IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 23, 2020, accepted April 5, 2020, date of publication April 13, 2020, date of current version April 30, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2987355

Label Propagation-Based Parallel Graph Partitioning for Large-Scale Graph Data

MINHO BAE[1], MINJOONG JEONG[2], AND SANGYOON OH[1]
[1]Computer Engineering, Ajou University, Suwon, South Korea
[2]Supercomputing Department, Korea Institute of Science and Technology Information, Daejeon 34141, South Korea

Corresponding author: Sangyoon Oh (syoh@ajou.ac.kr)

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant NRF-2018R1D1A1B07043858, and in part by the Supercomputing Department, Korea Institute of Science and Technology Information (KISTI) under Grant K-19-L02-C07-S01.

ABSTRACT The increasing importance of graph data in various fields requires large-scale graph data to be processed efficiently. Furthermore, well-balanced graph partitioning is a vital component of parallel/distributed graph processing. The goal of graph partitioning is to obtain a well-balanced graph topology, where the size of each partition is balanced while the number of edge cuts is reduced. Moreover, a graph-partitioning algorithm should achieve high performance and scalability. In this study, we present a novel graph-partitioning algorithm that ensures a high edge cutting quality and excellent parallel processing performance. We apply formulas based on the label propagation algorithm to improve the quality of edge cuts and achieve fast convergence. In our approach, the necessity of applying the label propagation process for all vertices is removed, and the process is applied only for candidate vertices based on a score metric. Our proposed algorithm introduces a stabilization phase in which remote and highly connected vertices are relocated to prevent the algorithm from becoming trapped in local optima. Comparison results show that a prototype based on the proposed algorithm outperforms well-known parallel graph-partitioning frameworks in terms of speed and balance.

INDEX TERMS Data processing, graph data, parallel processing, partitioning algorithms.

I. INTRODUCTION

Recently, graph data have become increasingly important for applications in various fields, such as e-science, medical information systems, and social data management systems [1]. The structure of graph data is very effective for representing the relationships between data, a fact that is driving its rising significance. A popular example of large-scale graph data is the index of the World Wide Web, which has a size of approximately 50 billion [2]. Other examples include Facebook's social data, which comprise over 100 billion links [3], and Alibaba's graph data, which contain over one billion users and two billion items [4].

Although a graphical representation is effective for representing the relationship between data objects, the processing of large-scale graph data is challenging because of the intense resource consumption it requires [5]. Traditionally, graph algorithms are designed under the assumption that a single machine has sufficient system memory to store all the input graph data to be processed [6], [7]. However, physical limitations make it almost impossible for a single machine to be equipped with sufficient memory to process large-scale graph data. To address this issue, several parallel/distributed graph processing frameworks for handling large-scale graph data effectively have been proposed. Noteworthy frameworks include Pregel [8], PowerGraph [9], Trinity [10], Apache Giraph [11], GPS [12], GraphLab [5] and GraphX [13]. These parallel/distributed frameworks divide large-scale graphs into multiple segments, which are then processed in parallel on multiple computers. Each machine then applies the graph algorithm to the vertices corresponding to that specific machine. This approach is called vertex-centric graph processing.

Balanced graph partitioning is an efficient data decomposition operation that is essential for vertex-centric graph processing of large-scale graph data. A poor execution of the data decomposition not only results in an imbalance of the workload distribution over parallel machines, but also

The associate editor coordinating the review of this manuscript and approving it for publication was Daniel Grosu.

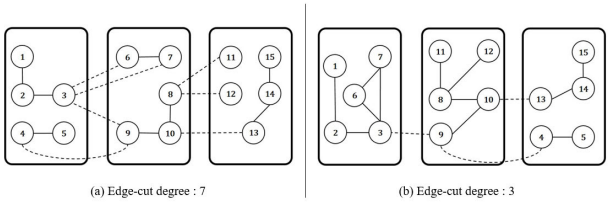


FIGURE 1. An example of graph partitioning.

increases the communication overhead. Thus, the quality of graph partitioning significantly affects the performance of graph processing, as well as the scalability of large-scale parallel/distributed graph processing.

The goal of graph partitioning, which can be considered a key preprocessing phase in any graph processing task, is to obtain a well-balanced graph topology. So that the graph partitioning is well balanced, the size of each partition must be balanced (for workloads) [14]–[16] and the edge cut degree must be minimized to allow communication between the partitions [17], [18]. Figure 1 shows an example of a comparison of (a) ''poor'' and (b) ''good'' edge cut graph partitioning results. Although the partitioning shown in both Figure 1(a) and (b) satisfies the balance of the vertex, (b) has fewer edge cuts than (a). Thus, the case in Figure 1(b) shows a better overall performance for most graph processing algorithms, because it requires less network communication. The graph-partitioning algorithm is used in a preprocessing phase to find the graph topology that can be processed efficiently in a real distributed environment; the results are as shown in Figure 1(b). Because real-world graph data show a skewed power-law distribution, we also claim that the graph-partitioning algorithm should consider not only vertex balance but also edge balance.

The balanced graph-partitioning problem is known to be NP-complete [19]. For this reason, most approaches are designed to solve the balanced graph-partitioning problem by using heuristic methods, similarly to various other NP-complete problems. Many previous approaches for graph partitioning are based on a local search algorithm, such as the Kernighan–Lin (KL) [20] and Fiduccia–Mattheyses (FM) algorithms [21]. These algorithms require considerable computation to obtain the optimal edge cut as the number of vertices, i.e., nodes, and partitions increases. Hence, because of the huge computational cost, these approaches are applied only to small graphs [22].

To reduce the computational cost, some studies applied multilevel methods [23]–[32] to reduce the computation time by using the coarsening process, which splits the original graph into smaller graphs by contracting the connectivity information of vertices. The multilevel approach, when combined with a heuristic approach such as KL, improves performance and yields significantly better edge cuts than a standalone heuristic approach. However, some graphs are not effectively coarsened during the coarsening phase. The overhead of the coarsening process when applied to these graphs can be so large that the advantages of the multilevel approach are effectively nullified. In addition, the multilevel approach is very memory-intensive, which is a disadvantage in terms of scalability.

TABLE 1. Characteristics of previous approaches.

Approaches	Edge-cut	Scalability	Performance
KL / FM	X	X	X
Label-Propagation	△	O	O
Multi-Level	O	△	O

The label propagation (LP) algorithm [33] is a different approach used to solve the graph clustering problem. This algorithm typically runs in near-linear time over a single iteration, but its iterative operations can be disadvantageous. Although LP can reduce the volume of computation as compared to KL and FM, it still requires many iterative operations [34]. In addition, the resulting edge cut performance of this approach has been shown to be poorer than that of the multilevel approach. We summarize the edge cut, scalability, and performance characteristics of the aforementioned approaches in Table 1.

In this paper, we propose a novel parallel graph-partitioning algorithm that provides a low edge cut degree and high performance processing capability for large-scale graph data. In our proposed method, we consider scalability the most important characteristic for achieving large-scale graph data processing. Thus, the design of our algorithm is based on the LP algorithm and is aimed to solve the problems of the original LP algorithm, such as the edge cut quality.

To improve edge cut quality and achieve fast convergence, we use an LP algorithm with a score metric based on the basic formula of the KL algorithm. In our approach, rather than selecting random vertices from the partition and applying the LP process for all vertices, we apply the LP process only for candidate vertices with lower scores. We also use a process that reduces the number of candidate vertices in every iteration utilizing a defined ratio to improve the LP performance. We named this process ''quick-converging label propagation'' (QCLP). To ensure that QCLP does not cause graph topological convergence to some local optimum, we introduce a stabilization phase. In the QCLP phase, we change the graph topology based on the vertices in each partition that are not required, whereas in the stabilization phase, the graph topology is changed based on the vertices that are important in each partition. In other words, our algorithm is processed in two directions.

