typically the most important. LiveJournal, ND-Web, Wikipedia, UK-2005, and Twitter, have more than 94% of their vertices merged into communities during the first iteration. The sequential algorithm is able to identify the hierarchical community structures up to five levels, three levels for Wikipedia and Twitter, four levels for UK-2005, ND-Web, and DBLP, and five levels for LiveJournal, Amazon, and YouTube. The memory requirements of UK-2007 prevented us from running the sequential version.

To provide further insight, we have also examined the distribution of the sizes of the detected communities. Figure 5 shows the distribution for two different small size social graphs, Amazon and ND-Web, all characterized by the presence of very few large communities and many smaller ones. The largest communities discovered by the sequential algorithm have 358 and 5020 vertices, respectively for each of the two graphs. With the parallel algorithm the largest corresponding communities have 278 and 5286 vertices. Furthermore, the parallel algorithm achieves a very similar distribution of the community size.

TABLE III. QUALITY COMPARISON ON COMMUNITY STRUCTURE

Graphs	*NMI*	*F-measure*	*NVD*	*RI*	*ARI*	*JI*
Amazon	**0.9734**	0.8159	0.1461	0.9989	0.6775	0.5123
ND-Web	**0.9848**	0.9270	0.0510	0.9998	0.9219	0.8552
LFR(μ=0.4)	**0.9903**	0.9452	0.0404	0.9999	0.9415	0.8895
LFR(μ=0.5)	**0.9833**	0.9058	0.0683	0.9999	0.9034	0.8239

We also considered several other metrics to assess the similarity of the solutions. These metrics can be divided into three categories: *NMI* based on information theory; *F-measure* and *NVD* based on cluster matching; *RI*, *ARI*, and *JI* based on pair counting. Due to space limits, we show the results for four graphs including both social graphs and LFR synthetic graphs with different parameter μ [26]. Although different measures have different bias, the results shown in Table III indicate that the communities detected by the parallel algorithm are very similar to those of the sequential algorithms in general with *NVD* close to 0 and the rest close to 1[1], especially for the most widely used measure *NMI* [38][2].

[1]If the community structures are identical, *NVD* will be 0, and all the rest will be 1

[2]The code we used for calculating the metrics is from https://github.com/chenmingming/ParallelComMetric

While not conclusive, we believe these experiments provide strong evidence that the proposed parallel algorithm preserves the main convergence properties of the sequential one on real-world graphs.

C. Hash Behavior Analysis

Our parallel algorithm is built around a hash-based data representation. The hash operations used in the algorithm, sequential scan and insertion/update, are critical to its overall performance. The experiments performed in this subsection analyze the hash load factor. We use synthetic, scale-25 R-MAT graphs over 16 compute nodes with 32 threads running concurrently on each P7-IH node.

1) Hash-based Load Balancing

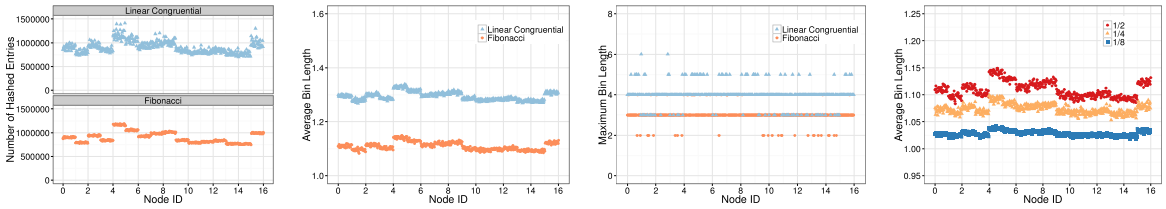
We have examined several classes of hash functions that can be computed with very low overhead at run-time: *concatenated hash*, *linear congruential hash* [39], *bitwise hash*, and *Fibonacci hash*. Our analysis shows that, for our purposes, linear congruential and Fibonacci hashing perform better than the aforementioned alternatives.

