GCache: Neighborhood-Guided Graph Caching in a Distributed Environment

Ye Yuan, Xiang Lian, Lei Chen, Guoren Wang, Jeffrey Xu Yu, Yishu Wang, Yuliang Ma, *Member IEEE,*

Abstract—Distributed graph systems are becoming extremely popular due to their flexibility, scalability, and robustness in big graph processing. In order to improve the performance of the distributed graph systems, caching is a very effective technique to achieve fast response and reduce the communication cost. Existing works include online and offline caching algorithms. Online caching algorithms (such as least recently used (LRU) and most recently used (MRU)) are lightweight and flexible, however, neglect the topological properties of big graphs. Offline caching algorithms (such as node pre-ordered) consider the graph topology, but are very expensive and heavy. In this paper, we propose a novel caching mechanism, GraphCache (GCache), for big distributed graphs. GCache consists of an offline phase and an online phase, which inherits the advantages of online and offline caching algorithms. Specifically, the offline phase provides a caching model based on the bipartite graph clustering and give efficient algorithms to solve it. The online phase caches and schedules the graph clusters output from the offline phase, based on the LRU and MRU strategies. GCache can be seamlessly integrated into the state-of-the-art graph processing systems, e.g., Giraph. Finally, our experimental results demonstrate the feasibility of our proposed caching techniques in speeding up graph algorithms over distributed big graphs.

Index Terms—Distributed Caching, Large Graph

1 Introduction

There is a huge amount of information surrounding us that can be represented in the form of graphs. Examples of these graphs include social networks, graphs of road networks, dependency graphs for software, etc. Moreover, the size of these graphs has rapidly grown and can now reach up to hundreds of billions of nodes and trillions of edges [1]. For example, the World Wide Web now contains more than 50 billion Web pages and more than one trillion unique URLs [2]. The de Brujin graph constructed for tackling genome assembly problems may contain as many as 4^{20} nodes [3]. Systems such as Pregel [4], GraphX [5] and Giraph [6] are some of the plethora of graph processing systems being adopted to process these large graphs. To enable fast computations on large graphs, these systems are typically run in a distributed manner.

One of the key performance factors of a distributed graph system, compared to its centralized counterpart, is the *communication cost*. In some cases, the communication cost can be the bottleneck of the system and suppress the advantages of the distributed graph system (such as parallelized computing power, storage capability, etc.). Hence, reducing the communication cost is one of critical issues for the success of a distributed graph system. We may encounter this challenge from different aspects, for example, when we design a communication-free approximate algorithm, optimize the network infrastructure, and so on. In this paper, we focus on the *caching*, that is, caching remote nodes' information (nodes in other machines) in local memory to reduce the communication costs during big graph processing.

Distributed Graph Caching. Caching is one of effective mechanisms to reduce remote communications. Whenever a node needs to access its remote neighbors' information, the caching system first checks from its local cache. If the information is already available in the cache, this information will be directly fetched. Otherwise, remote accesses are issued. Obviously, if caching is effective (i.e., if the *hit ratio* is high), we can avoid a large number of remote accesses.

Example 1: Figure 1 illustrates an example, where machine M_1 contains nodes $\{1,2,3,4,5,6\}$ in its memory and their *remote neighbors* $\{a,b,c,d\}$ are included in the memory of machine M_2. When we process nodes of M_1, we usually access their remote neighbors in M_2. We set a cache in M_1 to save the remote accesses from $\{1,2,3,4,5,6\}$. Assume that the maximum size of the cache is 2. We can save remote accesses from nodes $\{1,2,3\}$, if we cache nodes $\{a,b\}$, i.e., the hit ratio for nodes $\{1,2,3\}$ is very high. In this caching, however, the hit ratio may be zero for nodes $\{4,5,6\}$. □

Existing caching mechanisms for graphs include online and offline fashions. Typical online works include the works such as [7], [8], and [9]. These works focus on a replacement strategy to improve the cache hit ratio. Their common principle is to discard the information that was not needed for the longest time. Hence, their adopted caching schemes such as LRU and MRU tend to use the previous access patterns to predict whether a data item will be accessed in the future. This fashion makes replacement decision according to the most recently accessed information.

The advantage of online algorithms is that they are lightweight and flexible, i.e., an online algorithm takes only an $O(S)$ time complexity (where S is the cache size) and the algorithm is triggered whenever needed. This feature is very suitable for distributed environments. The shortcoming of online algorithms is that they neglect the topological structure of graph data. Following Example 1, the online algorithm may cache nodes $\{b, c\}$, as this pair of nodes obtains more remote accesses than any pair of nodes in $\{a, b, c, d\}$ according to the MRU or LRU scheme. However, as shown in Figure 1, it is better to cache nodes $\{a, b\}$ or $\{c, d\}$ than nodes $\{b, c\}$. The reason is that remote requests usually come from the nodes $\{1, 2, 3\}$ (resp. $\{4, 5, 6\}$) simultaneously. In addition, we can alternately cache nodes $\{a, b\}$ and $\{c, d\}$ to maximize the hit ratio for nodes $\{1, 2, 3, 4, 5, 6\}$.

Most offline works concentrate on the CPU-based graph caching schemes, which predefine a total order on nodes to speed up graph processing [10], [11], [12], [13], [14] (i.e., graph algorithms access the nodes in the defined order.). The predefined order keeps nodes frequently accessed together stored closely in main memory so that they are more likely to be loaded into cache together by one single cache line transfer. The predefined order of nodes is determined by the topological structure of graph data. The advantage of offline algorithms is that they consider the graph structure. The shortcoming of offline algorithms is that they regard the entire graph as a whole, which is unsuitable for distributed environments, especially the node-centric computational model introduced as follows.

Current distributed graph computing frameworks advocate the *node-centric computational model* [4], [5], [15], for which no order is posed on nodes to be processed. The node-centric computational model works very well for distributed graph algorithms, since nodes under this model independently update their information and the computation on a node only depends on its neighbors. This *local* computing mode is very suitable for graph algorithms performed in a distributed environment.

Our Solutions and Contributions. Motivated by the advantages of online and offline caching algorithms, in this paper, we propose a novel graph caching scheme (GCache) for the node-centric computational framework. Specifically, GCache is inspired by an observation: *nodes sharing similar remote neighbors should be accessed together.* In other words, if local nodes can be clustered in a way such that nodes in the same cluster share many similar remote neighbors, the caching will be quite effective.

Example 2: Figure 1 briefly illustrates how GCache is performed, where the maximum cache size is 2 (i.e., the cache contains at most 2 nodes.). GCache consists of an offline phase and an online phase. In the offline phase, GCache groups nodes $\{1, 2, 3, a, b\}$ as one cluster C_1 and nodes $\{4, 5, 6, c, d\}$ as another cluster C_2, based on the topological structure of the bipartite graph. In the online phase, GCache processes and caches clusters instead of nodes. For instance, when we process nodes of M_1, we process the node set $\{1, 2, 3\}$ of C_1 (resp. $\{4, 5, 6\}$ of C_2) as an unit. While we process the node set $\{1, 2, 3\}$, we cache the node set $\{a, b\}$ also in C_1. GCache replaces $\{a, b\}$ with $\{c, d\}$ in the cache, while we process the node set $\{4, 5, 6\}$

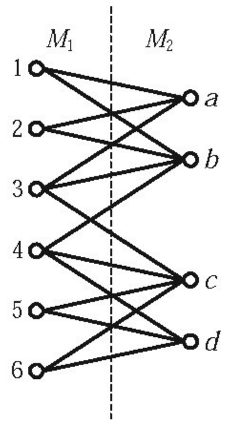


Fig. 1: Example of the proposed caching algorithm.