Our algorithm also considers the network communication problem that naturally occurs in distributed parallel processing. Distributed machines need to share the new position, i.e., partition, or vertex score for the next iteration in LP. In this process, a correlation exists between the overhead of the data update frequency and the accuracy of the vertex position. More frequent updating of the data results in the

vertex position being more accurate, which in turn increases the quality of the edge cut partitioning. However, an increase in the data update frequency also incurs a higher communication overhead. Thus, considering both the network communication overhead and vertex position accuracy, we apply the bulk synchronous parallel (BSP) style [35] lazy updating scheme to reduce the amount of network communication and share changes only in the location information and the vertex scores. In addition, when we execute the LP process, we consider the possibility of vertex relocation to increase vertex position accuracy. The details of our approach are presented in Sections III and IV.

The primary contributions of this paper are as follows.

1) A novel two-way graph-partitioning algorithm based on LP with a score metric to prevent the algorithm becoming trapped around local optima is proposed.
2) Improvement in the edge cut results as a result of considering the possibility of vertices being relocated is shown.

The remainder of this paper is organized as follows. We review and analyze previous balanced graph-partitioning approaches in Section II. In Section III, we describe the details of our proposed graph-partitioning algorithm. We describe our proposed data structure in Section IV. In Section V, we present and evaluate the results of experiments in which our proposed method was applied and present a performance comparison of our method with other major approaches. Section VI presents concluding remarks.

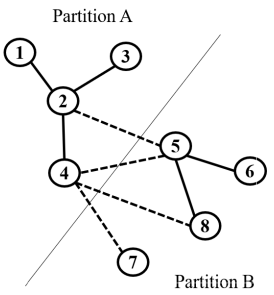
II. RELATED WORK

As described in Section I, several researchers attempted to achieve balanced graph partitioning by using heuristic and multilevel approaches. Heuristic graph-partitioning approaches can be categorized as score- or LP-based. In the following, we analyze these two types of approaches and then outline the multilevel algorithm.

A. SCORE-BASED APPROACHES

Most heuristic approaches are designed to solve the balanced graph-partitioning problem by applying a local search, and many local search algorithms are based on the KL algorithm [3]. The primary concept of the KL algorithm is that, by repeatedly swapping vertices between the partitions in the direction of increasing the local score, the minimum edge cut can be obtained. Figure 2 shows an example of the KL algorithm. The graph in the upper left of Figure 2 shows the initial graph, which is randomly divided into two partitions (A/B).

The algorithm proceeds in the following order. First, the values of "external cost (E)," "internal cost (I)," and the "difference between external and internal costs (D)" are calculated based on each partition, which indicates the number of remotely connected vertices, the number of locally connected vertices, and the difference between the external edge and internal edge costs, respectively. Subsequently, "G"



Partition A	E_a	I_a	D
1	0	1	-1
2	1	3	-2
3	0	1	-1
4	3	1	2

Partition B	E_a	I_a	D
5	2	2	0
6	0	1	-1
7	1	0	1
8	1	1	0

Pair	C(a,b)	G	Pair	C(a,b)	G
G_{15}	0	-1	G_{35}	0	-1
G_{16}	0	-2	G_{36}	0	-2
G_{17}	0	0	G_{37}	0	0
G_{18}	0	-1	G_{38}	0	-1
G_{25}	1	-4	G_{45}	1	0
G_{26}	0	-3	G_{46}	0	1
G_{27}	0	-1	G_{47}	1	1
G_{28}	0	-2	G_{48}	0	0

FIGURE 2. An example of the Kernighan-Lin Algorithm.

is calculated using the values of "D" and "C(a,b)," which indicate the vertex connectivity of A and B (1 if connected; otherwise 0). Finally, the algorithm swaps the combination of vertices that obtains the largest value of G. In the example given in Figure 2, the highest value of G results from swapping Vertices 4 and 7. This swapping process is repeated iteratively, neglecting previously processed vertices, until no positive gain scores are available, i.e., all the values of $G \leqq 0$. KL provides a balanced partition by swapping vertices, but computation time increases exponentially with the number of vertices. Additionally, it is not guaranteed that the final edge cut is optimal when $G \leqq 0$.

Rather than swapping vertices, the FM [36] algorithm moves vertices to improve the performance of the KL algorithm algorithm [3]. The FM algorithm also differs from KL in that it uses a data structure called a bucket queue. Using this data structure, the FM algorithm can obtain the vertex with the highest gain score, move that vertex, and update the score of its neighbor vertices in constant time [37]. However, the calculation of the relevant scores when the FM algorithm is used is computationally expensive.

B. LABEL PROPAGATION-BASED ALGORITHM

The LP algorithm [33] is an additional basis applied in graph-partitioning algorithms to achieve balanced graph-partitioning. Figure 3 shows an example of the LP algorithm. The LP algorithm is simple and lightweight and runs each iteration in near-linear time. In the LP algorithm, only the specified vertex and its neighbors are processed. Thus, it takes only O(E) time. The LP algorithm is therefore advantageous in terms of performance and scalability for graph partitioning [38].

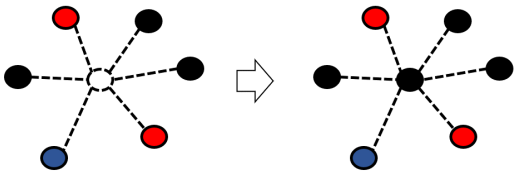


FIGURE 3. An example of the Label Propagation Algorithm.

Some issues still remain when the original LP algorithm is used for balanced graph partitioning [39]. LP-based algorithms require many iterations to achieve convergence, because the algorithm is executed repeatedly until a specific threshold iteration is reached. Although an increase in the number of iterations increases the probability of obtaining better edge cut quality, this process requires an excessive amount of computation time for graph partitioning. In addition, it must be executed for all vertices in every iteration, and it is difficult to predict the optimal threshold for iteration.

The LP algorithm also cannot guarantee a balanced graph topology, because it is a clustering rather than a partitioning approach. Thus, it is likely that the original LP algorithm will return a graph topology with partitions of various sizes rather than with balanced partitioning [39].

Finally, when using a distributed LP algorithm, an increase in the frequency of information updates comes at the cost of an increase in network congestion. Network overhead is an inevitable cost of communicating updates when vertex positions change while the LP process is being executed in distributed machines. Although edge cut results could be improved by executing the update process per vertex, performance is severely negatively affected if a network barrier is encountered for every calculation. Conversely, executing computation for all vertices independently while updating information per iteration rather than per vertex enhances performance; however, the quality of the resulting edge cut may be diminished as a result of slower updating. To improve performance, previous LP-based graph-partitioning approaches [34], [40] used the latter approach. Accordingly, the edge cut results were not optimal as compared to the results of multilevel approaches that use contracted connectivity information. Figure 4 shows an example of the graph-partitioning algorithm based on LP.

C. MULTILEVEL APPROACH

The key idea of the multilevel approach is to reduce the computational complexity and enhance the performance of a graph by reducing its size (i.e., making it coarser). Figure 5 illustrates the multilevel approach. Although this approach may appear to be independent, it uses heuristic approaches in the uncoarsening phase.

The multilevel approach consists of three phases: coarsening, initial partitioning, and uncoarsening. First, the coarsening phase processes the original graph into smaller graphs

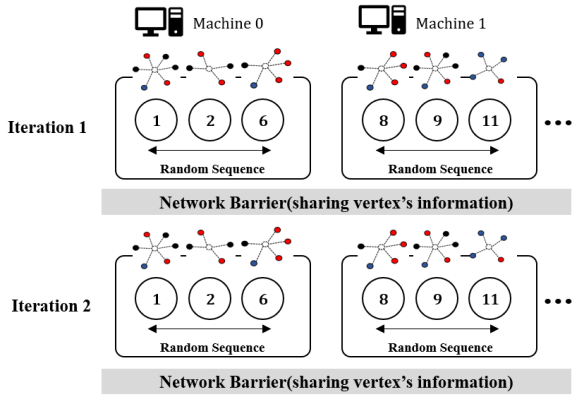


FIGURE 4. An example of the graph partitioning algorithm based on label propagation algorithm.

