Information Sciences 508 (2020) 200–213

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Towards a distributed local-search approach for partitioning large-scale social networks

Bin Zheng[a], Ouyang Liu[a], Jing Li[b], Yong Lin[c], Chong Chang[d], Bo Li[e,*], Tefeng Chen[e], Hao Peng[e]

[a] State Grid Zhejiang Electric Power Company, Zhejiang, China
[b] State Grid Jiaxing Electric Power Supply Company, China
[c] State Grid Taizhou Electric Power Supply Company, China
[d] State Grid Shaoxing Electric Power Supply Company, China
[e] School of Computer Science and Engineering, Beihang University, Beijing 100191, China

A R T I C L E   I N F O

Article history:
Received 3 May 2019
Revised 21 July 2019
Accepted 6 August 2019
Available online 27 August 2019

Keywords:
Graph partitioning
Social network
Local search algorithm

A B S T R A C T

Large-scale social graph data poses significant challenges for social analytic tools to monitor and analyze social networks. A feasible solution is to parallelize the computation and leverage distributed graph computing frameworks to process such big data. However, it is nontrivial to partition social graphs into multiple parts so that they can be computed on distributed platforms. In this paper, we propose a distributed local search algorithm, named dLS, which enables quality and efficient partition of large-scale social graphs. With the vertex-centric computing model, dLS can achieve massive parallelism. We employ a distributed graph coloring strategy to differentiate neighbor nodes and avoid interference during the parallel execution of each vertex. We convert the original graph into a small graph, *Quotient Network*, and obtain local search solution from processing the *Quotient Network*, thus further improving the partition quality and efficiency of dLS. We have evaluated the performance of dLS experimentally using real-life and synthetic social graphs, and the results show that dLS outperforms two state-of-the-art algorithms in terms of partition quality and efficiency.

1. Introduction

Security analytics, such as terrorist network identification [23], fraud detection [20] and spam mitigation [2], plays an important role in identifying potential threats and enhancing the security of social networks. However, as the scale of social networks increases, traditional security analytics approaches have shown weaknesses in handling the huge data generated by social networks.

A feasible solution is to partition the data and leverage parallel computation, e.g., MapReduce [4] and Pregel [16] and GraphLab [15] to further analyze the data. Data partitioning is the prerequisite for parallel computation, the quality of which greatly affects the efficiency of algorithms running on those parallel computing platforms. Since social networks can generally be represented by graphs, partitioning social networks can be viewed as a Graph Partitioning (GP) problem.

* Corresponding author.
E-mail addresses: bin_zheng@zj.sgcc.com.cn (B. Zheng), ouyang_liu@zj.sgcc.com.cn (O. Liu), jx72303@163.com (J. Li), 564839746@qq.com (Y. Lin), 22350378@qq.com (C. Chang), libo@act.buaa.edu.cn (B. Li), chentf@act.buaa.edu.cn (T. Chen), penghao@act.buaa.edu.cn (H. Peng).

https://doi.org/10.1016/j.ins.2019.08.024
0020-0255/© 2019 Elsevier Inc. All rights reserved.

GP problem has been extensively studied in the past decades [5,10,13,21,29]. Finding an optimal solution for this problem is NP-Complete [6], therefore many heuristics are proposed to seek sub-optimal solutions. Among the heuristics, the most famous family is local search algorithms (such as KL [13] and FM [5]) which try to minimize edge cuts and preserve partition balance by swapping selected pairs of nodes. Although local search algorithms can produce good partition quality, it is costly and only suitable for partitioning small graphs. In order to scale up to larger graphs, multi-level partition strategy has been proposed. Many well-known partitioning tools are based on it, such as Metis [10], JOSTLE [29] and Stoch [21]. The idea of the multi-level approach is to recursively coarsen a large graph into smaller graphs and then apply initial partitioning algorithms such as KL and FM on the smallest graph. Then uncoarsen the graph back to the original graph with local refinement algorithms. These algorithms are further parallelized to handle even larger graphs, including ParMetis [12] and Pt-Scotch [3].

Though the existing graph partitioning algorithms are continuously improving, there is still a gap between the huge size of data generated by large-scale social networks and the lack of capability of state-of-the-art partitioning algorithms. Social graphs differ from traditional graphs in that social graphs usually follow Power-law distribution and have Skew characteristics. Partitioning large-scale social graphs remains challenging in three aspects: (1) existing local search algorithms are incapable of dealing with large-scale social graphs due to their weakness of sequential execution; (2) existing coarsening algorithms do not effectively shrink social networks to small graphs that could be directly dealt with initial partitioning algorithms; (3) multi-level partitioning approaches may have bad coarsening results on some scale-free networks [9,24] (especially social networks), which could destruct the structure properties of the original graph. Consequence is the huge increase of computation overhead in the refinement phase.

To address the above challenges, we propose a distributed local search graph partitioning approach, called dLS, which trades-off between the quality of partitioning and the efficiency of processing. We choose local search because it is not only used as an independent partitioning algorithm in many application scenarios, but also serves as the initial partitioning and refinement algorithms for multi-level partitioning algorithms. dLS can directly partition social graphs with billions of nodes and edges in acceptable time. dLS can also be used by multi-level partitioning solutions to improve the efficiency of the refinement phase and enhance the scalability of the initial partitioning algorithms so that coarsening phase can be completed within reasonable iterations when handling big graphs. dLS is designed following the vertex-centric model which enables it to be easily adapted to most distributed graph-computing platforms [7,15,16,31]. Moreover, it is desirable for distributed graph computing platforms to further improve the partition quality after initial partitioning using random or hash-based approaches. Our main contributions are briefly summarized as follows:

1. We propose and implement a distributed heuristic graph partitioning algorithm for large-scale social networks under a vertex-centric framework.
2. We design a novel model of local search for graph partitioning to improve the partitioning in batch.
3. Empirical results of experiments on real large social networks and synthetic graphs demonstrate the effectiveness and efficiency of our graph partitioning approach.

The rest of this paper is organized as follows. In the following Section, we give preliminary knowledge about graph partitioning problem and FM-like algorithm. In Section 3, we present our distributed graph partitioning approach dLS. The model adopted by dLS is introduced in Section 4. Section 5 includes several optimization techniques for dLS. We evaluate dLS by extensive experiments in Section 6. Section 7 overviews related work and Section 8 concludes the paper.

2. Preliminaries

In this Section, firstly we formulate the graph partitioning problem. Then we introduce local search algorithms and summarize the challenges of parallelizing local search algorithm as the preliminary knowledge of our algorithm.

2.1. Graph partitioning

Given an undirected graph $G = (V, E)$ with the set of vertices V and the set of edges $E \subset V \times V$, $n = |V|$ and $m = |E|$. A k-way partitioning of G divides V into k non-empty disjoint subsets $V_1, V_2, \ldots, V_k$, each of which is referred to as a partition P, meaning that every node is assigned to a unique partition. Edge which connects nodes in different partitions is considered to be a cut c and the number of *cut* edges is defined as the *edge cut* of a partitioning. The balance of a k-way partitioning measures how equal-sized the k partitions are and is defined as the ratio $|V_m|/|V|$, where V_m has the maximum nodes among the k subsets of V. Usually, the balance is upper bounded by $1 + \varepsilon$,where ε is a preset imbalance ratio.