The cached node set $\{a, b\}$ or $\{c, d\}$ satisfies most requests from nodes in the same cluster. □

Example 2 shows that GCache consists of an online phase and an offline phase. The offline phase partitions the bipartite graph into different clusters. The online phase caches and schedules remote nodes in each cluster. Based on this scheme, GCache inherits the advantage of offline caching algorithms, i.e., GCache clusters nodes based on the graph structure. GCache also inherits the advantage of online caching algorithms, i.e., GCache can flexibly replace one cluster with another according to the LRU or MRU scheme. In this paper, we model the offline phase as a clustering problem on a bipartite graph. We call the problem *Local Clustering*, since the caching only depends on the local information on a machine. This local feature is very suitable for the node-centric computational model. In addition, GCache can efficiently solve the local clustering for a big graph with billions of nodes. Our experiment shows that GCache is time and communication efficient, at node-centric computational systems, e.g., Giraph [6].

GCache is a general caching scheme for any graph algorithm implementable in a node-centric computational model. We optimize GCache for a class of very important graph algorithms that are used to solve graph connectivity problems. Graph connectivity problems refer to finding connected components (CC) in an undirected graph, finding strongly connected components (SCC) in a directed graph and asking if two given nodes are reachable (RE) from one to the other in a directed graph. Obviously, the three graph problems have lots of applications in practice, and many advanced graph algorithms are based on the three graph problems. We apply the *percolation theory* [16] to optimize GCache. Specifically, we compute a very small probability p for a large graph by applying the percolation theory. As a consequence, we can cache $S \cdot p$ edges to achieve the same effect as we cache S edges for graph connectivity problems (S is the cache size). For example, if $p = 0.01$, it means that the caching size of GCache can be saved 99% without losing its effect.

The remainder of this paper is organized as follows. We formally define the problem and give its complexity in

TABLE 1: Notation.

Symbol	Description
G	the distributed graph
F_i	a fragment of G
$F_i.I$, $F_i.O$	the border nodes of F_i
$B(V,E)$=$B(X \cup Y, E)$	the meta graph of F_i
$C_i = (X_i \cup Y_i, E_i)$	a cluster of $B(V,E)$
$E(X_i, Y_i)$	the edge set of C_i
$E(C_i, C_j)$	the crossing edges between C_i and C_j
N_t	a maximum flow network
$f(u,v)$	an edge flow of N_t
$a(u,v)$	an edge cost of N_t
$c(u,v)$	an edge constraint of N_t
p_t	a threshold probability

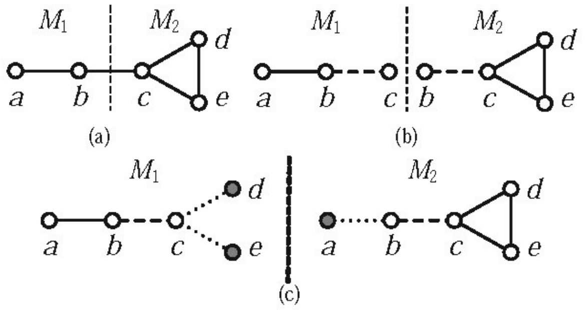


Fig. 2: A big graph G is distributed two machines M_1 and M_2: (a) the virtual G, (b) the actual G, and (c) the actual caching of G.

Section 2. In Section 3, we propose the general solution for the offline phase of GCache. In Section 4, we optimize the general solution for graph connectivity algorithms. In Section 5, we introduce the detailed offline phase of GCache. We discuss the results of performance tests in Section 6 and the related works in Section 7. Finally, in Section 8 we draw our conclusions.

2 Problem Definition

We summarize the notations of this paper in Table 1.

2.1 Basic Concepts

Graphs. We consider graphs $G = (V, E, L)$, directed or undirected, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges, and; (3) each node v in V (resp. edge $e \in E$) carries $L(v)$ (resp. $L(e)$), indicating its content, as found in knowledge graphs and social networks. Graph $G' = (V', E', L')$ is called a subgraph of G if $V' \subseteq V$, $E' \subseteq E$, and for each node $v \in V'$ (resp. each edge $e \in E$), $L'(v) = L(v)$ (resp. $L'(e) = L(e)$).

Distributed Graphs. Given a number c, a strategy P partitions graph G into disjoint fragments $F = (F_1, ..., F_c)$ such that each $F_i = (V_i, E_i, L_i)$ is a subgraph of G, $E = \bigcup_{i\in[1,c]} E_i$, $V = \bigcup_{i\in[1,c]} V_i$, and F_i resides at a machine M_i. Denote by

- $F_i.I$ the set of nodes $v \in V_i$ such that there is an edge (v', v) incoming from a node v' in F_j $(i \neq j)$;
- $F_i.O$ the set of nodes v' such that there exists an edge (v, v') in E, $v \in V_i$ and v' is in some $F_j (i \neq j)$; and
- $F.O = \bigcup_{i\in[1,c]} F_i.O$, $F.I = \bigcup_{i\in[1,c]} F_i.I$, $F.O = F.I$.

We refer to nodes in $F_i.I \cup F_i.O$ as the *border nodes* of F_i *w.r.t.* P. If G is an undirected graph, then it holds that $F_i.I = F_i.O$. Without loss of generality, we focus on the *undirected graph* in the following of this paper. For an edge $e = (v, v')$, $v \in F_i.O$ and $v' \in F_j.O$ $(i \neq j)$, we refer to e as a *crossing edge* of F_i and v' a *virtual node* of F_i. Machine M_i maintains crossing edges and virtual nodes so that F_i communicates with another fragment residing on another machine.

For example, in Figure 2, a graph G is partitioned into two fragments (F_1 and F_2) across two machines, M_1 and M_2. Figure 2(a) gives a simplified illustration, whereas Figure 2(b) shows the actual stored fragment in each machine. As shown in the figure, M_1 maintains the crossing edge (b, c) and its virtual node c. So does M_2. Thus, communication exists between F_1 and F_2, so that any graph algorithm can be processed over the distributed graph G.

To define distributed graph caching, we need the following definitions. All nodes in F_i except for the virtual nodes are referred to as *inner nodes* of F_i. Similarly, all edges in F_i except for the crossing edges are referred to as *inner edges* of F_i. All nodes in F_i are referred to as *local nodes* which include inner and virtual nodes. All edges in F_i are referred to as *local edges* which include inner and crossing edges. For example, in Figure 2(b), a and b are inner nodes of F_1, and (a, b) is an inner edge of F_1. Nodes a, b and c are local nodes of F_1. Edges (a, b) and (b, c) are local edges of F_1.

Note that a crossing edge (v, v') between F_i and F_j are both kept in F_i and F_j. Node v' is a virtual node of F_i, and a border node of F_j. For an inner edge (v', v'') of F_j, let v' be a border node of F_j and v'' an inner node of F_j. Then, for v' in F_i, v'' is referred to a *remote neighbor* of v' in F_j. For example, in Figure 2 (c), d and e are remote neighbors of c in F_1.

Distributed Large Graph Caching. We use $N(u)$ to denote the neighbor of a node u. Thus, the degree of the node u is $d(u) = |N(u)|$.

Given a graph $G(V, E, L)$ and its partitioning on distributed machines, we can define a meta graph for each machine. The following discussion will focus on the meta graph of each machine.

Definition 1 (Meta Graph on a Machine): A meta graph on a machine M_i is a bipartite graph $B(V, E)$, where $V = X \cup Y$ and X are virtual nodes in machine and Y are remote neighbors of X.