In Figure 6 (a-c) we compare the load induced by the two hash functions by storing an R-MAT graph partitioned using a 1D data distribution. Each node is assigned a uniform partition of the vertices of the graph, and the edges incident to those vertices are stored in the node's hash table. The bins of each node's hash table are again partitioned uniformly across the threads of the node. Figure 6 (a) shows the number of entries (i.e., edges) assigned to each thread. While the load per node is the same and is determined by the 1D distribution, within each node the Fibonacci hash provides a better load-balance. This is reflected by the shorter average bin length[3], as shown in Figure 6 (b), and maximum bin length, 3 vs 6, in Figure 6 (c).

2) Impact of the Load Factor

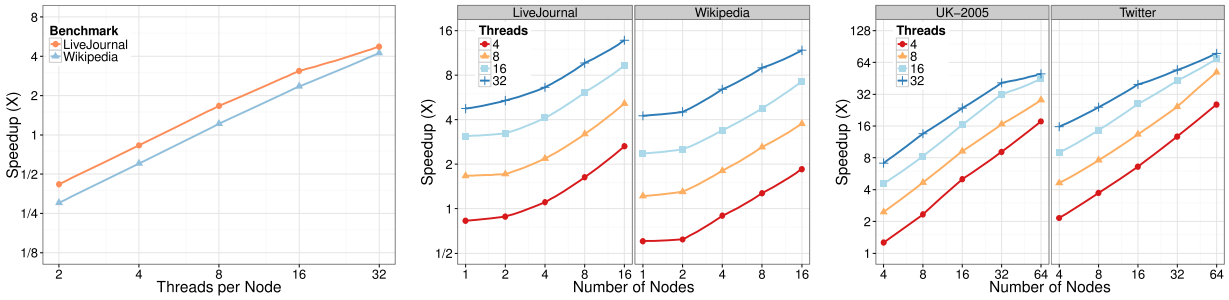
The load factor [40] is another critical performance factor. There is a clear trade-off between the amount of memory used and hash performance. A small load factor reduces the chance of collisions, but wastes more memory. Correspondingly, a large load factor increases the chance of collisions, but reduces the memory requirements. Figure 6 (d) compares the average bin length per thread with different load factors. As expected, a lower load factor implies a lower average bin length. The

[3]We compute the average bin length only considering the bins that have one or more entries.



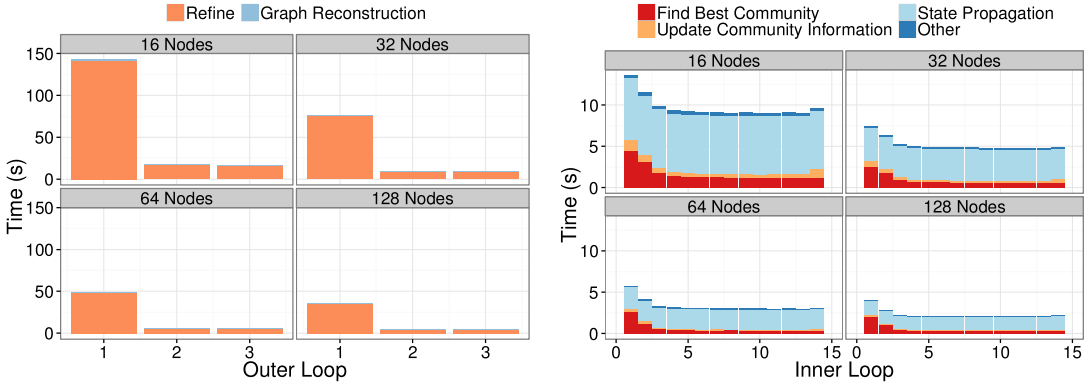
(a) Hashed Entries (b) Average Bin Length (c) Maximum Bin Length (d) Impact of Load Factor

Fig. 6. Hash performance. The graph reports a performance point for each of the 32 threads in a node, a total of 512 points.



(a) Thread Scaling on Single Node (b) Node Scaling for Medium social graphs (c) Node Scaling for Large social graphs

Fig. 7. Speedup with medium and large social graphs



(a) Execution Breakdown of the *outer loop* (b) Execution Breakdown of the *inner loop*

Fig. 8. Time breakdown with UK-2007

average bin length is close to 1 when the load factor is reduced to 1/8, which in turn gives the highest hit ratio. This experiment provides information that can be used to configure the load factor in the real parallel environment. When memory is available, smaller load factors should be chosen for good performance. In the experiments, we choose a 1/4 load factor, a good compromise between speed and memory requirements.