The *k-way graph partitioning* is a combinational optimization problem of finding a partition P of V into k subsets with minimum edge-cut subject to the balance constraint. Let $c(e) = 1$ denote that edge e is a cut edge that associates two vertices in different partitions, and $c(e) = 0$ otherwise. Now we formulate the k-way graph partitioning problem as follows:

$$\arg\min \sum_{i=1}^{m} c(e_i) \tag{1}$$

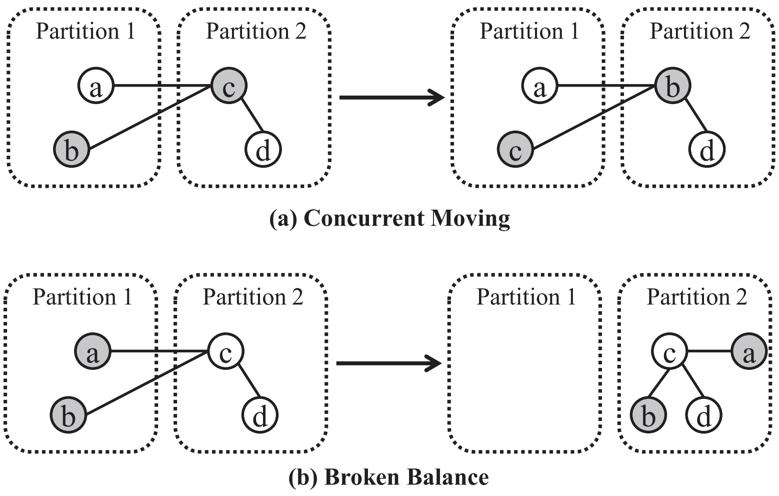

Fig. 1. Two problems when parallelizing FM algorithms.

$$s.t. \max_{i\in[k]} |V_i| \le (1+\varepsilon)\frac{n}{k} \quad and \quad \forall_{i,j\in[k]} : i \ne j \Rightarrow V_i \bigcap V_j = \emptyset \tag{2}$$

2.2. Local search partitioning algorithm

Local search is one of the meta-heuristic strategies to solve computation-intensive optimization problems. As a NP-Complete optimization problem, graph partitioning can be effectively solved by local search algorithms. A local search graph partitioning algorithm begins with an initial solution and then iteratively shifts to a better solution by moving vertices among graph partitions. More specifically, it starts from an initial partition and iteratively moves vertices between partitions to improve the edge-cut.

FM [5] algorithm is initially proposed for bisectual partitioning. It is an iterative-improvement approach that begins with an initial partition and iteratively moves vertices between partitions to improve the edge-cut under the constraint of balance. A round of FM algorithm runs as follows. Initially, all vertices are unmarked. For each vertex u, this algorithm computes the gain which is the decrease in the edge-cut obtained by moving u to other partitions. In each step of the algorithm, the vertex with the highest gain is marked and moved to its' target partitions if the size of that partition does not exceed the max size permitted. After the move, the gain of neighbors of the moved vertex is updated.

The algorithm keeps moving vertices until no unmarked vertex can be moved without violating the size constraint. After that, the algorithm unmarks all vertices and go to a next round if there is room for improvement. Note that the algorithm allows moving a vertex that has negative gain within several steps, which leads to an intermediate increase in the edge-cut but helps climbing out of local minima and hopefully obtain a better final result. FM has been extended to be used in k-way graph partitioning [11,25], where a vertex will move to the partition that can obtain highest gain among other k-1 partitions.

2.3. Challenges of parallelizing local search algorithms

Local search partitioning algorithms are computation-intensive. For instance, the time complexity of KL algorithm is $O(|V|^3)$. Though the variant version of KL, FM algorithm, reduces the time complexity to $O(|E|)$, it is still very costly for local search algorithms to process large-scale social networks with billions of nodes and edges. An intuitive idea is to leverage parallelization to accelerate local search algorithms and achieve scalability. However, it is non-trivial to parallelize local search algorithms since they are inherently sequential. In Fig. 1, we take FM algorithm as an example and identify two challenges when parallelizing FM algorithms.

Concurrently Moving. The computation of the gain of a vertex is based on the partitioning assignment of its' neighbors, which means moving adjacent vertices at a time may not reduce edge-cut and even increase it when the input graph is weighted, in which case the edge-cut is the sum of weights of cut edges. For example, in Fig. 1(a), moving node b and c individually will lead to edge-cut reduction by 1 respectively. However, if both moves are performed concurrently, the overall edge-cut is still 2.

Balance Broken. Moving vertices in parallel may even violate the size constraint, a global variable. For example, in Fig. 1(b), moving node a and b individually will not exceed the maximum capacity of a partition where we assume it as 3. However, if both moves are performed concurrently, the size of partition j will increase to 4 while partition i becomes empty.

Due to the difficulty of parallelization, most of the existing local search partitioning algorithms are sequential although they use some optimization techniques to reduce runtime. There are some parallel formulation of the algorithm [12,14], but they still cannot solve the two problems aforementioned well.

Table 1
Notation of Algorithm 1.

Notation	Description
color(u)	the color of vertex u.
partition(u)	the partition vertex u belongs to.
gain(u)	the reduction in edge-cut if vertex u move to another partitions.
target(u)	the partition vertex u wants to move into.
N(u)	the neighbors of vertex u.
B[i]	the size of partition i.

3. Distributed local search algorithm

We present the distributed formulation of the proposed dLS for partitioning large-scale social networks. We firstly explain our basic idea to address the challenges aforementioned and then give an overview of dLS.

3.1. Basic idea

1) Guarantee data consistency. We employ graph coloring to schedule vertices to eliminate the concurrent moving of adjacent vertices because they are assigned to different colors. In particular, one single round of dLS is divided into c iterations, where c is the number of colors of the graph. That is to say, local search is performed in a group at a time and each group contains the vertices in the same color. Therefore, each group forms an independent set, which means the total reduction in the edge-cut achieved by moving arbitrary vertices at the same time is equal to the sum of the edge-cut reductions achieved by moving these vertices in sequence. After each iteration, a global synchronization is implemented.

2) Maintain balance. Instead of moving vertices individually, we move vertices with a global view. It is a MapReduce-like solution, which decentralizes the gain computing but approves the move of vertices centrally to maintain balance. Specially, after computing the gains, we collect vertices with non-negative gains and then create a network to model the vertices moving between partitions. For ease of reference, we denote this network as a *QuotientNetwork* and define it as follows:

Definition 1. Quotient Network: Quotient network is a weighted directed graph $G_q = (V_q, E_q)$, where (1) each vertex with weight v_w represents a partition with size vw of the partitioning; (2) each edge (u, v) with weight e_w indicates there are e_w vertices that would like to move from partition u to partition v. (3) there is a size constraint *cons* in weight of each vertex v_q.

Thus, we can take a whole view of the move of vertices between partitions from the small network. We simulate the move of vertices in the quotient network to coordinate the number of vertices moving in and out from a partition under the constraint of partition size, like a circulation problem in the network. During this process, we modify the quotient work along with the move of vertices. For example, when one vertex is approved to move we subtract the weight of the corresponding edge by one, and if vertices comes from the same partition and would like to move into the same target partition can all move, we delete the corresponding edge. As a result, each edge that remains in the quotient work shows the number of vertices that cannot move and also the source and target partition. Therefore, we exclude corresponding amount of vertices moving in this type. Finally, For the vertices selected to move, we update their assignment concurrently.