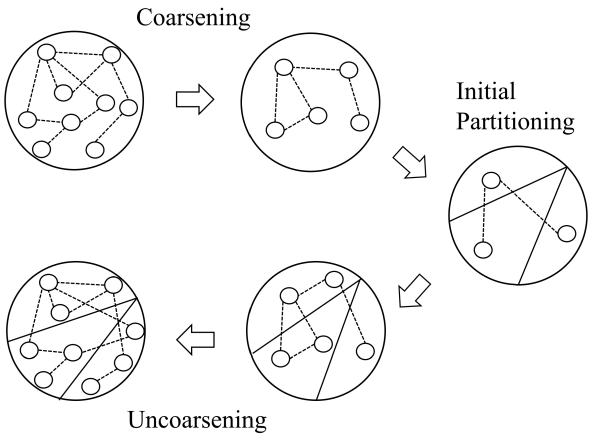


FIGURE 5. Illustrate of the multilevel approach.

by contracting the connectivity information. In this process, algorithms such as random matching (RM) and heavy edge matching (HEM) can be used. RM randomly selects a vertex from the entire graph, whereas HEM selects the vertices with the highest connectivity among the visited vertices.

Next, the initial partitioning phase divides the graph resulting from the coarsening phase into k partitions. In this phase, algorithms such as random clustering and breadth-first search are used. Finally, the uncoarsening phase obtains the minimum edge cut while restoring the original graph. In this phase, the process of moving or swapping vertices is repeated by applying a local search approach, such as the KL or FM algorithm. Many well-known software packages employ the multilevel approach, including Metis [24], Chaco [27], Scotch [28], ParMetis [29] and Pt-Scotch [30]. As mentioned in Section I, the multilevel approach shows considerable improvements in performance and edge cut as compared to standalone heuristic algorithms. However, the requirement for large amounts of system memory presents a severe drawback in terms of scalability.

III. PROPOSED GRAPH-PARTITIONING ALGORITHM

We propose a novel parallel graph-partitioning algorithm that utilizes the LP algorithm and a stabilization process with a score metric. In the proposed algorithm, we use the score of vertex position appropriateness as a critical factor, because it is used to identify the vertices to be relocated. By calculating only the vertices that should be relocated, the edge cut quality and performance can be improved.

A. DEFINITION OF THE SCORE

First, we define the concepts and metrics used in the score of our proposed approach. As a KL approach, our method distinguishes between local edges and remote edges in corresponding lists of vertex edge information. Next, the vertex partition score (VP Score) is defined to indicate the appropriateness of a vertex in its current partition. Formula (1) is used to calculate the score of V_i based on the partition to which it belongs.

$$\text{VP Score}(V_i) = V_{il} - V_{ir}, \tag{1}$$

where :

V_{il} : is the number of connected edges to the same partition as V_i (local edge)

V_{ir} : is the number of connected edges to other partitions as V_i (remote edge)

In this expression, the number of edges connected to other partitions (remote edges) belonging to V_i is subtracted from the number of edges connected to the same partitions (local edges). This formula is almost identical to the ''difference between external and internal'' of the KL algorithm [3] and has been verified experimentally. However, the KL algorithm uses these scores to obtain the gained score when this vertex is swapped with all vertexes in other partitions, which requires a considerable amount of computation. In our scheme, this formula is used to reduce the LP process, which is applied for all nodes in the previous graph-partitioning approach based on LP. This formula expresses the appropriateness of a vertex in the partition as the number of local edges in excess of the remote edges; that is, the larger the score, the more appropriate the vertex is to the current partition. Using this characteristic, we advance to the LP process only vertices with a lower VP score to benefit from the computation and communication volume. This approach has the same effect as the coarsening process in the multilevel approach; however, we do not store additional information data, because there is no uncoarsening process. Thus, we can obtain an advantage in terms of memory consumption.

The following formula gives the Total Score (T Score), which is the sum of the VP Score for all vertices in the entire graph.

$$\text{T Score} = \sum \text{VP Score}(^{\forall}V) \tag{2}$$

The edge cut indicates the sum of the remote edges for all vertices in the entire graph, and thus, the edge cut of the entire graph decreases as the T Score increases, and vice versa. In other words, it indicates the appropriateness of the current graph topology status.

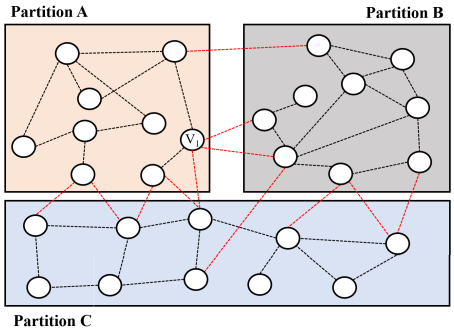


FIGURE 6. An example of graph topology.

Figure 6 shows an example of a graph partition in which the number of partitions is set to three. The red lines indicate remote edges connecting two vertices in different partitions, and the black lines indicate local edges between two vertices in the same partition. In this example, the VP Score of V_1 is 2 (i.e., 4 minus 2) and the T Score is 22 (i.e., 34 minus 12).

B. PROPOSED GRAPH-PARTITIONING ALGORITHM

In this section, our proposed LP algorithm is described in detail and the stabilization process is presented. In the proposed algorithm, we apply the novel QCLP algorithm. Then, the result of the QCLP algorithm from the given partitioning stage yields the total number of remote edges reduced by the next iteration, which increases the T Score. Our QCLP algorithm is aimed to improve edge cuts and accelerate convergence. It is known that the LP algorithm changes the partition (label) of a vertex according to the maximum number of partitions within its neighbor vertices; its performance is good in terms of graph clustering. However, the conventional approach applies LP for all vertices without considering their current positions, and therefore, all vertices are recalculated regardless of whether they are located in the most suitable partition. Thus, the algorithm may suffer from serious performance degradation in terms of accuracy and processing.

To address this issue, we apply the LP process only for vertices with a lower VP Score, instead of for all vertices. Furthermore, we gradually reduce the number of candidate vertices in every iteration of the LP process. This process is discussed in more detail in Section IV.A. In this phase, we refer to the candidate vertices with lower scores as the local lower score vertices (LLSV). The definition of LLSV is as follows.

Definition 1:

$$\text{LLSV}(P_i) = \{\text{list of vertices with low score in partition } P_i, \; P_i \in P\}$$

In summary, the QCLP phase identifies and relocates the LLSV using LP. Figure 7 shows an overview of the QCLP

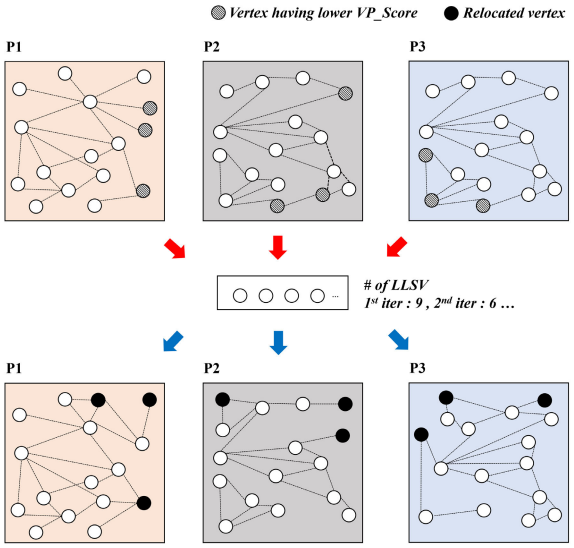


FIGURE 7. Overview of Quick Convergence Label Propagation.

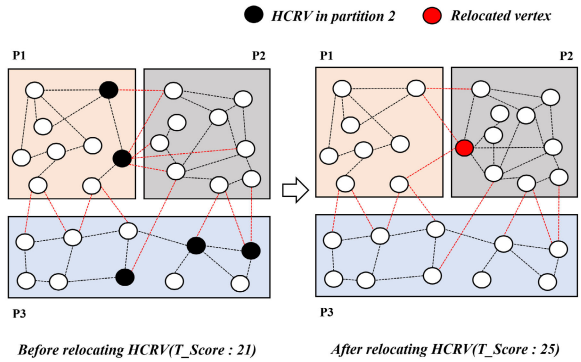


FIGURE 8. Overview of Stabilization process.