For example, in Figure 2(c), $X = \{c\}$ and $Y = \{d, e\}$ for M_1.

As claimed in the introduction, our aim is to reduce traffic costs of a distributed graph algorithm. We observe that the behaviors of distributed graph algorithms are as follows: *nodes sharing similar remote neighbors are usually accessed together.* In other words, if local nodes can be grouped in a way such that nodes in the same group share many similar remote neighbors, the caching will be quite effective. In addition, F_i is usually still very large after G is partitioned. Therefore, local nodes X of F_i can be logically divided into parts $(X_1, X_2, ..., X_k)$ such that nodes in each X_i share many similar remote neighbors.

Here each X_i corresponds to a cache C_i which maintains X_i's remote neighbors. When nodes of X_i request their remote neighbors, they first check if C_i contains the remote neighbors. If the answer is yes, they fetch the neighbors from C_i; otherwise, they initiate remote requests.

An important factor in a real caching system is that, the caching system can only cache a limited number of remote neighbors. To reflect this requirement, we use a parameter S as the cache size constraint.

Problem Model. Based on the caching mechanism above, we next give an optimal model for the distributed graph caching. Given a meta graph $B(X \cup Y, E)$ and $X' \subseteq X, Y' \subseteq Y$, we use $E(X', Y')$ to denote the edge set between X' and Y'. Now we are ready to give the problem. In this problem, we hope to maximize the number of remote accesses which are saved.

We define the local clustering to achieve the maximum savement of communication costs.

Definition 2 (Local Clustering (LC)): Given a meta graph $B(X \cup Y, E)$ and an integer S, find a partitioning $C_1, C_2, ...,$ and C_k on the meta graph, such that

- each $C_i = (X_i \cup Y_i, E_i)$ is a bipartite graph with $X_i \subseteq X$, $Y_i \subseteq Y$,
- for each i, $|Y_i| \leq S$;
- $\sum_i |E(X_i, Y_i)|$ is maximized.

Here, $|E(X_i, Y_i)|$ is the number of the saved communications for nodes X_i, if we cache the Y_i nodes. Thus, $\sum_i |E(X_i, Y_i)|$ is the total number of the saved communications. For X_i, its **cached nodes** are Y_i and **cached edges** are edges $E(X_i, Y_i)$. In the caching, we also cache the nodes' and edges' attributes, i.e., $L(v)$ and $L(e)$ for $v \in Y_i$ and $e \in E(X_i, Y_i)$. As we will show later, GCache consists of an online phase and an offline phase. The offline phase partitions the meta graph into different clusters. The online phase caches and schedules Y_i nodes in each cluster. The caching process is executed along with graph algorithms. Therefore, the attribute values of cached Y_i nodes are naturally updated by graph algorithms.

Note that, X_i and X_j $(i \neq j)$ are not necessarily disjoint with each other, but Y_i and Y_j should be disjoint. This point will lead to low network overhead. For example, in Figure 3, partial overlap of X_1 and X_2 is better than their complete separation for the caching. Disjoint Y_1 and Y_2 consume fewer space costs. Moreover, we have a size-constraint on Y_i and nothing on X_i, which assures in a cluster C_i many nodes of X_i have common remote neighbors in Y_i.

This problem definition is for the offline phase which is the basis of the online phase.

Note that the local clustering is designed for any graph algorithm, not some specific graph algorithms. The design is similar to an index that can support any type of queries. To achieve this goal, the local clustering assumes that all nodes are active and have equal access frequencies of their neighbors across all round of iterations.

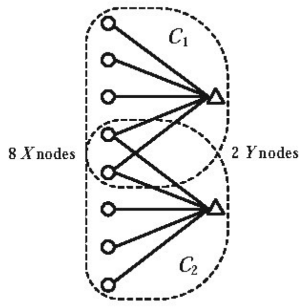


Fig. 3: Example of the constraint on Y_i and nothing on X_i.

2.2 Problem Complexity

In this subsection, we will show that the optimization problem (LC) (given in Definition 2) in our model is NP-hard.

Theorem 1: The local clustering problem is NP-hard.

3 General Solution for Offline Phase of GCache

3.1 Connection to Graph Partitioning

In this subsection, we solve the LC problem by graph partitioning (GP) which is defined as follows.

Definition 3 (Graph Partitioning (GP)): Given a meta graph $B(X \cup Y, E)$, find a partitioning $C_1, C_2, ..., C_k$ on the meta graph, such that

- (1) each $C_i = (X_i \cup Y_i, E_i)$ is a bipartite graph with $X_i \subseteq X$, $Y_i \subseteq Y$,
- (2) $|E(X_i, Y_i)| = |E(X_j, Y_j)|$ for $i \neq j$, and
- (3) $\sum |E(C_i, C_j)|$ is minimized.

where $E(C_i, C_j)$ denotes the crossing edges between C_i and C_j.

There are three differences between GP and LC:

- In GP, the objective is to minimize the number of crossing edges between different clusters, whereas in LC the objective is to maximize the number of inner edges within clusters.
- In GP, the partition number is given, whereas in LC no partition number is given.
- In GP, no size constraint is posed, whereas in LC we need to ensure that the size constraint is satisfied.

Next, we solve the first issue, i.e., the different objectives. The solutions to the second and third issues above (i.e., partition number and size constraint) will be introduced later.

Different Objectives. Given a meta graph $B(X \cup Y, E)$, since its edge set $E(X, Y)$ is fixed, the maximization of inner edge number equals to the minimization of the crossing edge number. Then, the objective of GP and LC is the same. In the sequel, we propose a graph partitioning algorithm by solving the second and third issues.

3.2 Algorithmic Framework

Since the LC problem is NP-hard, our solution to it consists of the following two greedy steps. In the first step, we get an initial partitioning P, which should be a lightweight algorithm e.g., a random partitioning. In the second step, we advocate the technique of *label propagation* (LP) to iteratively refine the initial partitioning P until a halting condition is satisfied. The reason we adopt LP is that LP can be easily parallelized, affording a scalable implementation on a node-centric system. Algorithm 1 is the pseudo-code of our local clustering framework which consists of two steps, i.e., initialization (line 2) and refinement (lines 3-6).

Note that LP assumes undirected graphs [17]. However, usually graphs are directed (e.g., social network). To use LP, we would need to convert a directed graph into undirected one. The approach is to create an undirected edge between nodes u and v whenever at least one directed edge exists between nodes u and v in the directed graph.

Below, we first give the haling condition, and then introduce the detailed contents of the two steps in the sequel subsections.

Algorithm 1 LocalClustering($B(V = X \cup Y,\ E)$)

```
1: Make $B$ undirected if $B$ is a directed bipartite graph;
2: $P$ = Obtain an initial solution by a random partitioning;
3: repeat
4:     UpdateX($P$);
5:     UpdateY($P$);
6: until A halting condition is satisfied
```

Halting Condition. LP usually needs a very large number of iterations to converge [17] and hence we provide a hallting condition to guarantee a fast convergence of LP without sacrificing the partitioning quality. We take the following strategy: At a given iteration, we define the score of the partitioning for the bipartite graph B as the sum of the current scores of each node, $score(B) = \sum_{v \in B} score(v, l)$. The value of $score(v, l)$ will be explained later in Equation (1).

We consider a partitioning to be in a stable state, when the score of the bipartite graph B is not improved more than a given ϵ for more than γ consecutive iterations. LP halts when a stable state is reached. Although through ϵ we can control the trade-off between the cost of executing LP for more iterations and the improvement obtained by the score function, with γ it is possible to require a stricter definition of stability, as the absence of improvement is accepted for a larger number of iterations.