From the above analysis, each vertex computes its gain and updates its own data independently in the coloring order and then delivers the task of probing whether its move would violate balance or not to a “manager”, which only needs to access local information of neighbors and itself. Therefore, such idea is very suitable to be implemented by the vertex-centric framework, where each vertex can be viewed as an independent execution unit.

3.2. Overview

We outline the major steps of the baseline algorithm dLS_b in Algorithm 1 and some notations used in the pseudocode are listed in Table 1. Given a graph G, the algorithm divides G into k partitions with imbalance ε. Initially, each vertex v is assigned to a unique partition P_i randomly. Then we perform graph coloring on G and each v will be assigned a color c that is different with adjacent vertices. After that vertices are iteratively executed by group according to their colors to reduce the edge-cut of the partition. In the end, when there is no room for improvement or meeting stop criteria, the algorithm output a k-way partitioning of G.

During the processing of each color, each vertex executes in 3 phases. In the first step, each vertex computes the highest gain which is the most reduction in edge-cut if the vertex moves to the partition that maximum neighbors are assigned to except its' own partition, and record the partition to be used for moving. Then vertices that have non-negative gain are collected as input to create the quotient network G_q. In the second step, we run Algorithm 2 to process the quotient network which will return the reduced quotient network G'_q. In G'_q, each edge (u, v) with weight e_w indicates there are e_w vertices remaining that cannot be moved from partition u to partition v, so we exclude e_w vertices with smaller gain in this type of move. Finally, the remaining vertices move into their target partition at a time. Note that we don't need to move the vertices

Algorithm 1 dLS$_b$.

```
Input: $G(V, E)$, $k$, $\varepsilon$
Output: A k-way partitioning on V
1: $P \leftarrow$ partitioning that uniform randomly assign vertices to k partitions
2: perform graph coloring on $G$, $c \leftarrow$ the number of colors
3: repeat
4:   for $i \in [1, c]$ do
5:     $V_i \leftarrow \{u|u \in V \& color(u) = i\}$
6:     for $u \in V_i$ do
7:       $target(u) \leftarrow$the partition which maximum neighbors of u assigned to except its own partition, break ties randomly
8:       $gain(u) \leftarrow |\{v|v \in N(u)\ \&\ partition(v) = target(u)\}| - |\{v|v \in N(u)\ \&\ partition(v) = partition(u)\}|$
9:     end for
10:    $V_m \leftarrow \{u|u \in V_i\ \&\ gain(u) \geq 0\}$
11:    $V_q \leftarrow \{(1, B[1]), (2, B[2]), \cdots, (k, B[k])\}$, $E_q \leftarrow \emptyset$
12:    for $s \in [1, k]$ do
13:      for $t \in [1, k]$ do
14:        $w \leftarrow |\{u|u \in V_m, partition(u) = s\ \&\ target(u) = t\}|$
15:        if $w > 0$ then
16:          $E_q \leftarrow E_q \cup \{(s, t, w)\}$;
17:        end if
18:      end for
19:    end for
20:    $G' = (V'_q, E'_q) \leftarrow$ run Algorithm 2 on $G_q = (V_q, E_q)$
21:    for $(s, t, w) \in E'_q$ do
22:      $V_r \leftarrow \{u|u \in V_m, partition(u) = s\ \&\ target(u) = t\}$
23:      sort $V_r$ by the gain of vertices in ascending order.
24:      $V_m \leftarrow V_m - \{u|u \in V_r$ and the order of u is not more than w$\}$
25:    end for
26:    for $u \in V_m$ do
27:      $partition(u) \leftarrow target(u)$
28:    end for
29:  end for
30: until there is no improvement on edge-cut of P or the algorithm has run with the maximum number of iterations
31: return P
```

physically as they change partition but only the partition number associated with each vertex is changed. After all colors are processed, we count the edge-cut and stop execution when the edge-cut cannot be improved anymore or a maximum number of rounds has been reached, otherwise next round begins.

3.3. Coloring algorithm

We employ the asynchronous parallel graph coloring algorithm JP-LLF [8] to color a large social graph. It is a greedy heuristic algorithm that a vertex colors itself with the smallest color number that has not been colored by its' neighbors, which use the fewest colors possible(at most $\Delta + 1$, where Δ is the maximum degree of a graph) and works well in practice although the problem of finding an optimal coloring of a graph is NP-Complete. In addition, it is highly efficient due to the asynchronous parallel computation of coloring.

4. Quotient network processing

We propose an algorithm to process the quotient network, which is outlined in Algorithm 2 and some notations used are listed in Table 2. It has three phases. In the first two steps, we circulate vertices among partitions, where each partition receives and sends the same amount vertices so the size of partitions will still the same. In the third step, we move the vertices whose target partitions haven't reached the maximum capacity yet. In the following, we explain our process with the example shown in Fig. 2, where we assume the maximum capacity of a partition is 106.

4.1. Balance move

1) Exchange. When both two partitions have vertices that should be moved into opposite partitions, these vertices should be exchanged between the two partitions. For example, in Fig. 2(a), 11 vertices in partition 1 and 28 vertices in partition

Algorithm 2 Quotient network process.

```
Input: a weight directed graph $G(V, E, VW, EW)$, the maximum capacity of partitions $M_w$
Output: the modified quotient network $G'$
1:
                                1.Exchange
2: for $u \in V$ do
3:     for $v \in N^{out}(u)$ and $v \in N^{in}(u)$ do
4:         $w \leftarrow \min\{w(u, v), w(v, u)\}$
5:         $w(u, v) \leftarrow w(u, v) - w$, $w(v, u) \leftarrow w(v, u) - w$
6:         if $w(u, v) = 0$ then
7:             delete v from $N^{out}(u)$
8:         end if
9:         if $w(v, u) = 0$ then
10:            delete v from $N^{in}(u)$
11:        end if
12:    end for
13: end for
14:
                                2.Circle Move
15: mark all vertices $u \in V$ active
16: for $u \in V$ do
17:     $A^{in}(u) \leftarrow N^{in}(u)$, $A^{out}(u) \leftarrow N^{out}(u)$
18:     if $|A^{in}(u)| = 0$ or $|A^{out}(u)| = 0$ then
19:         Inactive(u)
20:     end if
21:     for $u \in V$ do
22:         if $u$ is active then
23:             DFS(u)
24:         end if
25:     end for
26: end for
27:
                                3.Imbalance Move
28: mark all vertices $u \in V$ active
29: for $u \in V$ do
30:     $A^{in}(u) \leftarrow N^{in}(u), A^{out}(u) \leftarrow N^{out}(u)$
31:     if $|A^{out}(u)| = 0$ and $|A^{in}(u)| > 0$ then
32:         Income(u)
33:     end if
34: end for
35:
                                Function Inactive(u)
36: mark $u$ inactive
37: for $v \in A^{out}(u)$ do
38:     delete $u$ from $A^{in}(v)$
39:     if $|A^{in}(v)| = 0$ then
40:         Inactive(v)
41:     end if
42: end for
43: for $v \in A^{in}(u)$ do
44:     delete $u$ from $A^{out}(v)$
45:     if $|A^{out}(v)| = 0$ then
46:         Inactive(v)
47:     end if
48: end for
49:
                                Function Income(u)
50: $R_w \leftarrow M_w - w(u)$
51: $T_{in} \leftarrow$ sum of the weight of all in-edges of $u$ whose source vertices are active
52: for $v \in A^{in}(u)$ do
53:     if $R_w \geq T_{in}$ then
54:         $w(v, u) \leftarrow 0$
55:         delete v from $N^{in}(u)$
56:         delete u from $N^{out}(v)$
57:     else
58:         $w(v, u) \leftarrow w(v, u) - (w(v, u)/T^{in}) * R_w$
59:     end if
60:     mark $u$ inactive
61:     delete $u$ from $A^{out}(v)$
62:     if $|A^{out}(v)| = 0$ and $|A^{in}(v)| > 0$ then
63:         Income(v)
64:     end if
65: end for
```

Table 2
Notation of Algorithm 2.