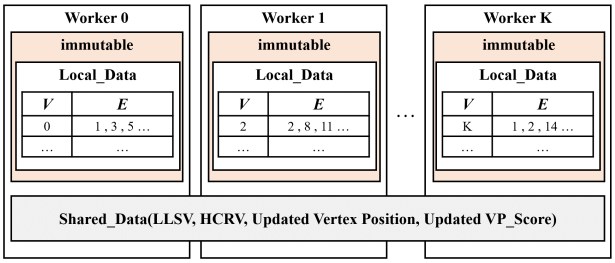


FIGURE 9. Overview of Data Structure.

algorithm. In this figure, the QCLP phase identifies the LLSV and relocates these vertices to another partition that can obtain the highest score. For every iteration, the number of LLSV is gradually decreased.

We also propose a stabilization process to prevent the algorithm becoming trapped in the local optima. If we use only the LP algorithm for graph partitioning, we may overlook vertices that would otherwise have gained a score improvement by being relocated to other partitions. That is, T Score can be increased by relocating certain vertices with a high VP Score to other partitions; some vertices may have been trapped in the partition because the VP Score is a local rather than a global score. As the proposed QCLP algorithm processes vertices based only on the VP Score, it is difficult to process vertices that are missed. To resolve this problem, we add a stabilization process to our graph-partitioning algorithm. This stabilization process changes the graph topology based on the most-required vertices in each partition rather than on their lower VP Score. In this phase, we refer to candidate vertices connected to remote partitions as higher connectivity to remote vertices (HCRV). The HCRV are defined as follows.

Definition 2:

$$\text{HCRV}(P_i) = \{\text{list of vertices that have many remote connections in partition } \text{P}_i, \text{P}_i \in \text{P}\}$$

In summary, the stabilization phase identifies and relocates the HCRV. Figure 8 shows an overview of the stabilization process. In this figure, "Before relocating HCRV" shows the identification of the HCRV (black vertices) in partition 2 (P2 in Figure 8) based on the remotely connected edges. "After relocating HCRV" shows the process of relocating vertices (red) that will increase the T Score when migrated to partition 2. We describe the details of our approach in Section IV.

IV. PARALLEL GRAPH-PARTITIONING PROCESS

For use in parallel processing, our proposed algorithm requires an appropriate data structure. We describe our graph data distribution scheme with the data structure for multiple workers (machines) in Section IV.A. In Section IV.B, we describe the parallel processing mechanism of our graph-partitioning algorithm.

A. DATA STRUCTURE

The proposed algorithm requires two types of data: locally stored data (Local Data) that each worker processes, and shared data (Shared Data) that all workers process.

First, we distribute the graph data randomly and evenly among the workers using a random hash function. These distributed graph data are stored in each worker as the Local Data (i.e., edge information). Each worker is responsible for updating its own vertices, and the Local Data stored in each worker remains unchanged over the entire partitioning process. In other words, each worker conducts an LP process, updating VP Score and identifying the LLSV, as well as the HCRV, based on the local edge information of vertices (Local Data). Next, the values that should be shared to all workers through BSP communication on each iteration (e.g., LLSV, HCRV, updated vertex position, and VP Score) are stored in Shared Data. Figure 9 illustrates the data structure of the proposed algorithm. Additionally, the initial partition position of each vertex is determined using a random hash function, which is similar to the initial graph distribution.

B. PARALLEL GRAPH-PARTITIONING ALGORITHM

In this section, we describe the parallel processing of our graph-partitioning algorithm consisting of QCLP, as well as the stabilization phase. We designed our proposed graph-partitioning algorithm using three principles to maintain a user-specified imbalance ratio, e.g., vertex: 1.03 and edge: 1.30, because constraining both the vertex and edge size results in partitioning that is too complex, i.e., has too many constraints, to allow decisions in the LP process. For example, the LP process used to increase edge quality is tightly coupled to the balancing process. Thus, we designed the algorithm to focus on different factors in each step to decouple the balancing process and the LP process. We also describe the principles for achieving balanced partitions in each step.

1) QUICK-CONVERGING LABEL Propagation(QCLP)

Our QCLP algorithm is designed to reduce computation and to increase the quality of the edge cut by implementing the LP process only for limited vertices based on the VP Score.

For efficient implementation of the LP, we consider two overhead issues and provide a solution for each. The recalculation of the VP Score of all vertices incurs too large a computational overhead and is therefore ineffective. When $P[V_{LLSV}]$, i.e., the partition position of LLSV, is changed during the QCLP process, only the VP Score of V_{LLSV} and V_{LLSV}'s neighbor vertices, i.e., the edges connected to LLSV, are changed. Thus, we recalculate the VP Score only for V_{LLSV} and their neighbor vertices, rather than for all vertices. Additionally, we designed our algorithm such that the calculation of the VP Score and the determination of new $P[V_{LLSV}]$ are overlapped in the LP process to reduce computational overhead.

We also considered the relationship between the accuracy of the LP result and the frequency of network communication. If network requests are not sent to other workers, no means exists of informing a worker whether the $P[V_{LLSV}]$ computed by other workers has changed. Thus, each worker conducts the LP process using outdated vertex information from the previous LP step. Conversely, an increase in network communication provides more recent vertex information, which enables a better LP result to be achieved.

To resolve this issue, we propose the lazy-update BSP communication paradigm. In our lazy-update BSP scheme, we designed a network communication scheme that is more effective, because it is iteration- rather than vertex-based. However, to compensate for lazy updates, the vertex information is updated only when vertices on the same machine are changed. We also consider the possibility of vertex relocation. We ignore the V_{LLSV}'s neighbor vertices belonging to the LLSV during the LP processing of $P[V_{LLSV}]$, because it is very likely that these vertices will move to another partition. If we include such vertices when we perform the LP process, the new $P[V_{LLSV}]$ will be inaccurate.

The QCLP algorithm is illustrated in Algorithm 1. It consists of three steps: 1) LLSV identification, 2) LP and updating of the VP Score, and 3) reupdating of the VP Score in LLSV.

Algorithm 1 Quick Converging Label Propagation

Description : The QCLP algorithm is performed for at least α iterations and it is finalized when the T Score has not increased by more than k% for β iterations. (α, β, and k are user-defined values.)

LLSV : array of LLSV
V[] : array of vertex's edge information
P[] : array of vertex's partition position
VP[] : array of vertex's VP Score
U : updated data, L : Local Data in each machine
P : previous, N : neighbor vertex

```
1:  Iter ← 0, T_Score ← 0
2:  while Iter < α || count < β do
3:      N ← 100 - EXP(Iter / 10)
4:      for i = 1 ... p do
5:          LLSV_L ← SortingandExtract(V_L[],N%)
6:      LLSV ← AllGather(LLSV_L)
7:      for all V ∈ LLSV_L do
8:          VP[V], P_U[V], VP_U[V_N] ← LPandUpdate(V)
9:      P[V] ← AllGather(P_U[])
10:     VP[] ← ISendRecv(VP_U[])
11:     for all V ∈ LLSV_L do
12:         VP[V] ← ReUpdate(V)
13:     Update T_Score_L
14:     T_Score ← AllReduce(T_Score_L)
15:     if (T_Score / T_Score_P < k) then
16:         Count ← Count+1
17:     Iter ← Iter +1
```

All three steps are processed independently for each machine based on its Local Data. As mentioned above, we gradually and iteratively reduce the number of candidate vertices. To achieve this, we use the natural exponential function as the criterion to determine the extent to which the VP Score should be reduced, i.e., N%. The processes achieve the same effect as the coarsening process in the multilevel approach, but without storing contracting connectivity information data, and thereby provide an advantage in terms of memory usage. The algorithm also provides a better performance in terms of edge cut quality than the original LP algorithm, because the QCLP calculates the location of vertices by their need to be relocated. In Algorithm 1, this process is pseudo-coded in lines 3–5. Next, we share the local LLSV to all machines (line 6) using collective network communication, i.e., the BSP scheme.

Next, we conduct the LP process to determine a new $P[V_{LLSV}]$ and update the VP Score. We update the VP Score only for V_{LLSV} and neighbors of V_{LLSV}. We also enable the VP Score updating and LP processes to be overlapped to achieve a higher performance through the modification of the original LP algorithm.