3.3 Initialization Step

In this step, an initial partitioning could be done by random partitioning (e.g., RANDOM [18]) or other algorithms (e.g., METIS [19]). In general, the random partitioning is the lightest and can handle a very big graph. Thus, we choose the random method to perform the initial partitioning. One widely used random partitioning mechanism is that each node chooses a partition uniformly at random from all k possible ones. After all the nodes finish their selections, the number of nodes on each partition will follow the Binomial distribution $B(|n|, \frac{1}{k})$, where $n = |V(X, Y)|$. Under this random partitioning, an edge's two end nodes have the probability $1 - \frac{1}{k}$ to be on different partitions. The edge cut therefore will follow the Binomial distribution $B(m, 1 - \frac{1}{k})$, where $m = |E(X, Y)|$.

Partitioning Number. The random partitioning algorithm needs a pre-determined partition number [18], which however is not required in the LC problem. Thus, we should select an appropriate partition number. A naive solution is to first enumerate all valid partitioning numbers, then run RANDOM under each partitioning number, and finally choose the best partitioning as the final result. Note that the best partitioning is the partition with smallest $\sum |E(C_i, C_j)|$. Obviously, the key to improving the performance of this naive solution is to establish tighter upper and lower bounds on the partitioning number. The tighter bound will lead to less enumeration cost. Next, we establish such bounds on the valid partitioning number k.

According to Definition 2, k has a lower bound $\lceil |Y|/S \rceil$ and an upper bound $|Y|$. These two bounds imply that we need to run RANDOM $O(|Y|)$ times, which is very expensive. A wiser enumeration strategy is *stopping the enumeration when* $k > \lfloor \frac{2|Y|}{S} \rfloor + 1$, because in Theorem 2 we prove that there exists an optimal solution when $\lceil |Y|/S \rceil \le k \le \lfloor \frac{2|Y|}{S} \rfloor + 1$.

Lemma 1 (Monotonicity): Let $Q(k)$ be the minimal number of crossing edges among all possible k-partitionings on the meta graph B. When $k \ge \lfloor \frac{2|Y|}{S} \rfloor + 1$, $Q(k) \le Q(k+1)$. □

Then we have a direct consequence of Lemma 1

Theorem 2 (Optimal Solution): There is an optimal k-partitioning when

$$\lceil |Y|/S \rceil \le k \le \lfloor \frac{2|Y|}{S} \rfloor + 1$$

□

Algorithm. Now we are ready to give the solution based on RANDOM. The algorithm (shown in Algorithm 2) takes the bipartite meta graph B and size constraint S as inputs. In the algorithm, we enumerate the values of k from $\lceil |E(X, Y)|/S \rceil$ to $\lfloor \frac{2|E(X,Y)|}{S} \rfloor + 1$ based on Theorem 2.

Algorithm 2 RANDOM based LC algorithm

```
Input: $B(X \cup Y, E)$, $S$
1: $k \leftarrow \lceil |Y|/S \rceil$;
2: while $k \le \lfloor \frac{2|Y|}{S} \rfloor + 1$ do
3:     Run RAONDOM on $B$ under the partition number
       $k$;
4:     $k \leftarrow k + \pi$;
5: end while
6: return The best partitioning found in the loop;
```

The criterion of determining the best partition P is that the cut size of P is the smallest among all the partitions output from Algorithm 2. Note that the cut size of a partition P is the number of all edge cuts in P. A random algorithm is efficient to determine the best partition. AltThough the random algorithm cannot output a balanced partition P, the smallest cut of P guarantees one objective for the local clustering. Then, the refining phase balances the initial partition gradually.

The time complexity of Algorithm 2 is $O(\frac{|Y|}{S\pi} cost(RAND\ OM))$. π is actually a tradeoff between effectiveness and efficiency. A smaller π in general leads to a higher quality solution and consumes more running time. On the contrary, a larger π results in a lower quality solution but with less running time. In the experiments, we will set different π and determine the optimal one.

3.4 Refinement Step

Size-constraint. In this step, we refine the initial partitioning, in which the issue of size-constraint should be solved. Specifically, we propose the sequel refining mechanism.

For convenience, we use the *labeling* instead of partitioning in the following explanation. The nodes in one partition are assigned with the same label and different partitions have different labels. For convenience, we call the node in X *X node* and node in Y *Y node*. The refinement runs iteratively, and it could be divided into two parts, *Update X* and *Update Y*. Here we briefly introduce these two parts.

- In Update X, we fix the labels of Y nodes, and then a greedy strategy is applied to calculate the optimal labels for X nodes.
- In Update Y, we fix the labels of X nodes, and then calculate the optimal labels for Y nodes by solving the maximum-cost maximum-flow problem.

3.4.1 Update X

In this step, we fix the labels of Y nodes and refine the label for each X node. We greedily refine the label: for each node v in X, we choose the most popular label from its neighbors in Y. If there exists more than one the most popular labels, we randomly select one. More formally, an X node x shows the preference to label l with a high score:

$$score(x, l) = \sum_{y \in N(x)} \delta(\alpha(y), l) \qquad (1)$$

where δ is the Kronecker delta, and α is the labeling function such that $\alpha(y) = l$ if label l is assigned to node y. The node x updates its label to the label l_x that maximizes its score according to the update function:

$$l_x = argmax_l\ score(x, l) \qquad (2)$$

Obviously, we can achieve this step in linear time and this step will reach the optimal labels for X nodes under the case that all Y nodes' labels are fixed.

3.4.2 Update Y

In this step, we will fix the labeling of X nodes and refine the labels of Y nodes. Compared with Update X, here we need to take into account the size restrictions, which makes it more difficult. We first show how to model this problem as the maximum-cost maximum-flow (MCMF).

Definition 4 (Maximum Flow Problem): Given a directed graph $G = (V, E)$ with source $s \in V$ and sink $t \in V$, where edge $(u, v) \in E$ has capacity $c(u, v) > 0$. A flow is a mapping $f : E \to \mathbb{R}^+$, denoted by $f(u, v)$, subject to the following two constraints:

- Capacity constraints: $f(u, v) \le c(u, v)$
- Flow conservation: $\sum_{u:(u,v)\in E} f(u, v) = \sum_{u:(v,u)\in E} f(v, u)$ for each $v \ne s, t$

The maximum flow problem is to maximize $|f| = \sum_{v:(s,v)\in E} f(s, v)$.

Definition 5 (MCMF Model): Given a flow network, that is, a directed graph $G = (V, E)$ with source $s \in V$ and sink $t \in V$, where edge $(u, v) \in E$ has capacity $c(u, v) > 0$ and cost $a(u, v) \ge 0$. The maximum-cost maximum-flow problem is to find a maximum flow $f : E \to \mathcal{R}$ such that the total cost of the flow is maximized: $\sum_{(u,v)\in E} a(u, v) \cdot f(u, v)$.

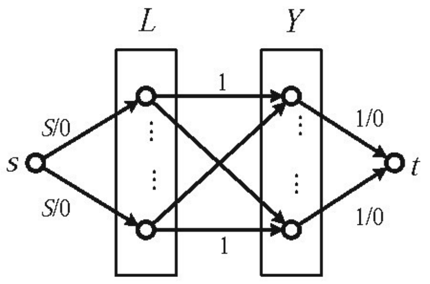


Fig. 4: Flow network model N_c of Update Y nodes.