D. Speedup and Execution Time Breakdown

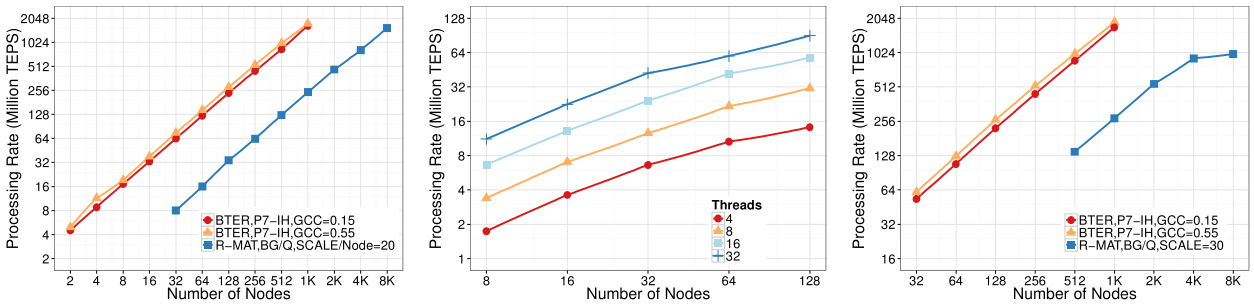
We now examine the main performance components of our parallel algorithm on P7-IH, using medium and large social graphs: LiveJournal, Wikipedia, UK-2005 and Twitter. Figure 7 (a) shows the thread speedup on a single node, from 2 to 32 threads, and (b), (c) show the node speedup, from one to 64 nodes. All the speedups are relative to the original single-threaded implementation [41]. Our parallel algorithm exhibits a fair amount of speedup in all cases. To put this in perspective, the 49.8x speedup obtained with UK-2005 on 64 nodes, is 5.6 times faster than the result reported in [42], which examines a random sampling generated sub-network with 31% edges of the original UK-2005 graph.

To better characterize the speedup, we examine the execution breakdown of the largest real-world graph, UK-2007. We also include a summary of the best reported results in the literature in Table IV. The corresponding detailed timing result is shown in Figure 8, where (a) is the breakdown for the *outer loop* and (b) is breakdown for the *inner loop* executed within the first *outer loop*. We analyze different phases as described in algorithm 2, REFINE and GRAPH RECONSTRUCTION for the *outer loops* and FIND BEST COMMUNITY, UPDATE COMMUNITY INFORMATION, STATE PROPAGATION for the *inner loop*. Due to memory constraints, we could not run the sequential algorithm.

TABLE IV. PERFORMANCE RESULTS OF UK-2007 IN THE LITERATURE

Reference	Time	Modularity	Processors	System
[7]	504.9 seconds	N/A	4	Intel E7-8870
[10]	8 minutes	N/A	2	Intel E5-2680
[12]	few hours	0.994	50 nodes	Intel Xeon
This paper	44.90 seconds	0.996	128 nodes	Power 7

As shown in the figure, the very first *outer loop* is the most time consuming step and accounts for over 90% of the total execution time, which is very similar to the sequential Louvain



(a) Weak Scaling (b) UK-2007 On P7-IH (c) Strong Scaling

Fig. 9. Scaling Analysis

algorithm, and is validated by the profiling shown in [42]. It is worth noting that the largest fraction of the run-time is spent on the REFINE phase, and the graph reconstruction (GRAPH RECONSTRUCTION) is negligible for all the *outer loops*. In addition, for each *inner loop* in Figure 8 (b), the execution time is different. This is because the computation and the updates for the vertices' community are different. It is not surprising that the STATE PROPAGATION takes similar time for all the *inner loops*. The time for FIND BEST COMMUNITY and UPDATE COMMUNITY INFORMATION decreases because more and more vertices form communities. When increasing the number of compute nodes, the execution time for each *inner loop* decreases which contributes to the reduction of the *outer loop* and the total execution time.