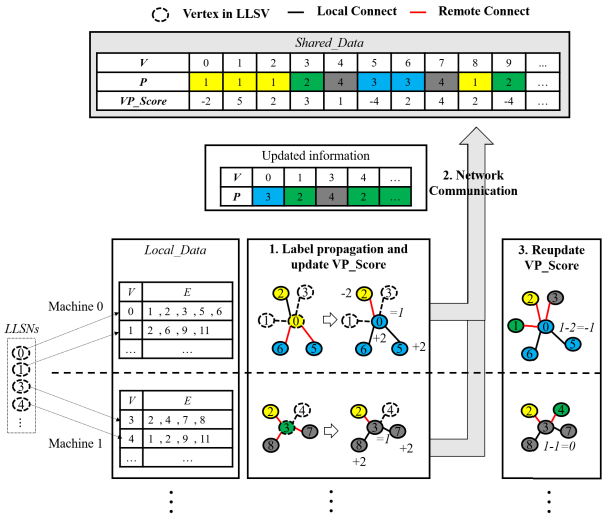


FIGURE 10. Illustrate of the LP and updating VP_Score.

The principle of the QCLP process is to prioritize the vertex balance. In the QCLP phase, the more vertices a partition has, the greater the number of additional external vertices that are likely to be relocated to the partition in the LP process. This causes most vertices to be gathered in a particular partition. To avoid this issue, we relocate vertices in an order that provides each partition with almost the same number of vertices. In addition, to prevent the number of edges in a particular partition from becoming too large, the criterion for identifying the LLSV is changed to the number of edges rather than the VP Score when the number of edges in a particular partition exceeds a specific ratio (we set this ratio twice as much as the user-configured edge imbalance ratio, empirically).

Then, BSP communication is used to share only the partition positions of the changed vertex (line 9) and the VP Score of only the changed vertex's neighbor (line 10). Finally, we update the VP Score for V_{LLSV} a second time using the updated partition position (lines 10 and 11).

Figure 10 illustrates an example of lines 7–12 of Algorithm 1. In this example, Vertices 1, 2, 3, 5, and 6 are connected to Vertex 0 of Machine zero. As mentioned above, to improve accuracy, we consider only the nodes that are not included in the LLSV (dotted circle). Thus, we consider the partition position only for Vertices 2, 5, and 6 (solid circles). To determine the new $P[V_{LLSV}]$, we obtain the previous P[V] with the Shared Data updated in every iteration. For Machine zero, $P[V_5$ and $V_6]$ is 3 (blue background) and $P[V_2]$ is 1 (yellow background). Therefore, the highest T Score is obtained when $P[V_0]$ is moved to 3. Meanwhile, we update the VP Score of V_{LLSV} and V_{LLSV}'s neighbor vertices. Then, VP Score$[V_2]$ is decreased by two, because Vertex 0 ($P[V_0] : P_1 \rightarrow P_3$) is no longer considered local (i.e., in the same partition) and VP Score$[V_5$ and $V_6]$ is increased by two, because V_0 becomes local. Finally, we calculate VP Score$[V_0]$.

After V_0 is relocated, there are two locally connected vertices (V_5 and V_6), one remotely connected node (V_2), and two unknown connected nodes (V_1 and V_3). Because we do not know the information of $P[V_1$ and $V_3]$, which may be changed, we calculate only the locally connected and remotely connected vertices. Thus, VP Score$[V_0]$ is 1 (locally connected – remotely connected). Then, we share the updated information through network communication.

Next, as we know the information of V_1 and V_3 (unknown connected vertices in the previous phase) from the updated information, we can reupdate VP Score$[V_0]$. If $P[V_1$ and $V_3]$ are the same as $P[V_0]$, we either increase VP Score$[V_0]$ by 1 or decrease VP Score$[V_0]$ by 1. In this example, because both V_1 and V_3 have different partitions from V_0, we decrease the score of V_0 by 2. Thus, the final VP Score$[V_0]$ is negative (-1), where VP Score$[V_0]$ is 1 before reupdating.

2) STABILIZATION PROCESS

In this phase, we conduct graph stabilization to resolve the local optima problem. We stabilize the graph partitions by relocating the HCRV and relocate V_{HCRV} to increase the T Score. The stabilization algorithm of the HCRV is described in Algorithm 2. Because QCLP and HCRV are similar but use different flows, we describe only those parts that differ from the QCLP phase.

Algorithm 2 Stabilization Process

Description : The stabilization process is performed for α iterations. (α is a user-defined value.)

HCRV : array of HCRV
V[] : array of vertex's edge information
P[] : array of vertex's partition position
VP[] : array of vertex's VP Score
U : updated data, L : Local Data in each machine
C : candidate vertex, N : neighbor vertex

```
1:  Iter ← 0, T_Score ← 0
2:  while Iter < α do
3:    for all V ∈ LLSV_L do
4:      if (VP_Score[V] < 0) then
5:        HCRV_C ← LP(V)
6:    for all V ∈ HCRV_C do
7:      VP[V], P_U[V], VP_U[V_N] ← DecideandUpdate(V)
8:    P[V] ← AllGather(P_U[V])
9:    VP[V] ← ISendRecv(VP_U[])
10:   Update T Score_L
11:   T Score ← AllReduce(T Score_L)
12:   Iter ← Iter +1
```

To extract the candidate vertices of HCRV, each node conducts LP on the local vertices where VP Score is less than 0 (lines 3–5 in Algorithm 2). If VP Score is greater than 0, more than half of the neighbor vertices already exist in the current partition, and there is no partition that is more suitable than the current partition.

After the LP process, we select vertices that satisfy the size-constrained conditions of partitions among the HCRV

candidates. For the stabilization process, we select the vertices for which the relocated partition does not exceed the edge/vertex imbalance ratio. Then, we decide on the final HCRV in descending order of the number of edges among the vertices to ensure that the edge balance is not excessive.

Next, we update the partition position and VP Score of those vertices (lines 6 and 7). Finally, as in the QCLP process, the updated values are shared via network communication (lines 8 and 9).

3) EDGE-BALANCED PARTITIONING PROCESS

During QCLP and the stabilization process, we focus on vertex balance more than on edge balance. We advance these processes while maintaining the near-perfect vertex balance, but, in the case of edges, only to ensure that the edge balance is not excessive. After QCLP and the stabilization process, to maintain the near-perfect edge balance, we prioritize edge balance without considering edge cut quality as the principle. Therefore, it could serve to degrade the overall edge cut quality.

In this process, a partition with many edges identifies as many vertices as possible while satisfying the imbalance ratio in descending order of edge quantity. Then, identified vertices are relocated randomly to another partition that does not exceed the edge/vertex imbalance ratio. We repeat this process until the number of edges is balanced.

V. EXPERIMENT

In this section, we describe our experiment and analyze the results by comparing the execution time, edge cuts, and balance ratio of our proposed method against those of conventional graph-partitioning frameworks such as Metis [24] (version 5.1.0), ParMetis [29] (version 4.0.3) and XtraPuLP [34] (version 0.2).

Metis is a widely used graph-partitioning algorithm based on the multilevel approach. Although it provides only a single machine execution, it yields a good edge cut result and a balanced ratio. Therefore, we compare the edge cut result and the balance ratio of our proposed method with those of Metis.

ParMetis (a parallel version of Metis) and XtraPuLP are parallel graph-partitioning algorithms. ParMetis divides the graph over the number of machines used, and the subsequent partitioning is accomplished in a manner similar to that of Metis. XtraPuLP is a parallel version of PuLP [41] that uses the LP algorithm instead of the multilevel approach for graph partitioning. We compared these two algorithms with the proposed approach in terms of performance, edge cut results, and balance ratio.

A. EXPERIMENTAL ENVIRONMENT

We used two different Amazon Web Service (AWS) [42] clusters for the experiment. In the small cluster, we focused on demonstrating a scalability and performance comparison with a limited amount of memory. The large cluster was configured to have a large amount of memory for the large graph data experiment. In addition, to show the graph partition results when the number of machines is increased, we configured the large cluster with a larger number of machines than the small cluster. The configurations are summarized in Table 2.

TABLE 2. Specifications of clusters used in the experiment.