Model. Let $L = \{l_1, l_2, \ldots, l_t\}$ denote the labels of X nodes. After Update Y, each node in Y is assigned with a label (say l_i) from L, which means that $v \in Y$ joins the cluster with label l_i. Let $E(l_i \to v)$ be the number of edges between v and nodes in X that have labels l_i.

In our model, the network N_c is constructed in the following manner (see Figure 4):

1) Except for the unique source node (s) and a unique sink node (t), two groups composing the intermediate nodes: the first group is the label set L and the second group is Y.
2) From each label l_i to each node v in Y, there is a directed edge e assigned with the unit capacity. The cost of e is assigned as $E(l_i \to v)$. From each node in Y to the sink, there is an edge with unit capacity and zero cost. From the source to each label, there is an edge with capacity S and zero cost.

The motivation behind the construction above is as follows: Since each node in Y can be labeled by any label in L, there is a directed edge (l_i, v) from each $l_i \in L$ to each node $v \in Y$. $f(l_i, v) = 1$ means that the edge from label l_i to node v has unit *flow*, which implies that v will be labeled as l_i. Note that each node in Y can have only one label, and each label cannot be assigned to more than S nodes in Y. Hence, $c(s, l_i) = S$ for each $l_i \in L$ and $c(v, t) = 1$ for each $v \in Y$.

Lemma 2: Any integral maximum flow in network N_c corresponds to a valid labeling solution to $B(X \cup Y, E)$ and the cost of the flow is the number of inner-partition edges. □

Theorem 3: Any feasible labeling solution corresponds to a maximum flow, whose cost is equal to the number of inner edges in the labeling solution. □

From the theorem above, we could see that the maximum integral flow with the maximum cost is corresponding to the labeling solution with the maximum number of inner edges. Notice that in N_c, each edge has an integral capacity, so there exists an integral maximum flow with the maximum cost and many algorithms can solve this problem, for example, the FordCFulkerson algorithm [20]. The algorithm is very efficient, as the algorithm applied only to a bipartite graph (e.g., Figure 4).

Theorem 4: Given a bipartite graph $B(V, E)$ on machine M, the time complexity of the refinement step is $O(|E| + f|E|)$, where f is the maximum flow of the network N_c. Therefore, the total time complexity of LC is $O(|E| + f|E| + \frac{|Y|}{S\pi}cost(RANDOM))$. □

3.5 Extending to Overlapping Clusters

Recall that a good caching strategy should assure that different clusters overlap in X nodes and disjoin Y nodes. The proposed algorithm so far leads to completely disjoint clusters in X nodes and Y nodes. Thus, we need to extend the proposed algorithm to obtain overlapping clusters only in X nodes.

In the above algorithm, an X node label identifies a single cluster to which the node belongs. To find overlapping clusters, we need to allow a node label to contain more than one cluster identifier.

Specifically, we could label each X node x with a set of pairs (c, b), where c is a cluster identifier and b is a belonging coefficient, indicating the strength of x's membership of cluster c, such that all belonging coefficients for x sum to 1. Each update step would set x's label to the union of its neighbors' labels, sum the belonging coefficients of the clusters over all neighbors and normalize. More precisely, assuming a function $b_t(c, x)$ that maps an X node x and cluster identifier c to its belonging coefficient in updating iteration t,

$$b_t(c, x) = \frac{\sum_{y \in N(x)} b_{t-1}(c, y)}{|N(x)|}, \tag{3}$$

where $N(x)$ denotes the set of neighbors of x.

In the algorithm, the label of a node in iteration t is always based on its neighbors' labels in iteration $t - 1$.

In the algorithm, we require to retain more than one cluster identifier in each label without maintaining all of them. Our algorithm uses the belonging coefficients for this purpose: during each update step, we first build the node label as above and then delete the pairs whose belonging coefficient is less than some threshold. We set this threshold as a reciprocal, $1/\mu$, where μ is the parameter of the algorithm. Since the belonging coefficients in each label sum to 1, μ represents the maximum number of clusters to which any node can belong to.

After removing pairs from the node label, we renormalize it by multiplying the belonging coefficient of each left pair by a constant so that they sum to 1. This process continues till the halting condition is satisfied.

The extended algorithm generalizes the above proposed algorithm. When $\mu < 2$ they are essentially the same: the label of a node can contain only one cluster identifier, each update step retaining the identifier used by the maximum number of neighbors. In the experiment, we set $\mu = 4$, i.e., a node belongs at most 4 clusters.

The next section will introduce how to optimize the general caching algorithm.

4 Optimized General Solution for Graph Connectivity Problems

Graph connectivity problems are important and fundamental in the field of graph algorithms, since many advanced graph algorithms need to solve the graph connectivity problems firstly. Generally speaking, graph connectivity problems include finding connected components (CC) in an undirected graph, finding strongly connected components (SCC) in a directed graph and asking if two given nodes are reachable (RE) from one to the other in a directed graph. Obviously, the three graph problems have plenty of applications in practice, and many advanced graph algorithms are based on the three graph problems. In this section, we aim to optimize the general solutions for the graph connectivity problems.

Many graph algorithms can be utilized to solve the graph connectivity problems. The general caching methods could reduce communication costs when these algorithms are performed in a distributed environment. We can further reduce the communications greatly by exploring the properties of graph connectivity problems. Our main idea is to apply the *percolation theory* [16] to the general caching schemes. The motivation of applying percolation theory is that both graph connectivity problems and percolation theory emphasize on the connected properties of a graph.

First, we define a typical percolation model to understand the percolation theory easily. Imagine a 2-dimensional grid of nodes, large enough so that the effects from its boundaries are negligible. Edges are drawn between neighbor nodes. Each edge can be open with probability p, or closed with probability $1 - p$. A cluster is defined as a group of nodes connected by open edges. Percolation theory analyzes the statistical and geometrical properties of these clusters as the probability p varies. We say that a cluster percolates the grid if it extends from one side of the grid to the opposite side. The size of such a cluster, termed the giant cluster, is proportional to the size of the grid. As p increases, the emergence of the giant cluster for the first time marks the *threshold point* where the grid becomes *connected* from one boundary to the opposite boundary. The value of p at this point is called the *threshold probability*, denoted by p_t.

Optimized Strategy. Given an instance $I = (B, S)$ of the LC problem, we obtain the cached edges E_I after the general caching scheme is applied to I. Given a threshold probability p_t of I, we pick out each edge $e \in E_I$ with probability p_t and obtain a new cached edge set E_c. If p_t

is small enough that $|E_c|$ will be much smaller than $|E_I|$. In other words, we can cache $|E_I| \cdot p$ edges to achieve the same effect as we cache $|E_I|$ edges for graph connectivity problems.

To utilize the caching capability, we intend to cache more edges to reduce the communication costs further. We take the following strategy: In the general solution, we set the cache size constraint to S/p_t. After the general solution, we select each edge with probability p_t and cache the selected edges for $I = (B, S)$.

Calculating Threshold Probability p_t. Below, we show how to compute the threshold probability for a big graph. The real-world graphs often follow the power-law distribution [21]. Thus, we compute the threshold probability for a power-law graph.

For a graph of size N, $N \to \infty$, with connectivity distribution $P(k)$, the percolation theory shows that its threshold probability is given by:

$$p_t = \frac{1}{\alpha - 1}, \tag{4}$$

where $\alpha = \frac{\langle k^2 \rangle}{\langle k \rangle}$ is the ratio of the second to first moment of the distribution of the actual graph.