Notation	Description
w(u)	weight of vertex u.
w(u,v)	weight of edge (u,v).
N_{out}(u),N_{in}(u)	out-neighbors and in-neighbors of vertex u.
A_{out}(u),A_{in}(u)	the active out-neighbors and active in-neighbors of vertex u respectively.

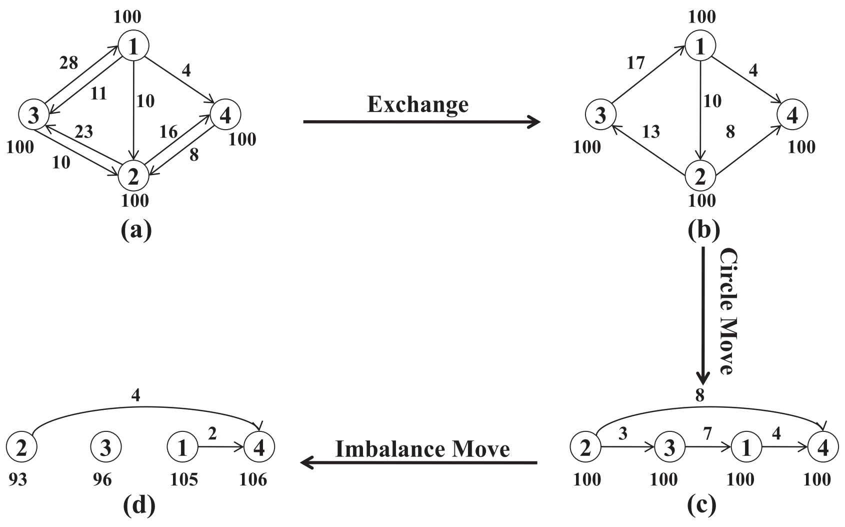

Fig. 2. Evolution of a sample quotient network.

3 want to move into opposite partition respectively. In order to maintain balance during the partitioning, we exchange the same number of vertices between partitions so 11 vertices are exchanged between partition 1 and partition 3. Thus, the edge with smaller weight is deleted and the weight of the other edge is subtracted by the weight of the former. Specially, each vertex search neighbors that is both one of its in-neighbors and out-neighbors, and then update the weight of their associated edges and delete edges with zero finally(the first phase of Algorithm 2).

2) Circle move. When the flow of vertices among partitions form a circle, we can move the same number vertices along with these cyclic paths. The maximum amount of vertices to be moved is the minimum weight of edges in this circle. As shown in Fig. 2(b), for the circle $1 \to 2 \to 3 \to 1$, we move 10 vertices in all directions and update the weights of these edges, finally delete edges with zero weight.

The *circle move* phase in Algorithm 2 has two steps(the second phase of Algorithm 2). Initially, all vertices in the graph are marked active. Note that when a vertex lies in circles, both its in-degree and out-degree are no less than 1. Based on this observation, the first step is an iterative parallel procedure that determines whether there exists circle in the quotient network. In each iteration, vertices that have only active in-neighbors or active out-neighbors are marked inactive and their neighbors will be informed. As a result, all vertices are marked inactive if there is no circle in the graph, otherwise we should only search for cycles in the subgraph that is composed of remaining active vertices and edges that associate with two active vertices. In the second step, we use DFS algorithm to detect cycles, where we handle each circle once finding it instead of collecting all circles and handling them one by one. Because we can skip circles that go through invalid edges when these edges have been deleted (marked invalid in practice) by other circles that passed the same edges and have been handled earlier.

4.2. Imbalance move

After *exchange* and *circle move*, the quotient network becomes a directed acyclic graph (DAG) including one or more vertices whose out-degree is zero. That is to say, there is at least one partition without a single moving out vertex. We denote such partition as ending partition for ease of reference. We iteratively receive vertices in ending partitions. For an ending partition, it receives the vertices from more than one partition according to our algorithm. However, a situation that the ending partition has no enough capacity to hold all these vertices existed, so we move these vertices based on the to-be-moved vertices counts of different partitions. For example, in Fig. 2(c), both of partition 1 and 2 has 4 and 8 vertices that want to move into ending partition 4, and we move 2 vertices and 4 vertices into partition 4 from these partitions respectively and process partition 1 next. The pseudocode of this phase is presented in third phase of Algorithm 2, which

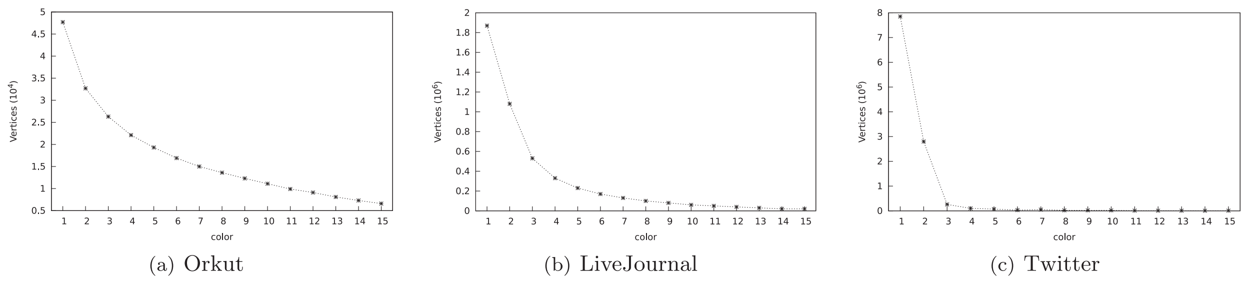

Fig. 3. The comparison results of coloring distribution for three social network datasets.

is an iterative parallel procedure. Finally, after the three phases, Algorithm 2 returns the reduced quotient network G_q' to Algorithm 1.

5. Optimizations and implementation

In this section, firstly we present a truncation strategy that improve the performance of dLS$_b$ algorithm without compromising the convergence guarantee and effectiveness. Then, we discuss several issues during the implement of our solution, where we investigate how a cache mechanism can reduce communication volume to speed up computation and present an aggregation method to parallel collect vertices. Finally we briefly introduce the mechanism of the distributed graph-computing framework we adopt. We refer dLS$_o$ as an optimized version of dLS$_b$ with truncation, cache and aggregation mechanisms.

5.1. Truncation

Social networks are usually power-law structures that most vertices are with small degrees, meaning that these vertices can be colored with a small number because the max color number of them is d+1(where d is their degree). In our experiments, we observe that the distribution of the number of vertices with the same color is right-skewed. The coloring results of three real-life social networks are shown in Fig. 3. We can see that most vertices reside within the first several colors. Therefore, dLS$_b$ can obtain high parallelism in these vertices but shows low efficiency when performing parallel computation in later colors with few vertices.