E. Scalability Analysis

We complete the experimental evaluation by analyzing the scalability using both very large real-world and synthetic graphs, R-MAT [35] and BTER [37], on Blue Gene/Q and P7-IH [18]. We use Traversed Edges Per Second or TEPS, a metric borrowed from the Graph 500, to provide a global indication of the graph processing time. As in the sequential version of the Louvain algorithm, with our parallel algorithm the graph shrinks significantly during the first iteration, which generates the most informative community structure. Therefore we compute the TEPS rate by dividing the number of input edges by the execution time it takes to finish the first level.

Figure 9 (a) shows weak scaling results. On BG/Q, we use an R-MAT graph with 2^{20} vertices and 2^{24} undirected edges per node, from 32 to 8,192 nodes. On P7-IH, we use a BTER graph with 2^{22} vertices per node and an average degree of 32, with two Global Clustering Coefficients (GCC), [43], [44] 0.15 and 0.55, to differentiate the community structure. Our parallel algorithm achieves good scalability. The processing rate is proportional to the *number of nodes*. We observed 1.54 billion TEPS (GTEPS) for R-MAT graph with 138 billion edges on 8,192 BG/Q nodes and 1.75 GTEPS for BTER graph with 138 billion edges on 1,024 P7IH nodes. For BTER generated graphs, higher GCC gives higher modularity, 0.926 for GCC=0.55 and 0.693 for GCC=0.15, which implies better community structure, with a slightly faster processing rate for the graph with better community structure. Figure 9 (b) shows the strong scaling of the very large social graph with 3.7 billion edges on P7-IH. Figure 9 (c) shows the strong scaling results on synthetic graphs. On BG/Q, we run an R-MAT graph with problem scale of 30 from 512 to 8,192 nodes with a processing rate of 0.99 GTEPS. This is lower than in the weak scaling mode because the problem scale is not big enough to generate enough parallelism. On P7IH, we run from 32 nodes to 1,024 nodes and get a processing rate of 1.89 GTEPS, using a graph with 34.6 billion edges.

VI. Related Work

Given the centrality of community detection in a number of scientific domains it is unsurprising that a good deal of recent work has been done in the area. Riedy et al. proposed a parallel agglomerative algorithm by partitioning a graph into subgraphs and merging the intermediate subgraphs [7]. Martelot et al. [9] demonstrated a multi-threaded algorithm for fast community detection with a local criterion that dispatches the work to multiple threads. Bhowmick et al. [8] recently introduced a shared-memory implementation of Louvain, which updates the community on-the-fly, but with limited scalability. Staudt et al. [10] presented an alternative approach based on label propagation to parallelize Louvain. Recently, Lu et al. [11] proposed heuristics using OpenMP for Louvain. These approaches can only be utilized in shared-memory architectures, which unfortunately can only run small to medium-sized graphs. Our approach differs from this body of related work on shared memory machines because we can use distributed memory systems, therefore scaling to much larger graph configurations.

Soman et al. [45] implemented a GPU algorithm based on the label propagation algorithm [46]. Cheong et al. [42] mapped Louvain onto a GPU, reaching a 5X speedup with a single GPU and an additional 2X speedup with multiple GPUs. However this algorithm focuses on the first pass (the *outer loop*) and it does not report the capability of abstracting the hierarchical level community structure, which is one of the important features of the original algorithm. Due to the memory limitation of the server, it examines the speedup with random sampling for large graphs. Zhang et al. [47] proposed an algorithm that built upon the mutual update between network topology and topology-based proximity. Their work generated communities through a self-organizing process that distributes the workload among thousands of machines. The approach has a large search space per iteration which might lead to high overall compute time. Yang et al. [48] implemented a parallel community detection scheme on MapReduce [49], which is based on maximal cliques. However this approach suffered from runtime complexity and led to poor performance. Ovelgönne et al. [12] applied an ensemble learning scheme for community detection and presented a distributed implementation using label propagation on top of Hadoop, as discussed in the previous Section. Wickramaarachchi et al. [13] parallelized the first iteration of Louvain using MPI. This approach has a demanding memory consumption, 1TB for a graph with 60 Million edges, which limits the scalability. Using the same input configuration, our parallel algorithm only requires 8GB of memory. All those algorithms fail to unfold the hierarchical organization, which is an important feature [50] displayed by most networked systems in the real world [1].