Based on the calculation, we can obtain:

$$p_t = \frac{1}{\alpha - 1} = \frac{\mathbf{Li}_{\tau-1}(e^{-1/\nu})}{\mathbf{Li}_{\tau-2}(e^{-1/\nu}) - \mathbf{Li}_{\tau-1}(e^{-1/\nu})}. \tag{5}$$

where $\mathbf{Li}_\tau(x) = \sum_{k=1}^{\infty} \frac{x^k}{k^\tau}$ is the τ-th polylogarithm of x.

5 Online Phase of GCache

The last two sections show the offline phase which outputs clusters $C_i = (X_i \cup Y_i, E_i)$. In each cluster, we have $|Y_i| \leq S$, where S is the constraint of cache size. This section introduces the online phase, in which Y_is in the cache are replaced with other Y_is according to an eviction policy.

The online phase is based on the cluster-based processing principle: *nodes in the same cluster should be processed together.* Below we show the principle.

Online Phase of GCache. Given a meta graph $B(X \cup Y, E)$ on machine M, the offline phase outputs k clusters $C_i = (X_i \cup Y_i, E_i)$ for $1 \leq i \leq k$. In the node-centric computational model, a graph algorithm processes a C_i as an unit and the k clusters as an order, based on the cluster-based processing principle. Given a graph algorithm A_g, W.L.O.G, we can assume that the cluster order processed by A_g is $C_1, C_2, ..., C_k$.

While A_g is running, we first put $Y_1, Y_2, ..., Y_i$ in the cache such that $|Y_1| + |Y_2| + ... + |Y_i| \leq S$ and $|Y_1| + |Y_2| + ... + |Y_{i+1}| > S$. The cache is full at the moment and an eviction policy is invoked. The eviction replaces an Y_j ($j \leq i$) with Y_{i+1} such that the size constraint is still satisfied. The eviction policy can extend any existing replacement scheme (e.g., LRU and MRU) but treats a cluster as an unit. For example, the cluster-based LRU discards the least recently used Y_j first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards the least recently used Y_j. The least recently use can be measured by the total number of remote requests to the nodes in Y_j.

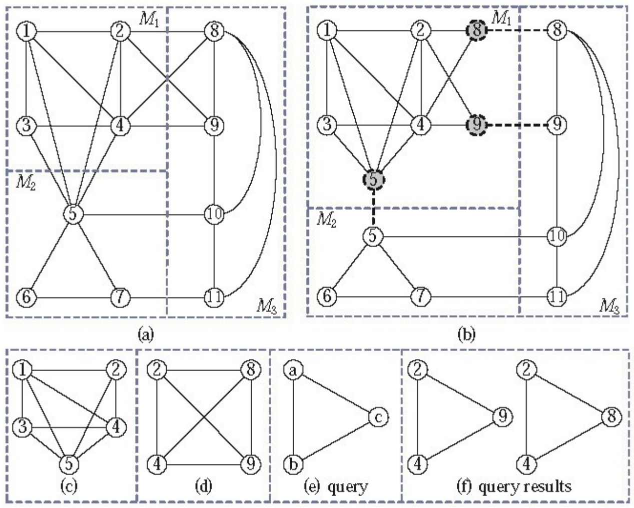


Fig. 5: Examples of the online processes for SSSP, CC, PR and SI: (a) the distributed graph G, (b) G with cached Y nodes, (c) the illustration for SSSP, CC and PR, and (d) the illustration for SI.

We use examples to show how the online phase is performed for typical graph algorithms. We consider four categories of graph algorithms: sequential traversal-based, parallel traversal-based, random walk-based, and localizable graph algorithms [22]. Table 2 shows typical algorithms in each category.

Example 3: We select one typical graph algorithm (i.e., SSSP, CC, PR, SI) in each category to show how the online phase is executed for such algorithm. Figure 5(a) shows a graph G distributed to three machines M_1, M_2 and M_3. We use M_1 to show the offline and online phases. In M_1, nodes $\{1, 2, 3, 4\}$ are local nodes, and nodes $\{5, 8, 9\}$ are remote nodes. In the offline phase, GCache outputs two clusters for M_1, $C_1 = X_1 \cup Y_1 = \{1, 2, 3, 4\} \cup \{5\}$ and $C_2 = X_2 \cup Y_2 = \{2, 4\} \cup \{8, 9\}$. In Figure 5(b), the dotted nodes denote the possible cached nodes. In the figure, if node 5 is cached, it has two copies in M_1 and M_2. Assume that the cache size constraint is 2 for M_1, i.e., the cache of M_1 maintains at most two nodes.

Online phase for SSSP. Assume that node 1 is the source, and the Dijkstra algorithm is performed. First, node 1 needs to request nodes $\{2, 3, 4, 5\}$ in which 5 is a remote node. GCache then caches node 5 and its current distance value (∞). Note that GCache only caches node 5, though it can cache two nodes. This is because the online phase caches clusters instead of nodes. Second, we continue computing SSSP for graph consisted of nodes $\{1, 2, 3, 4, 5\}$ given in Figure 5(c). The shortest-distance from node 1 to node 5 is 1. After that, the distance of 1 is sent to node 5 in M_2. Third, nodes $\{2,4\}$ need to request remote nodes $\{8, 9\}$. GCache then caches nodes $\{8, 9\}$ in replace of node 5. Fourth, we compute the SSSP for graph shown in Figure 5(d), and we obtain the shortest-distances 2 both

TABLE 2: Four categories of graph algorithms.

Category	Algorithms
Sequential traversal-based	single source shortest path (SSSP), breadth-first search (BFS), depth-first search (DFS), keyword search (KWS)
Parallel traversal-based	connected components (CC), label propagation (LP)
Random walk-based	pagerank (PR), simrank (SR)
Localizable algorithms	subgraph isomorphism (SI), Clique

to nodes 8 and 9. Finally, the distance of 2 is sent to M_3 in which the Dijkstra algorithm continued to be performed.

Online phase for connected component (CC). We use a standard sequential traversal (e.g., DFS) to compute the local connected components for M_1 and determine $v.id$ for each node v in M_1. We create a "root" node v_c carrying the minimum node id as $v_c.id$ (i.e., node 1), and link all the nodes in M_1 to v_c, and set their ids as $v_c.id$. These can be completed in one pass of the graph in M_1 via DFS. Therefore, we have the following steps. First, GCache caches node 5 and the node list (from M_2) with the same id as that of node 5, since node 5 is first requested by DFS. Second, we compute CC for graph shown in Figure 5(c). Note that all nodes in this graph have their ids as 1, since the graph is connected. Then the node list with id 1 is sent to M_2. Third, GCache caches nodes $\{8, 9\}$ in replace of node 5. Fourth, we compute CC for graph in Figure 5(d), and we also obtain the node ids as 1. Finally, the nodes with id 1 are sent to M_3.

Online phase for pagerank (PR). Every node v in M_1 is assigned to an initial value, which is its initial rank. We compute a new rank for node v by combining its previous rank and the aggregation of the ranks of its neighbors. We then send the new rank of v to its neighbors, and set the new rank of v to be held for the next iteration. The termination condition is based on the pre-determined number of supersteps, and all nodes will terminate at the same time. It is important to notice that all nodes are active sending their updated ranks to their neighbors, because such rank values have impacts on the ranks of their neighbors, and their neighbors neighbors, etc. Based on this process, the online phase for PR is similar to that for SSSP.