Based on this observation, we adopt a “truncation” operation, which takes these vertices with latter color order into consideration together and executes computation in all these vertices concurrently. As a result, it can significantly limit the required number of synchronization steps to improve efficiency. This strategy may lead to some vertices moving into the partition that most of its neighbors reside in due to data inconsistency with their adjacent vertices but help to escape a local optimum consequently. The results of our experiments verify this judgment that shows that this strategy enhance the effectiveness of our approach. Specially, we set a threshold θ that is the ratio of vertices that execute in color order and other remain vertices execute concurrently. We refer to the version of the algorithm that includes truncation as dLS$_{tr}$.

5.2. Cache

In our solution, when a vertex computes its’ gain, it needs to know the assignment information of all neighbors. In a vertex-centric framework, the vertex data is stored by its own so a vertex obtain neighbors’ data through sending request messages. Note that our solution is iterative-improvement and move vertices in batch so there are less and less vertices that will change their partition assignment. However, a vertex repeatedly sends request messages to all neighbors to gather their assignment information, but many of which remain unchanged, thereby wasting much time. Therefore, we maintain a cache of neighbors’ data from the previous gather for each vertex since the second round. And then, if a vertex changes its’ partition, it initiatively inform its’ neighbors instead of being requested by them and a neighbor that receive message will update its cache, which also help to skip vertices whose neighbors did not change their partition in the previous round to avoid redundant computations

5.3. Aggregation

dLS needs to collect a set of vertices that meet some conditions as preliminary operation to execute parallel computation in this set. For example, in each color, the set of vertices that has the same color should be collected in advance. In addition, in dLS, when the reduced quotient network remains edges we need to sort a set of vertices to exclude some vertices. A common way to collect them is to traverse all vertices sequentially and put a vertex into the set when it meet the condition, but it’s not suitable for large graph. Consequently, we can search vertices in parallel either by the same machine through multi-threading or by different machines and aggregate to results each unit found. Fig. 4 shows the process that how the

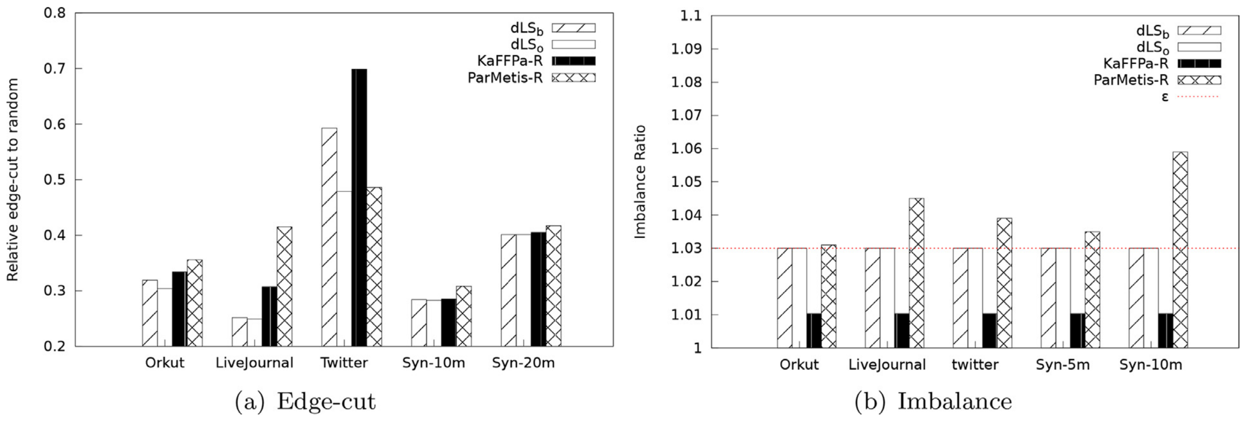

Fig. 4. Partitioning quality.

aggregation works. In particular, to collect vertices the rank of whose gain is within fixed range, we can collect the fixed number vertices in each unit and discard redundant vertices when it aggregates with the other unit.

5.4. *The vertex-centric framework*

Recall that dLS is built on the top of a general-purpose graph infrastructure-the vertex-centric framework. Most of current distributed graph processing systems adopts the frameworks and are mainly based on bulk synchronous parallel model [28], which provides a synchronous superstep model of computation and communication. We implement dLS in PowerGraph [7], which supports both bulk synchronous and asynchronous execution. PowerGraph adopts a GAS programming abstraction model, which splits one superstep into three minorsteps including gather, apply and scatter. In each superstep, each active vertex collects information from its' neighbors in gather step and then executes computation in apply step with the collected neighborhood information, finally sends messages to or activates its' neighbors in scatter step. From the above, PowerGraph is very suitable for dLS algorithm especially its' asynchronous parallel model can enhance the efficiency of coloring algorithm and processing the quotient network.

6. Experimental evaluation

We present experiment results in this section. We compare our baseline algorithm dLS_b and the optimized version dLS_o with two popular k-way local search partitioning algorithms. One is a sequential k-way partition algorithm, which is the default local refinement algorithm used by KaFFPa [25], denoted by KaffPa-R. The other one is a parallel version, which is the local refinement algorithm used by ParMetis [12], denoted as ParMetis-R. It's worth noting that we did not directly compare dLS with the KaFFPa and ParMetis since they are not in the same level. ParMetis and KaFFPa are two multi-level graph partitioning tools, while dLS is a local search partitioning algorithm. Actually, we have obtained the comparison results of execution time between dLS and the two partitioning tools. The two partitioning tools are much slower than dLS, and we omit this due to space limitation.

6.1. *Experimental settings*

6.1.1. *Metrics*

We measure edge-cut and imbalance for partitioning quality and runtime for performance. These metrics are summarized as follows:

- Edge-cut: the number of edges that have endpoints in different partitions.
- Imbalance: the ratio of the size of the largest partition compares against the average partition size.
- Runtime: the time expenditure of a algorithm running, not including the time cost by loading graph into memory

6.1.2. *Paramaeters*

The maximum imbalance ratio ε is set to 1.03 and the maximum iterations is set to 15 in all algorithms. In addition, we set θ (the ratio of vertices executed in color order) to 0.95 for dLS_{tr} through extensive experimental evaluations.

Table 3
Datasets.

Graph	$\|V\|$	$\|E\|$
Orkut	3,072,441	117,185,083
LiveJournal	4,846,609	42,851,237
Twitter	11,316,811	63,555,749
Syn-10m	10,000,000	153,462,384
Syn-20m	20,000,000	92,638,212

6.1.3. *Datasets*

We use three large real-life social networks: Orkut, LiveJournal[1],Twitter[2]. Orkut is a free on-line social network where users form friendship with each other. Orkut also allows users form a group open to other outside members. LiveJournal is a free on-line community with members, it allows members to maintain journals, individual and group blogs, and it allows people to declare who are their friends in this community. Twitter is a famous social nets website, and we use a published dataset collected by social network research group from Arizona State University. We also generate two synthetic power-law random networks using the generator of PowerGraph [7]. One has 5 million vertices with the alpha parameter in the power law distribution set to 1.85, the other one has 10 million vertices and the parameter is set to 2.1. All aforementioned graphs are listed in Table 3. Note that these graphs are directed graphs originally and we translate them into undirected graphs by eliminating one of doubles edges between two vertices randomly.