Compared with all the above parallel solutions, our approach is designed for highly scalable distributed memory systems; it can be scaled to more than 8,192 BG/Q nodes. Meanwhile, the hashing-based, vertex-level parallelism and the greedy

optimization help our algorithm preserve the same convergence properties and perform accurate community analysis efficiently.

VII. Conclusions

We have presented a parallel version of the Louvain algorithm for fast community detection on distributed memory systems. Along with this algorithm, we have introduced a novel hash-based strategy to efficiently store and process dynamic graphs. Our parallel algorithm preserves the same convergence, modularity and community properties as the original sequential Louvain algorithm. We have performed an extensive performance evaluation on a wide variety of real-world social graphs and synthetic graphs, using a Power7-IH system and a massively parallel machine, Blue Gene/Q. The proposed algorithm can efficiently analyze R-MAT and BTER graphs with 138 billion edges on 8,192 BG/Q nodes and 1,024 P7-IH nodes. Our study indicates the effectiveness of distributed memory systems in supporting massive graph computations. We believe that our techniques can also be used to analyze other large-scale dynamic graph problems.

References

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

[2] M. E. J. Newman and M. Girvan, "Finding and Evaluating Community Structure in Networks," *Physical Review E*, vol. 69, no. 2, p. 026113, February 2004.

[3] V. Blondel, J. Guillaume, R. Lambiotte, and E. Mech, "Fast unfolding of communities in large networks," *J. Stat. Mech*, p. P10008, 2008.

[4] R. Guimerà, M. Sales-Pardo, and L. A. N. Amaral, "Modularity from fluctuations in random graphs and complex networks," *Phys. Rev. E*, vol. 70, p. 025101, Aug 2004.

[5] J. Duch and A. Arenas, "Community detection in complex networks using extremal optimization," *Physical Review E*, vol. 72, p. 027104, 2005.

[6] J. Ruan and W. Zhang, "An efficient spectral algorithm for network community discovery and its applications to biological and social networks," in *ICDM*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 643–648.

[7] E. J. Riedy, D. A. Bader, and H. Meyerhenke, "Scalable multi-threaded community detection in social networks," in *IPDPS Workshops*, 2012, pp. 1619–1628.

[8] S. Bhowmick and S. Srinivasan, "A template for parallelizing the louvain method," in *Dynamics on and of Complex Networks, Volume 2*. Birkhauser Verlag GmbH, 2013.

[9] E. L. Martelot and C. Hankin, "Fast multi-scale community detection based on local criteria within a multi-threaded algorithm," *CoRR*, 2013.

[10] C. Staudt and H. Meyerhenke, "Engineering high-performance community detection heuristics for massive graphs," *CoRR*, 2013.

[11] H. Lu, A. Kalyanaraman, M. Halappanavar, and S. Choudhury, "Parallel heuristics for scalable community," in *IPDPS Workshops*, Phoenix, AZ, USA, 2014, pp. 1375–1385.

[12] M. Ovelgönne, "Distributed community detection in web-scale networks," in *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ser. ASONAM '13. New York, NY, USA: ACM, 2013, pp. 66–73.

[13] C. Wickramaarachchi, M. Frincu, P. Small, and V. K. Prasanna, "Fast parallel algorithm for unfolding of communities in large graphs," in *proceeding of: 18th IEEE High Performance Extreme Computing Conference*, Waltham, MA, USA, 2014.

[14] J. M. Pujol, V. Erramilli, and P. Rodriguez, "Divide and conquer: Partitioning online social networks," *CoRR*, 2009.

[15] T. Raeder and N. V. Chawla, "Market basket analysis with networks." *Social Netw. Analys. Mining*, vol. 1, no. 2, pp. 97–113, 2011.

[16] D. Meunier, R. Lambiotte, A. Fornito, K. D. Ersche, and E. T. Bullmore, "Hierarchical modularity in human brain functional networks," 2010.

[17] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[18] R. Rajamony, L. B. Arimilli, and K. Gildea, "Percs: the ibm power7-ih high-performance computing system," *IBM J. Res. Dev.*, vol. 55, no. 3, pp. 233–244, May 2011.

[19] M. E. J. Newman, "Communities, modules and large-scale structure in networks," *Nature Physics*, vol. 8, pp. 25–31, Jan. 2012.