	Small-Cluster	Large-Cluster
Instance	c5.2xlarge	r4.8xlarge
CPU	3.90 GHz	2.30 GHz
Network Bandwidth	maximum 10Gb/s	10Gb/s
Memory	16GB	244GB
# of workers(cores)	4(4*4= 16)	16(16*16=256)

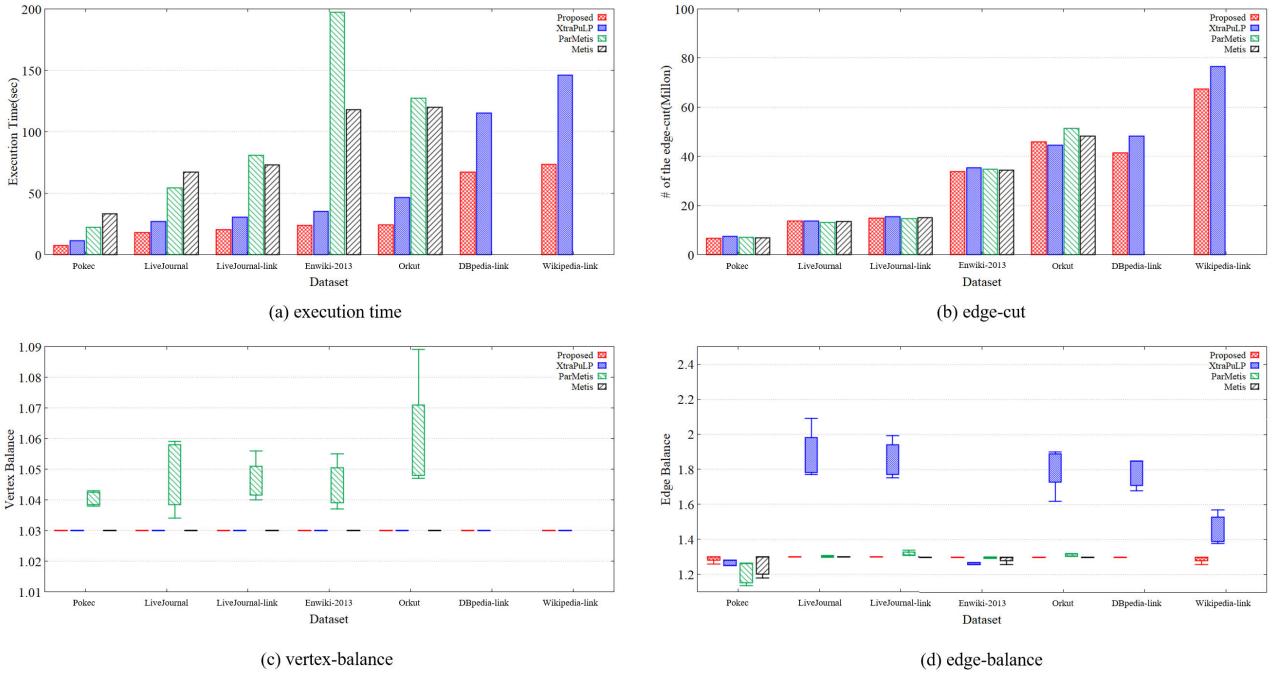
TABLE 3. Graph Information used in the experiment.

Name	# of Vertices	# of Edges	Type
Pokec	1M	30M	Social
LiveJournal	4M	68M	Social
LiveJournal-link	5M	49M	Social
Enwiki-2013	4M	101M	Hyperlink
Orkut	3M	117M	Social
DBpedia-link	18M	172M	Hyperlink
Wikipedia-link	12M	378M	Hyperlink
Twitter	41M	1.4B	Social

We used graph data from the Stanford Network Analysis Platform (SNAP) [43] and the Koblenz Network Collection (KONECT) [44]. Table 3 lists information about the graph data used in the experiment. All seven graphs listed in the table, except "Twitter," were tested on the small cluster, and the three large graphs in our datasets, i.e., Twitter, Wikipedia-link, and DBpedia-link, were tested on the large cluster.

In the experiment, we measured the performance, edge cut results, and balance ratio of the proposed algorithms and compared them with those of ParMetis, Metis, and XtraPuLP using the dataset in Table 2. For an experimental parameter value, the vertex-imbalance ratio and edge-imbalance ratio were set at 1.03 (3%) and 1.30 (30%), respectively.

For the QCLP process of our proposed method, we need to reduce the number of candidate vertices gradually (the LP process is applied only to candidate vertices) in every iteration to reduce the amount of computation. In this process, the Top-N% vertices with the lowest score are selected as candidate vertices. We used the natural exponent function to determine N%. N% was obtained by "100 - value of natural exponent function having input value of 1/p of the current iteration count"; i.e., when $p = 10$, the value of n in the first iteration is "100 - natural exponent function (1/10) = 98.89". In the experiments, we selected $p = 10$, empirically. Additionally, the number of iterations in the QCLP phase was configured to be at least 40. After 40 iterations, the algorithm continued until the T Score did not increase by more than 0.01% for 10 iterations. These settings gave us the best performance in terms of edge cut and execution time. We implemented our algorithm using MPI and OpenMP. For efficient memory usage, we perform the label propagation,



(a) execution time (b) edge-cut

(c) vertex-balance (d) edge-balance

FIGURE 11. Result of performance, edge-cut, vertex-balance and edge-balance on small-cluster.

sorting and extracting vertex process through OpenMP within multi-core parallel processing. In addition, we used MPI for network communication among distributed nodes.

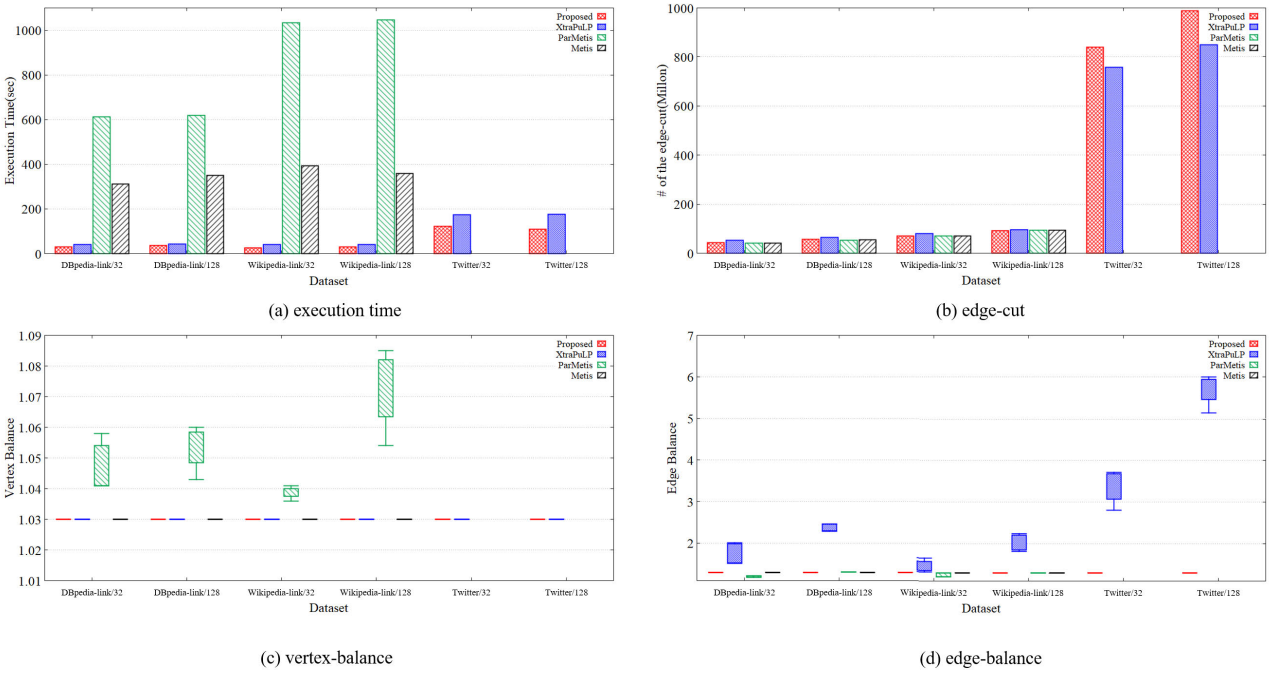
B. COMPARISON WITH OTHER SYSTEMS ON SMALL CLUSTER

We conducted the partitioning experiment using seven graphs for measuring the performance, edge cut results, and balance ratio on the small cluster. The number of partitions was configured as 32. The experimental results on the small cluster are shown in Figure 11.