Online phase for subgraph isomorphism (SI). Figure 5(e) shows a query graph q and assume that the answers are two triangles given in Figure 5(f). First, the query q partially matches edge $(2, 4)$. Query q then probes remote node 5 in M_2 or remote nodes $\{8, 9\}$ in M_3 to find the third matched node. Second, GCache caches nodes $\{8, 9\}$ instead of node 5. This is because nodes $\{8, 9\}$ are two nodes which are more than one node. Third, query q matches the graph in Figure 5(d) and obtains two answers. Fourth, GCache caches node 5 in replace of nodes $\{8, 9\}$. Finally, query q cannot find any match over graph in Figure 5(a). The matching process terminates for M_1. □

6 Performance Evaluation

In this section, we report the effectiveness and efficiency test results of our newly proposed techniques. As mentioned before, We implement GCache on a node-centric computational system. We select Apache Giraph 1.1.0 [6] which is an open-source alternative to the proprietary Pregel. Our hardware environment is a cluster of 17 machines in a high-speed kilomega network, where one machine is selected as the master and the remaining machines are selected as computational nodes. Each machine has 2 Intel Xeon E5345 CPUs and 32GB memory, and is running CentOS Linux 5.6. The cache size constraint of each machine is set to 128M, 256M, 512M, 1G and 2G; the default value is 512M. All programs are coded in Java.

We evaluate our proposed scheme, GCache, in comparison with 3 the state-of-the-art algorithms, using 8 large graphs.

Implementation of GCache on Giraph. The implementation of the online phase of GCache is straightforward on Giraph. Here, we introduce how to implement the offline phase of GCache over Giraph. First, each machine M fetches all its remote neighbors in its memory to construct its meta graph $B(V, E)$. Second, we implement each step of Algorithm 1 entirely in M. Consequently, every machine can perform Algorithm 1 in parallel without any communication. The parameters used in the halting condition are set as follows: $\epsilon = 0.001$ and $\gamma = 5$.

Programming Model. Note that Giraph only sends messages from a machine to another machine. However, a machine may request values from another machine, such as computing CC and SI. To copy with this issue, we implement the request-respond functionality in Giraph as follows:

The request-respond paradigm supports all the functionality of Pregel. In addition, it supplements the node-to-node message channel with a request-respond message channel. In a superstep, a node v in machine M_i may call request(u) in its compute() function to request to node u in machine M_j for its attribute value $a(u)$ (which will be used in the next superstep). After compute() is called for all active nodes, the node-to-node messages are first exchanged. Then, machine M_i sends the request set to machine M_j. Finally, M_j collects the requests and sends a response set to M_i.

TABLE 3: Four categories of graph datasets.

Category	Graph datasets ($\|V\|$, $\|E\|$)
Social network	LiveJournal (LJ) (4.84M 68.47M), Google+ (28.94M 462.99M)
Web graph	Wikilink (Wiki)(11.19M 340.24M), Pld-arc (42.88M 623.05M)
Random graph (RG)	RG1 (50M, 2B), RG2 (100M, 3.5B)
Power-law graph (PG)	PG1 (100M, 1B), PG2 (200M, 4B)

Experimental setting.

(1) Datasets: The eight datasets (including 4 real and 4 synthetic graphs) we tested are with at least 4 million nodes and 60 million edges as shown in Table 3. Here, LiveJournal (LJ)[1] and Google+ (GO+)[2] are online social

1. http://snap.stanford.edu/
2. www.cs.berkeley.edu/ stevgong/dataset.html

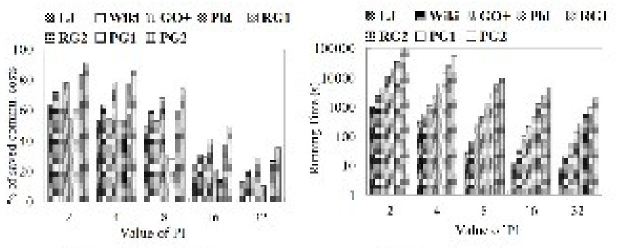
networks. Wikilink (Wiki)[3] and Pld-arc (Pld)[4] are two large Web graphs. Wikilink is the hyperlink graph inside the English Wikipedia, and Pld-arc is a hyperlink graph crawled in 2012. We use the R-MAT [23] to generate two random graphs RG1 and RG2, and two power-law graphs PG1 and PG2. Each graph is uniformly distributed to the 16 machines [18].

(2) Compared Algorithms: To compare with GCache, we adapt the CPU-based caching algorithms to a distributed scenario. Recall that the CPU-based caching algorithms predefine a total order for all the nodes of a graph. In the adaption, we order the X nodes of the bipartite graph $H(X,Y)$ as the CPU-based caching algorithms in the offline phase. During the online phase, we cache the related Y nodes as the order of X nodes and then replace the cached nodes as the LRU or MRU strategy. The difference between this adapted scheme and GCache is as follows. GCache processes a cluster as an unit in ether offline or online phase, whereas the adapted scheme treats all the X nodes as an unit in the offline phase and each Y node as an unit in the online phase. Note that the default eviction policy is the LRU in the following evaluations.

Specifically, we implement two state-of-the-art CPU-based caching algorithms, MINLA [11], Gorder [10] and a partitioning algorithm METIS [19], in terms of caching effectiveness. MINLA is the node ordering π for the Minimum Linear Arrangement problem, which is to minimize $\sum_{(u,v)\in E(G)} \parallel \pi(u) - \pi(v) \parallel$. Gorder is also a graph ordering, which is to find the optimal permutation ϕ among all nodes in a given graph G by keeping nodes that will be frequently accessed together locally in a window of size w. We have tried MINLA and Gorder for the distributed graph, but neither of them can be scalable to deal with large graphs effectively. Instead, we use the simulated annealing technique to compute the result, which has good scalability and shows comparable performance. METIS [19] is a widely-used graph partition algorithm which divides graph into partitions to minimize edge-cut which is the number of edges that cross different partitions. METIS cannot be applied to our problem, since it does not satisfy the size constraint. To remedy this, we plug METIS into Algorithm 2 to take place of RANDOM, and then select the optimal partition satisfying the size constraint. The optimal partition can be applied to our problem.

The caching effectiveness is measured by the *percentage of saved communication cost* which equals to $\frac{C_n - C_c}{C_n}$ (also referred to as hit ratio), where C_n and C_c are the communication costs of a graph algorithm without and with the caching strategy. The caching efficiency is measured by the *processing time* of a graph algorithm after the caching scheme is applied.

Note that we perform the local graph clustering algorithm for every machine of the 16 machines in parallel. The processing time and communication costs include the total time and communication costs over all the 16 machines.



(a) % of Saved Costs (b) Running Time

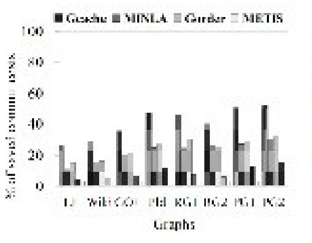
Fig. 6: Determine the value of π in Algorithm 2.

We use the number of communication messages to measure communication costs. The total messages include messages between the master and the slaves, as well as the messages between the slaves.