6.1.4. *Experimental environment*

All algorithms were implemented in C++, under Debian Linux 3.2.04 system. Each server is powered by two 8-cores Intel Xeon 2.00 GHz CPU with 64 GB memory. Unless stated otherwise, all algorithms produced 4-way partitions and each experiment ran 10 times and the average is reported here due to uncertainty nature of heuristic algorithms.

6.2. *Partitioning quality*

6.2.1. *Edge-cut*

Fig. 4 (a) shows the relative edge-cut of all algorithms to the random partitioning, where lower ratio means fewer edge-cut. As we can see, both of our algorithms significantly improve the edge-cut of the initial random partitioning, and dLS_o performs even better than dLS_b, which means our optimization strategies are effective. From the comparisons, both dLS_b and dLS_o produce partitions with fewer edge-cut than KaFFPa-R and ParMetis-R on all graphs except Twitter where dLS_b is not comparable to ParMetis-R but dLS_o is still better than ParMetis-R.

KaFFPa-R is a typical greedy algorithm because it repeatedly looks for the vertex with the highest gain from all unmoved vertices in current round and doesn't adopt any strategy to help climb out of a local optimum. Therefore, edge-cut, as the objective function of the algorithm, will not improve any more when it reaches a local optimum. In ParMetis-R, a graph is initially distributed among the processors and each round is split into two passes, once where vertices can only move from partitions with a lower label than the partition they belong to, and a second time where they can only move to partitions with higher label than the one they reside in. However, this method can only avoid concurrently moving of two adjacent vertices that would like to move into partitions opposite vertex reside in but still lead to severe data inconsistency.

6.2.2. *Imbalance*

The imbalance ratios of all partitions are given in Fig. 4(b). We can see that the imbalance ratios of our algorithms and KaFFPa-R do not exceed the maximum value preset and KaFFPa-R obtains better balance. However, ParMetis-R cannot obtain partitions with desired balance on all graphs. In addition, it will produce more skewed partitions when the number of partitions increases as the example on Twitter shown in Fig. 5. We found that ParMetis-R leads to severe imbalance in the first several rounds and it is very likely to converge to an imbalance ratio that is still higher than the preset maximum value though it improves its balance in later iterations.

In ParMetis-R, to maintain the balance, each processor creates a table that saves the number of vertices of each partition. A pass of the algorithm is split into three phases. Firstly, each processor individually performs the sequential local refinement routine on the vertices assigned to it. In addition, it records every vertex that changes its partition assignment and modifies the table with vertices moving during this process. Then, tables of all processors are gathered to ascertain the size of each partition and the global result is sent to every processor to update their tables. Finally, each processor will withdraw the move of vertices that have moved to the partition whose size exceeds the constraint. However, after vertices withdrawn goes back to their original partitions, new imbalance may occur when the "rational" room spared by these vertices has been occupied by newly moving vertices. In consequence, the imbalance of these partitions are very likely to be irreversible if few vertices that belong to them want to move out again.

[1] Download from http://snap.stanford.edu/index.html.
[2] Download from http://socialcomputing.asu.edu/datasets/Twitter.

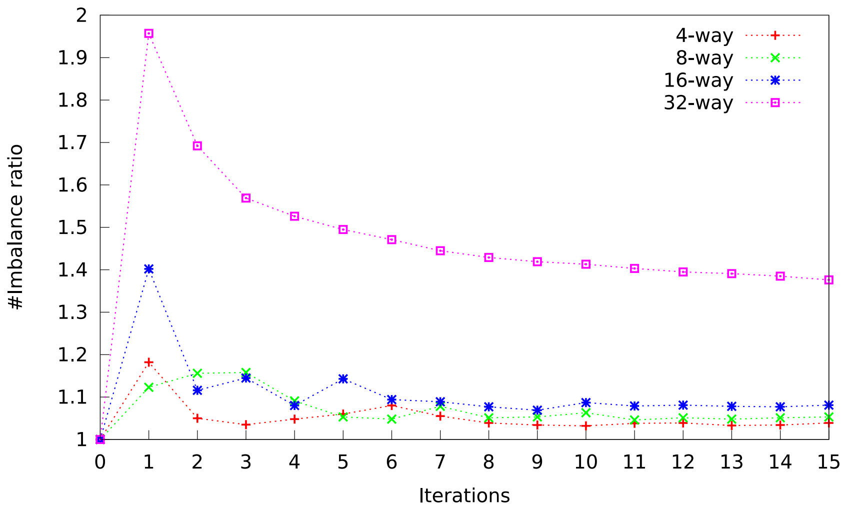

Fig. 5. Imbalance of ParMetis-R on Twitter.

Table 4
The "Runtime" comparison results of the four algorithms.

Graph	dLS_b	dLS_o	ParMetis-R	KaFFPa-R
Orkut	81.6	39.9	30.0	2106.9
LiveJournal	134.2	29.2	18.9	505.7
Twitter	157.8	33.5	43.1	22959.8
Syn-10m	57.2	53.5	29.7	23785.4
Syn-20m	63.4	63.3	27.8	13215.1

6.3. *Efficiency*

6.3.1. *Runtime*

Here, all algorithms were run on one machine. We present the runtime of all algorithms in Table 4. As shown, both of our algorithms and ParMetis-R consume significantly less time than KaFFPa-R. The runtime of dLS_o is less than dLS_b because all vertices are run within less iterations in dLS_o than dLS_b so that the parallelism is enhanced.

ParMetis-R needs the least time in most cases. Recall that each round of ParMetis-R is split into two passes and each pass only need twice inter-processor communication. One is to synchronize their tables that records size of partitions, the other one is to update the data of vertices that are processed by other processors. As a result, ParMetis-R usually costs less number of times of global synchronization in one round and obtains higher parallelism than our approach because all vertices are run during two passes in one round to perform local search concurrently. Therefore, ParMetis-R consumes less time than our approach, as expected.

However, dLS_o is comparable to ParMetis-R and even better on Twitter, where only vertices coloring the first 3 colors were executed in order. Therefore, we can conclude that dLS_o is effective and efficient for social graphs because the distribution results of the coloring of these graphs are always right-skewed. Consequently, a round of dLS_o can only be splited into several iterations.

6.3.2. *Convergence*

Fig. 6 shows the speed of convergence of different algorithms on Twitter. Results on other graphs are similar so they are not reported here due to lack of space. From the results, we can observe that both of our algorithms and KaFFPa-R converge faster than ParMetis-R. In addition, our approach can obtain most of edge-cut reduced in first several rounds so that we can further optimize it by early termination.

Recall that ParMetis-R withdraw move of vertices when they lead to unbalanced partition. But such way is easily lead to overreaction among processors. For example, provided that the size of a partition over the maximum capacity is m, but finally there may at most p*m vertices to be withdrawn when there are p processors. Therefore, ParMetis-R converge slowly because too many moves of vertices that can help to reduce edge-cut are canceled.

In summary, the results of aforementioned experiments show that dLS_o can effectively improve the edge-cut of initial random partitions as well as maintain the balance of partitions well within comparable runtime. In contrast, the main competitor of dLS_o, ParMetis-R, simply parallelizes the FM algorithm and doesn't solve the problem we discussed in Section 2. Though it runs faster than dLS_o on some graphs, it has to sacrifice at the expense of partitioning quality.