First, as shown in Figure 11(a), we compared and analyzed the performance results in terms of execution time. Our proposed method is the fastest for every set of graph data. ParMetis, which is a distributed version of Metis, shows a poorer performance than Metis in some cases. We speculate that this is a result of the dataset exhibiting a skewed power-law distribution and that the data may not efficiently be reduced in the coarsening process of ParMetis. ParMetis is also significantly affected by the initial graph distribution and may require considerable communication with other machines depending on the quality of the initial partitioning. A comparison of the performance with that of the LP-based XtraPuLP shows that its convergence time is greater than that of the proposed algorithm, because all of the vertices undergo the LP process. Furthermore, for larger datasets, the proposed algorithm performed considerably better in terms of execution time than the other algorithms. For example, in the ''Orkut'' graph, the proposed algorithm was 4.98 and 1.93 times faster than Metis and XtraPuLP, respectively.

Using the same performance results, we could also analyze the scalability of the different algorithms. Only the proposed algorithm and XtraPuLP were able to handle the ''DBpedia-link'' and ''Wikipedia-link'' graph data in the small cluster environment. As mentioned above, graph algorithms based on the multi-level approach require a large memory footprint. For example, they required almost 20 GB memory when partitioning the ''Wikipedia-link'' graph data (12 M vertices and 378 M edges). Thus, graph algorithms based on the multi-level approach are not able to process large-scale graph data in a small-cluster environment. In addition, although not shown in Figure 11, only the proposed algorithm is able to process the ''Twitter'' data, which contain one billion edges, in the small-cluster environment. Thus, the proposed algorithm shows the best scalability.

Second, we analyzed the results in terms of edge cut performance. The proposed algorithm produced an edge cut performance almost identical to that of Metis, the edge cut quality of which has been shown to be best. Except for in the ''LiveJournal'' graph, our algorithm shows a slightly better edge cut performance than Metis. In comparison with the distributed processing framework, ParMetis and XtraPuLP seem to provide better edge cut quality than our approach in some cases. However, XtraPuLP and ParMetis fail to meet the edge or vertex balance requirement. For example, XtraPuLP shows 3% fewer edge cuts than the proposed algorithm in the ''Orkut'' graph, but its edge balance ratio exceeds the maximum edge-imbalance ratio by 1.44 times. However, our proposed algorithm produces 16% fewer edge cuts than XtraPuLP while also satisfying the balance when it processed



(a) execution time

(b) edge-cut

(c) vertex-balance

(d) edge-balance

FIGURE 12. Result of performance, edge-cut, vertex-balance and edge-balance on large-cluster.

"DBpedia-link" and produced 11% fewer edge cuts than ParMetis in the "Orkut" graph. This experiment showed that the proposed algorithm can proceed to graph partitioning with higher edge cut quality than the original LP approach.

Finally, we analyzed both the vertex balance and edge balance. As shown in Figure 11(c) and (d), the proposed algorithm and Metis do not exceed either the maximum vertex-imbalance value (1.03) or the maximum edge- imbalance ratio (1.30). However, ParMetis does not fulfill the maximum vertex-imbalance ratio, and XtraPuLP does not fulfill the maximum edge-imbalance ratio. In the worst case, the vertex-imbalance ratio of ParMetis is 1.09 in the "Enwiki-2013" graph and the edge-imbalance ratio of XtraPuLP is 2.10 in the "LiveJournal" graph. We speculate that these distributed graph-partitioning algorithms focus more on edge cut quality than on balance, because the balancing process lowers the edge cut quality. In contrast, our proposed algorithm provides comparable edge cut quality to Metis while satisfying both the vertex balance and edge balance.

C. COMPARISON WITH OTHER SYSTEMS ON LARGE CLUSTER

In a large cluster experiment, we conducted the partitioning experiment on three large graphs in our datasets. In addition, we increased the number of partitions to 32 and 128 to investigate the edge cut quality according to the number of partitions. The experimental results on the large cluster are shown in Figure 12.

Each node in a large cluster is equipped with 244 GB memory. Metis and ParMetis are able to process the graph partitioning in the "DBpedia-link" and "Wikipedia-link" graphs in the large cluster, but they cannot process the "Twitter" graph data even on a large cluster. Although we did find a reference [30] that contains a Metis partition result for the "Twitter" graph, it used 1024 GB of memory. The execution time in Figure 12(a) shows that our algorithm performs better than others on all the graphs. For Metis and ParMetis, the larger the data size, the more severe the performance degradation. However, the proposed algorithm and XtraPuLP produced faster execution times, demonstrating that the results for the same data improve as the number of machines increases.

The proposed algorithm also produced an edge cut performance almost identical to that of Metis on the large cluster. XtraPuLP showed 10% and 16% fewer edge cuts than the proposed algorithm in the "Twitter/32" and "Twitter/128" graphs, respectively. However, the edge-imbalance ratio of XtraPuLP became more severe on a large cluster. In particular, its edge-imbalance ratio (5.99) exceeded the maximum edge-imbalance ratio (1.30) by 4.6 times on "Twitter/32." In addition, their edge-imbalance ratio became more severe on a large cluster than on the small cluster, even for the same graph. However, the proposed algorithm can produce a balanced graph topology, regardless of the data size and the number of machines.

We also analyzed the edge cut performance according to the number of partitions. In terms of edge cut and balance, the proposed algorithm showed a performance similar to that of Metis, regardless of the number of partitions. Meanwhile, the vertex-imbalance ratio and edge-imbalance ratio tended to become more severe as the number of partitions increased in ParMetis and XtraPuLP, respectively.

For example, the vertex-imbalance ratio of ParMetis is increased 1.04 times in ''Wikipedia-link/128'' as compared with ''Wikipedia-link/32.'' Meanwhile, the edge-imbalance ratio of XtraPuLP increased by 1.61 times on ''Twitter/128'' as compared with ''Twitter/32.''

VI. CONCLUSION

In this study, we developed a novel graph-partitioning algorithm that provides a high edge cut quality and excellent performance processing capability for parallel processing while sustaining the user-configured partitioning balance. Our novel two-way graph-partitioning algorithm consists of LP and stabilization phases to avoid the local optima problem. The edge cut quality and performance are improved with effective LP based on a score metric and the effectiveness of BSP-style lazy updating.

To verify the effectiveness of the proposed algorithm, we conducted experiments and compared the results with those of several well-known approaches, Metis, ParMetis, and XtraPuLP. The results show that Metis and ParMetis perform well and yield high edge cut quality, but scale poorly because of their significant amount of memory usage. In contrast, XtraPuLP showed high memory utilization and a good performance as compared to Metis and ParMetis but produced deficient edge cut and balance ratio results as compared to the other approaches.

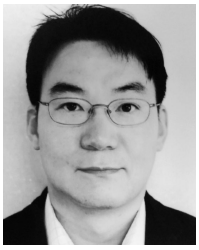
Our proposed algorithm outperformed the other approaches in terms of processing speed. In addition, it showed a better edge cut quality and balance ratio than the parallel graph-partitioning frameworks XtraPuLP and ParMetis. Because the proposed algorithm shows an improved graph-partitioning process in terms of scalability and performance, we expect to improve the performance of large-scale graph processing in distributed environments.