[20] S. E. Schaeffer, "Survey: Graph clustering," *Comput. Sci. Rev.*, vol. 1, no. 1, pp. 27–64, Aug. 2007.

[21] M. E. J. Newman, "Analysis of weighted networks," *Phys. Rev. E*, vol. 70, no. 5, p. 056131, Nov. 2004.

[22] M. Newman, "Fast algorithm for detecting community structure in networks," *Physical Review E*, vol. 69, September 2003.

[23] U. Brandes, D. Delling, M. Gaertler, R. Goerke, M. Hoefer, Z. Nikoloski, and D. Wagner, "Maximizing modularity is hard," 2006.

[24] A. Arenas, J. Duch, A. Fernandez, and S. Gómez, "Size reduction of complex networks preserving modularity," *CoRR*, 2007.

[25] R. A. Dunlap, *The Golden Ratio and Fibonacci Numbers*. World Scientific Publishing, 1997.

[26] A. Lancichinetti and S. Fortunato, "Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities," *Phys. Rev. E*, vol. 80, p. 016118, Jul 2009.

[27] F. Checconi and F. Petrini, "Traversing Trillions of Edges in Real-time: Graph Exploration on Large-Scale Parallel Machines," in *IPDPS*, Phoenix, AZ, May 2014.

[28] V. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal, "Scalable Single Source Shortest Path algorithms for Massively Parallel Systems," in *IPDPS*, Phoenix, AZ, May 2014.

[29] X. Que, F. Checconi, and F. Petrini, "Performance Analysis of Graph Algorithms on P7IH," in *Proceedings of International Supercomputing Conference (ISC'14)*, June 2014.

[30] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, 2012.

[31] R. Albert, H. Jeong, and A.-L. Barabási, "The diameter of the world wide web," *CoRR*, 1999.

[32] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *WWW*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[33] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[34] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW*. New York, NY, USA: ACM, 2010, pp. 591–600.

[35] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM'04)*, Lake Buena Vista, FL, April 2004.

[36] C. Seshadhri, T. G. Kolda, and A. Pinar, "Community structure and scale-free collections of erdös-rényi graphs," *Phys. Rev. E*, vol. 85, p. 056109, May 2012.

[37] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *CoRR*, 2013.

[38] J. Xie, S. Kelley, and B. K. Szymanski, "Overlapping community detection in networks: the state of the art and comparative study," *ACM Computing Surveys*, vol. 45, no. 4, 2013.

[39] "Linear Congruential Hash," http://www.cs.cmu.edu/afs/cs/academic/class/15210-f11/www/lectures/25/lecture25.pdf.

[40] "Load Factor," http://209.237.84.181:7600/index.php/Hash_Tables#Load_Factor.

[41] "The Louvain method for community detection in large networks," http://perso.uclouvain.be/vincent.blondel/research/louvain.html.

[42] C. Y. Cheong, H. P. Huynh, D. Lo, and R. S. M. Goh, "Hierarchical parallel algorithm for modularity-based community detection using gpus," in *Euro-Par*, 2013, pp. 775–787.

[43] A. Barrat and M. Weigt, "On the properties of small-world network models," *EUROP.PHYS.J.B*, vol. 13, p. 547, 2000.

[44] D. Watts and S. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, no. 393, pp. 440–442, 1998.

[45] J. Soman and A. Narang, "Fast community detection algorithm with gpus and multicore architectures," in *IPDPS*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 568–579.

[46] U. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks." *Phys Rev E Stat Nonlin Soft Matter Phys*, vol. 76, no. 3 Pt 2, p. 036106, 2007.

[47] Y. Zhang, J. Wang, Y. Wang, and L. Zhou, "Parallel community detection on large networks with propinquity dynamics," in *ACM SIGKDD*. New York, NY, USA: ACM, 2009, pp. 997–1006.

[48] S. Yang, B. Wang, H. Zhao, and B. Wu, "Efficient dense structure mining using mapreduce," in *ICDM*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 332–337.

[49] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Sixth Symp. on Operating System Design and Implementation (OSDI)*, pp. 137–150, Dec. 2004.

[50] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Statistical properties of community structure in large social and information networks," in *Proceedings of the 17th international conference on World Wide Web*. New York, NY, USA: ACM, 2008, pp. 695–704.