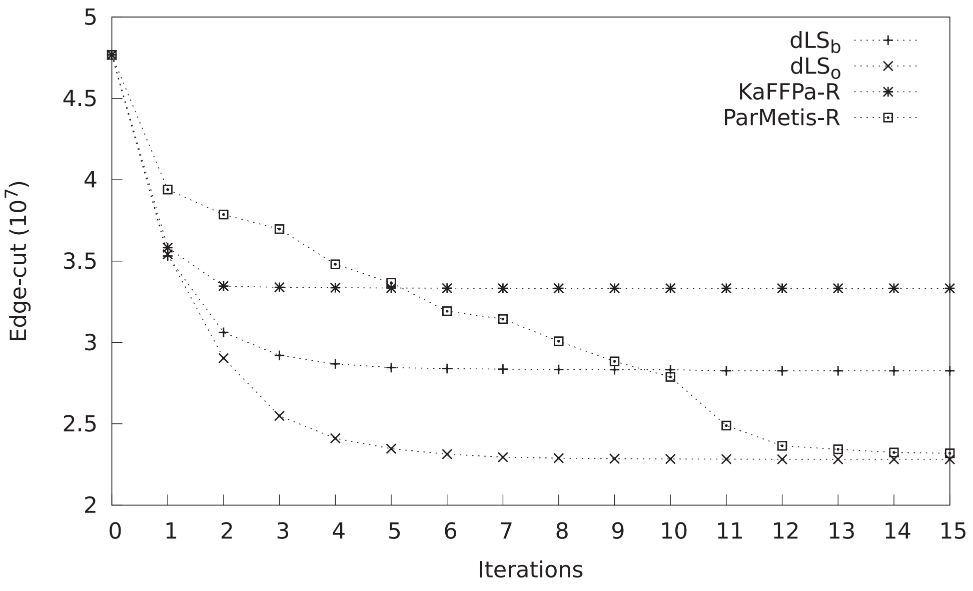

Fig. 6. Convergence of partitioning on Twitter.

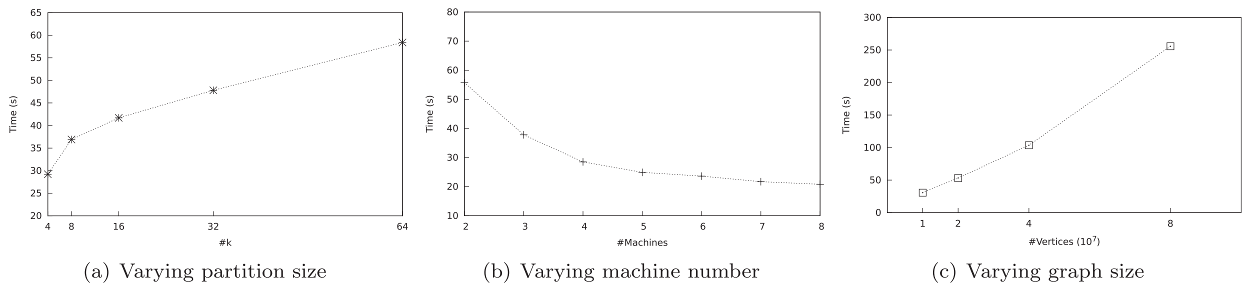

Fig. 7. Scalability of dLS$_o$.

6.4. Scalability

In this section, we evaluate the scalability of our approach. Only dLS$_o$ was examined because the previous experiments have proved that it is more effective and efficient than dLS$_b$.

6.4.1. Varying partition size

In this experiment, we investigated how the partition size k affect the running time of dLS$_o$. We varied the partition size from 4 to 64. As shown in Fig. 7(a), the runtime increases as k grows since vertices consume more time to ascertain the partition to which most of its neighbors belong from other k-1 partitions. Nevertheless, this ascent tendency will get less stark because vertices can exclude more of other k-1 partitions where not a single one of its neighbors lies in when k increases.

6.4.2. Varying machine number

Varying machine number. We ran dLS$_o$ on the LiveJournal graph by varying the number of machines from 2 to 8. Fig. 7(b) shows the experiment results. As shown, dLS$_o$ scales well with the increase of machines and the improvement is 2.7 times on 8 machines. With the increase of machines, the speed of runtime decreasing is getting slower because communication between machines costs more time which counteracts a part of time saved by additional processors.

6.4.3. Varying graph size

We generated four synthetic graphs with power-law degree distribution using the generator of PowerGraph and set the alpha parameter as 2.1 with varying the number of vertices from 10M to 80M. We run dLS$_o$ on all graphs with one machine to eliminate the effect of cost on communication between machines. The results are shown in Fig. 7(c), which tell us that dLS$_o$ scales well with the increase of the graph size.

7. Related work

In this section, firstly we review several local search graph partitioning algorithms, and then we present related work on large graph partitioning.

KL [13] and FM [5] are two typical aforementioned algorithms. They perform local search in rounds where a vertex can only move once per round. In each round of the KL [13] algorithm, it repeatedly looks for a pair of unmoved nodes in different partitions that can obtain the most reduction in edge-cut when their location is swapped. Instead of searching a pair of nodes at a step, FM [5] algorithm seeks the node with the highest gain from all unmoved vertices where the gain of a vertex is the decrease in the edge-cut obtained by moving it to another partition. But a vertex cannot move to its target partition if the move violates the size constraint of partitions. Finally, when all vertices have moved once, the algorithms keep doing the rounds until the best partitioning solution generated by the algorithm cannot be improved in a given number of steps.

An extension of FM algorithm in the case of k-way partitioning is described in [25], which uses a priority queue to look for the vertex with the highest gain (break ties randomly) to move each time. The queue is initialized with a subset of vertices whose neighbors reside in different partitions and then updated with the move of vertices. A round of the algorithm is stopped when the priority queue is empty or meeting the stopping criteria proposed by [19], which is derived using a random walk model. [11] proposed a k-way FM algorithm that simply traverses the vertices in a random order without maintaining a priority queue.

Due to the sequential nature of the greedy move-based local search, [12] made some significant diversions from the sequential algorithm [11] to realize a parallel formulation of k-way FM algorithm. The algorithm performs local search in all vertices that are evenly assigned to processors in a single round. Each round is split into two passes, once where vertices can only move from partitions with label lower than the partition they belong to, and a second time where they can only move to partitions with label higher than the one they reside in. However, this method can only avoid a part of adjacent vertices moving. In addition, the algorithm cannot maintain the partition balance well because each processor moves vertices locally with a backup of outdated record of partitions size which is updated once per pass. In order to address this issue, [14] proposed an approach that updates the record once every processor moves a fixed number of vertices, but it still cannot completely guarantee balance and it increase communication overheads. With the emergence of numerous large-scale graphs, the strong demand for methods to partition these graphs motivates many researchers to explore. Wang et al. [30] proposed a distributed coarsening algorithm derived from label propagation and applied it on the multilevel graph partitioning framework to improve the poor effectiveness of previous coarsening algorithms on graphs with community structure. [17] established the relationship between partition quality, tree decomposition theories and analytic centralities and proposed a distributed partitioning algorithm concerning the minimum communication volume partitioning problem. The algorithm transforms the input graph into a strictly smaller elimination tree and finds an upper-bounded partition of the original graph by decomposing the tree.