REFERENCES

[1] M. Junghanns, A. Petermann, M. Neumann, and E. Rahm, ''Management and analysis of big graph data: Current systems and open challenges,'' in *Handbook Big Data Technol.*, 2017, pp. 457–505.
[2] W. Fan and A. Bifet, ''Mining big data: Current status, and forecast to the future,'' *ACM SIGKDD Explorations Newslett.*, vol. 14, no. 2, pp. 1–5, Apr. 2013.
[3] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, ''The anatomy of the Facebook social graph,'' *CoRR*, vol. abs/1111.4503, pp. 1–17, May 2011.
[4] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, ''Billion-scale commodity embedding for E-commerce recommendation in alibaba,'' in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2018, pp. 839–848.
[5] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, ''Distributed GraphLab: A framework for machine learning and data mining in the cloud,'' *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012.
[6] U. Kang and C. Faloutsos, ''Big graph mining: Algorithms and discoveries,'' *ACM SIGKDD Explorations Newslett.*, vol. 14, no. 2, pp. 29–36, Apr. 2013.
[7] S. Yu, M. Liu, W. Dou, X. Liu, and S. Zhou, ''Networking for big data: A survey,'' *IEEE Commun. Surveys Tuts.*, vol. 19, no. 1, pp. 531–549, 1st Quart., 2017.
[8] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, ''Pregel: A system for large-scale graph processing,'' in *Proc. Int. Conf. Manage. Data*, 2010, pp. 135–146.
[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, ''PowerGraph: Distributed graph-parallel computation on natural graphs,'' in *Proc. Conf. Oper. Syst. Design Implement.*, 2012, pp. 17–30.
[10] B. Shao, H. Wang, and Y. Li, ''Trinity: A distributed graph engine on a memory cloud,'' in *Proc. Int. Conf. Manage. Data*, 2013, pp. 505–516.
[11] Apache. *Apache Giraph*. Accessed: Dec. 4, 2019. [Online]. Available: http://giraph.apache.org/
[12] S. Salihoglu and J. Widom, ''GPS: A graph processing system,'' in *Proc. 25th Int. Conf. Scientific Stat. Database Manage. (SSDBM)*, 2013, pp. 1–12.
[13] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, ''Graphx: Graph processing in a distributed dataflow framework,'' in *Proc. 11th USENIX Conf. Operating Syst. Design Implement.*, 2014, pp. 599–613.
[14] M. A. K. Patwary, S. Garg, and B. Kang, ''Window-based streaming graph partitioning algorithm,'' in *Proc. Australas. Comput. Sci. Week Multiconf.*, Sydney, NSW, Australia, Jan. 2019, Art. no. 51.
[15] J. Nishimura and J. Ugander, ''Restreaming graph partitioning: Simple versatile algorithms for advanced balancing,'' in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 1106–1114.
[16] S. Isabelle, ''Streaming balanced graph partitioning algorithms for random graphs,'' in *Proc. 25th Annu. ACM-SIAM Symp. Discrete Algorithms Soc. Ind. Appl. Math.*, 2014, pp. 1287–1301.
[17] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, ''Recent advances in graph partitioning,'' in *Proc. Algorithm Eng., Sel. Results Surv.*, 2016, pp. 117–158.
[18] Pothen, A., ''Graph partitioning algorithms with applications to scientific computing,'' in *Proc. Parallel Numer. Algorithms*, 1997, pp. 323–368.
[19] M. R. Garey, D. S. Johnson, and L. Stockmeyer, ''Some simplified NP-complete graph problems,'' *Theor. Comput. Sci.*, vol. 1, no. 3, pp. 237–267, Feb. 1976.
[20] B. W. Kernighan and S. Lin, ''An efficient heuristic procedure for partitioning graphs,'' *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, Feb. 1970.
[21] C. M. Fiduccia and R. M. Mattheyses, ''A linear-time heuristic for improving network partitions,'' in *Proc. 19th Design Autom. Conf.*, 1982, pp. 175–181.
[22] J. Huang and D. J. Abadi, ''Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs,'' *Proc. VLDB Endowment*, vol. 9, no. 7, pp. 540–551, Mar. 2016.
[23] C. Walshaw and M. Cross, ''JOSTLE: Parallel multilevel graph partitioning software—An overview,'' in *Proc. Mesh Partitioning Techn. Domain Decomposition Techn.*, 2007, p. 27–58.
[24] G. Karypis and V. Kumar, ''A fast and high quality multilevel scheme for partitioning irregular graphs,'' *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Jan. 1998.
[25] B. Monien, R. Preis, and R. Diekmann, ''Quality matching and local improvement for multilevel graph-partitioning,'' *Parallel Comput.*, vol. 26, no. 12, pp. 1609–1634, Nov. 2000.
[26] A. Abou-Rjeili and G. Karypis, ''Multilevel algorithms for partitioning power-law graphs,'' in *Proc. 20th IPDPS*, 2006, p. 10.
[27] B. Hendrickson. *Chaco: Software for Partitioning Graphs*. Accessed: Dec. 4, 2019. [Online]. Available: http://www.cs.sandia.gov/bahendr/chaco.html
[28] F. Pellegrini. *Scotch*. Accessed: Dec. 4, 2019. [Online]. Available: http://www.labri.fr/pelegrin/scotch
[29] G. Karypis and V. Kumar, ''Parallel multilevel k-way partitioning scheme for irregular graphs,'' in *Proc. ACM/IEEE Conf. Supercomputing (CDROM)*, 1996, pp. 278–300.
[30] C. Chevalier and F. Pellegrini, ''PT-scotch: A tool for efficient parallel graph ordering,'' *Parallel Comput.*, vol. 34, nos. 6–8, pp. 318–331, Jul. 2008.
[31] P. Sanders and C. Schulz. *KaHIP–Karlsruhe High Quality Partitioning*. Accessed: Dec. 4, 2019. [Online]. Available: http://algo2.iti.kit.edu/documents/kahip/index.html
[32] H. Meyerhenke, P. Sanders, and C. Schulz, ''Parallel Graph Partitioning for Complex Networks-Balanced Graph Partitioning,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2625–2638, 2017.
[33] U. N. Raghavan, R. Albert, and S. Kumara, ''Near linear time algorithm to detect community structures in large-scale networks,'' *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 76, no. 3, Sep. 2007, Art. no. 036106.
[34] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, ''Partitioning trillion-edge graphs in minutes,'' in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 646–655.

[35] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.
[36] N. K. Ahmed, N. Duffield, J. Neville, and R. Kompella, "Graph sample and hold: A framework for big-graph analytics," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2014, pp. 589–597.
[37] P. Sanders and C. Schulz, "High quality graph partitioning," *Graph Partitioning Graph Clustering*, vol. 588, no. 1, pp. 1–17, 2012.
[38] M. J. Barber and J. W. Clark, "Detecting network communities by propagating labels under constraints," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 80, no. 2, Aug. 2009, Art. no. 026129.
[39] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *Proc. IEEE 30th Int. Conf. Data Eng.*, Mar. 2014, pp. 568–579.
[40] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proc. 6th ACM Int. Conf. Web Search Data Mining (WSDM)*, 2013, p. 507.
[41] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct. 2014, pp. 481–490.
[42] EC2. *Amazon Elastic Compute Cloud*. Accessed: Dec. 4, 2019. [Online]. Available: http://aws.amazon.com/ec2
[43] J. Leskovec. *Stanford network analysis package (SNAP)*. Accessed: Dec. 4, 2019. [Online]. Available: https://snap.stanford.edu/data/
[44] J. Kunegis. *KONECT—The Koblenz Network Collection*. Accessed: Dec. 4, 2019. [Online]. Available: http://konect.uni-koblenz.de/

MINHO BAE received the B.S. and Ph.D. degrees in computer engineering from Ajou University, Suwon, South Korea, in 2012 and 2019, respectively. He is currently a Researcher with the Institute for Information and Communication, Ajou University. His general research area is parallel/distributed processing. His current research interests include optimizing distributed deep learning, large-scale graph partitioning, and large-scale data processing model.



MINJOONG JEONG received the Ph.D. degree in frontier science (mechanical design) from Tokyo University, Tokyo, Japan, in 2004. He is currently a Principal Researcher with the National Supercomputing Center, Korea Institute of Science and Technology Information (KISTI). He is also a Professor with the University of Science and Technology (UST), South Korea. His research interests include multiobjective/multicriterion optimization, data clustering/pattern recognition using soft computing, and evolutionary algorithm for optimization.

SANGYOON OH received the Ph.D. degree from the Computer Science Department, Indiana University Bloomington, Bloomington, IN, USA. He is currently a Professor with the School of Information and Computer Engineering, Ajou University, South Korea. Before joining Ajou University, he worked for SK Telecom, South Korea. His main research interests are parallel and distributed computing, high performance computing, distributed deep learning, and graph data processing.