In summary, the results on the synthetic graphs with embedded communities sufficiently show that MLP can effectively leverage the community structure of graphs to generate a good partition with less memory and time. In contrast, METIS, which is based on the maximal matching method, is not community-aware when it coarsens a graph, heavily relies on costly refinement in the uncoarsening phase to ensure quality of the solution. As a result, METIS incurs more time and space costs.

Stanton and Kliot [26] first proposed the streaming graph partitioning method, which assumes the vertices to arrive in a stream with the set of edges and uses a heuristic rule to decide the placing of a vertex into one of the k partitions. It is an one-pass partitioning approach where a vertex is never relocated after it has been placed. The method shows high efficiency while compromising some effectiveness. Later, [27] improved the effectiveness of the method by a novel heuristic rule and [18] presented a restreaming model which partitions the input graph once again with the assignment information of the first pass. These approaches are efficient in handling streaming graph data; however, their partitioning quality is much worse than static partitioning methods. Our approach can be used as a complementary strategy which can periodically refine the partitioning quality of these approaches.

Other researchers [1,22] resorted to EDGE partition techniques to construct balanced partitions. [22] presented a random vertex placement for its simplicity of implementation. However, since the graph partitioning problem is considered to be intrinsically difficult, this random partition does not perform well. Bourse et al. [1] discussed the design rationale of performing an edge partition is to allow for more flexibility for processing load balancing by allowing a single vertex program to span multiple machines and reduction of communication and storage overhead by evenly assigning edges to machines. Since each edge is stored exactly once, changes to edge data do not need to be communicated. The EDGE partition approaches do show some advantages, however in current graph computing platforms, vertex-centric model is still the mainstream, and it is still of great importance to study the vertex-centric partition approaches.

8. Conclusion

In this paper, we propose a novel graph partitioning approach based on vertex-centric model to process large-scale social networks. A distributed local-search algorithm which can effectively accelerate the partitioning and preserve balance among partitions is designed. Our experimental study has verified the efficiency and effectiveness of our approach.

Declaration of Competing Interest

All authors of this manuscript have participated in planning and analysis of this study.
The contents of this manuscript have not been published previously.

Acknowledgments

The authors would like to thank the helpful suggestions from reviewers. The work reported in this paper has been supported by National Key R&D Program China (2018YFB0803500), the 2018 joint Research Foundation of Ministry of Education, China Mobile (5-7) and State Key Laboratory of Software Development Environment (SKLSDE-2018ZX).

References

[1] F. Bourse, M. Lelarge, M. Vojnovic, Balanced graph edge partition, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2014, pp. 1456–1465.
[2] P.O. Boykin, V.P. Roychowdhury, Leveraging social networks to fight spam, Computer 38 (4) (2005) 61–68.
[3] C. Chevalier, F. Pellegrini, PT-scotch: a tool for efficient parallel graph ordering, Parallel Comput. 34 (6–8) (2008) 318–331.
[4] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[5] C.M. Fiduccia, R.M. Mattheyses, A linear-time heuristic for improving network partitions, in: 19th Design Automation Conference, IEEE, 1982, pp. 175–181.
[6] M.R. Garey, D.S. Johnson, L. Stockmeyer, Some simplified np-complete problems, in: Proceedings of the 6th annual ACM Symposium on Theory of Computing, ACM, 1974, pp. 47–63.
[7] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, PowerGraph: distributed graph-parallel computation on natural graphs, in: Presented as Part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12), 2012, pp. 17–30.
[8] W. Hasenplaugh, T. Kaler, T.B. Schardl, C.E. Leiserson, Ordering heuristics for parallel graph coloring, in: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2014, pp. 166–177.
[9] G. Karypis, V. Kumar, Analysis of multilevel graph partitioning, in: Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, IEEE, 1995. 29–29.
[10] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.
[11] G. Karypis, V. Kumar, Multilevelk-way partitioning scheme for irregular graphs, J. Parallel Distrib. Comput. 48 (1) (1998) 96–129.
[12] G. Karypis, V. Kumar, Parallel multilevel series k-way partitioning scheme for irregular graphs, SIAM Rev. 41 (2) (1999) 278–300.
[13] B.W. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell Syst. Tech. J. 49 (2) (1970) 291–307.
[14] D. LaSalle, G. Karypis, Multi-threaded graph partitioning, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IEEE, 2013, pp. 225–236.
[15] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed GraphLab: a framework for machine learning and data mining in the cloud, Proc. VLDB Endowment 5 (8) (2012) 716–727.
[16] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ACM, 2010, pp. 135–146.
[17] D. Margo, M. Seltzer, A scalable distributed graph partitioner, Proc. VLDB Endowment 8 (12) (2015) 1478–1489.
[18] J. Nishimura, J. Ugander, Restreaming graph partitioning: simple versatile algorithms for advanced balancing, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2013, pp. 1106–1114.
[19] V. Osipov, P. Sanders, n-Level graph partitioning, in: European Symposium on Algorithms, Springer, 2010, pp. 278–289.
[20] S. Pandit, D.H. Chau, S. Wang, C. Faloutsos, NetProbe: a fast and scalable system for fraud detection in online auction networks, in: Proceedings of the 16th International Conference on World Wide Web, ACM, 2007, pp. 201–210.
[21] F. Pellegrini, J. Roman, SCOTCH: a software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: International Conference on High-Performance Computing and Networking, Springer, 1996, pp. 493–498.
[22] F. Rahimian, A.H. Payberah, S. Girdzijauskas, M. Jelasity, S. Haridi, A distributed algorithm for large-scale graph partitioning, ACM Trans. Auton. Adapt. Syst.(TAAS) 10 (2) (2015) 12.
[23] S. Ressler, Social network analysis as an approach to combat terrorism: past, present, and future research, Homeland Security Affairs 2 (2) (2006) 1–10.
[24] I. Safro, P. Sanders, C. Schulz, Advanced coarsening schemes for graph partitioning, J. Exp. Algorithmics 19 (2015). 2–2.
[25] P. Sanders, C. Schulz, Engineering multilevel graph partitioning algorithms, in: European Symposium on Algorithms, Springer, 2011, pp. 469–480.
[26] I. Stanton, G. Kliot, Streaming graph partitioning for large distributed graphs, in: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2012, pp. 1222–1230.
[27] C. Tsourakakis, C. Gkantsidis, B. Radunovic, M. Vojnovic, FENNEL: streaming graph partitioning for massive scale graphs, in: Proceedings of the 7th ACM International Conference on Web Search and Data Mining, ACM, 2014, pp. 333–342.
[28] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[29] C. Walshaw, M. Cross, Mesh partitioning: a multilevel balancing and refinement algorithm, SIAM J. Sci. Comput. 22 (1) (2000) 63–80.
[30] L. Wang, Y. Xiao, B. Shao, H. Wang, How to partition a billion-node graph, in: 2014 IEEE 30th International Conference on Data Engineering, IEEE, 2014, pp. 568–579.
[31] R.S. Xin, J.E. Gonzalez, M.J. Franklin, I. Stoica, GraphX: a resilient distributed graph system on spark, in: First International Workshop on Graph Data Management Experiences and Systems, ACM, 2013, p. 2.