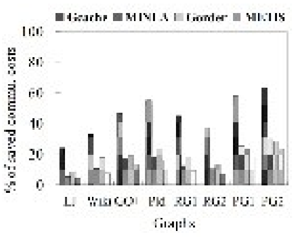
(3) Applied Algorithms: We apply 10 graph algorithms to test the caching mechanism. The 10 graph algorithms are classified into four categories, i.e., sequential traversal-based, parallel traversal-based, random walk-based, and localizable graph algorithms, which are listed in Table 2. Specifically, sequential traversal-based algorithms contain single source shortest path (SSSP) by the Dijkstra algorithm [20], breadth-first search (BFS) [20], depth-first search (DFS) [20], and keyword search (KWS) by the BANKS algorithm [24]. Parallel traversal-based algorithms contain connected component (CC) detection [25] and label propagation [26]. Random walk-based algorithms contain pagerank (PR) [27] and simrank (SR) [28]. Localizable graph algorithms contain subgraph isomorphism (SI) by the VF2 algorithm [29] and Clique [30]. All of them are fundamental for graph computing and provide the basis for the design of other advanced graph algorithms. For the accuracy of the results, except for PR, we repeat these algorithms 10 times when running the experiments and report the tested measures. PageRank algorithm is stopped after 50 iterations in every dataset. These graph algorithms are naturally parallelized over Giraph, i.e., implementing the algorithms as the node-centric computing mode.

(4) Determine Parameters: To perform GCache, we should determine the parameter π in Algorithm 2. The value of π could control the balance of the efficiency of GCache and the effectiveness of GCache. With five different π, Figure 6(a) reports the percentages of the saved communication costs of GCache on the 8 graphs for the PageRank algorithm. Figure 6(b) gives the running time of GCache on the 8 graphs with respect to five different π. The values of π are set to 2, 4, 8, 16 and 32 respectively. As expected, the saved communication costs become fewer as π increases, whereas GCache becomes more efficient as π grows. We observe: when π is from 2 to 8, the saved communications decrease slowly, and when π is from 8 to 32, the saved communications decrease fast; when π is from 2 to 8, the running time decreases slowly, and when π is from 8 to 32, the running time decreases fast. Thus, when π reaches 8, GCache has the best balance between efficiency and effectiveness. In the following experiments, we set π to 8.

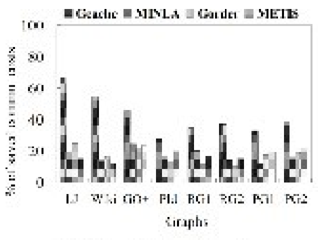
3. http://konect.uni-koblenz.de/
4. webdatacommons.org/hyperlinkgraph/



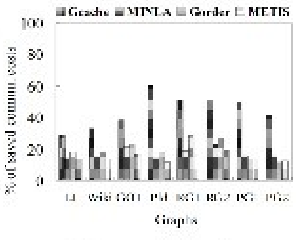
(a) *SSSP*



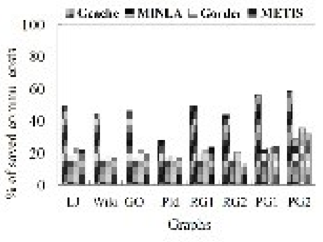
(b) Breadth-First Search



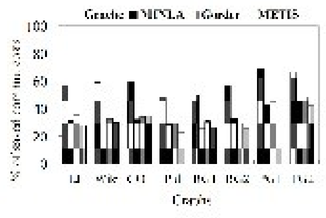
(c) Depth-First Search



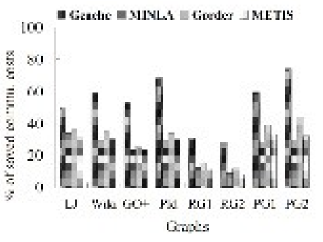
(d) Keyword Search



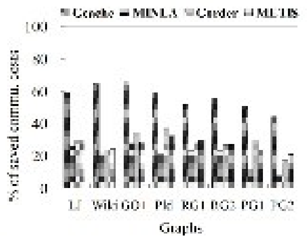
(e) Connected Component



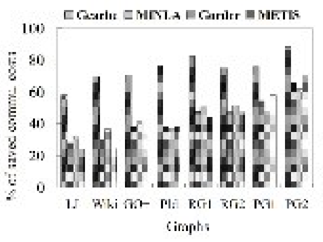
(f) Label Propagation



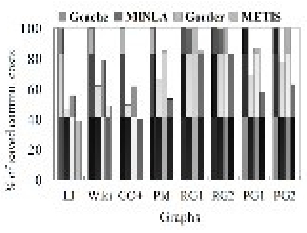
(g) PageRank



(h) *SimRank*



(i) Subgraph Isomorphism



(j) Clique

Fig. 7: Percentage of the saved communication costs of caching schemes on the 10 graph algorithms.

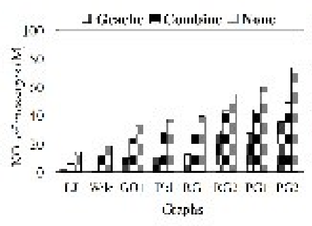
Experimental results.

Exp-1: Figure 7 demonstrates the percentages of the saved communication costs of GCache, MINLA, Gorder and METIS on 8 graphs for 10 graph algorithms. There are in total 80 (8×10) pairs of graph algorithms and graph datasets in the experiment. The ten graph algorithms are performed online. We feed LRU into the first five algorithms and MRU into the last five algorithms. These figures are classified into four categories as those for graph algorithms. Specifically, Figures 7(a)-7(d) show the results on the sequential traversal-based algorithms; Figures 7(e) and 7(f) show the results on the parallel traversal-based algorithms; Figures 7(g) and 7(h) show the results on the random walk-based algorithms; Figures 7(i) and 7(j) show the results on the localizable graph algorithms.

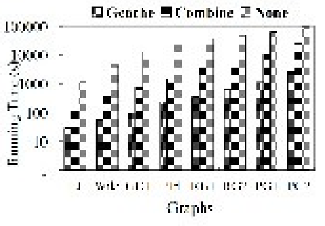
As shown in these figures, GCache saves most communication costs compared to MINLA and Gorder over 80 cases, in every testing. For example, in Figure 7(d), the average saved communication costs of GCache are 5.8, 5.2 and 4.6 times as large as those of METIS, MINLA and Gorder. Besides, we have the following observations. (a) In general, the saved traffic costs by GCache become more significant as the graph size increases. Compared with Flickr and LiveJournal, the other 6 datasets are much larger with at least 30 million nodes and 400 million edges. Given the relative small cache size, only limited graph information can be located within the cache and hence the effect of the cache usage becomes more significant for large graphs. But exceptions exist for some graph algorithms, e.g., DFS. (b) GCache shows similar communication behaviours within the same category and different communication behaviours on different categories. For instance, Clique and ISO are localizable algorithms and hence one-hop neighbor caching saves most traffic costs needed to perform clique and ISO. SSSP and KWS are BFS-like graph algorithms, and so GCache behaves similar performance tendency for BFS, SSSP and KWS algorithms. (c) GCache behaves the best performance for both LRU and MRU policies, which demonstrates that the cluster-based offline and online algorithms are suitable for any existing eviction policy.

Exp-2: In this experiment, we compare GCache with a state-of-the-art algorithm [31] (denoted by Combine) that replicates high degree nodes and combines messages during communications. We also compare with the scheme without any optimization, denoted by None. We select one typical graph algorithm from each category, i.e., SSSP, CC, PR and SI. Figure 8 reports the actual running time and communication costs of GCache, Combine and None on each typical graph algorithm. From the four results, we observe: (a) GCache is the fastest and lightest followed by Combine and None. This is because GCache caches any possible bottleneck node and combine any expensive messages during communications, whereas Combine only replicates high degree nodes and packets partial messages. (b) The benefit obtained by GCache is obvious, though it needs an offline process that clusters nodes. For example, GCache is two orders faster than None on all 4 typical algorithms and 8 datasets. However, GCache needs only one offline process and can support many online graph algorithms. In addition, as shown in Figure 10, the running time of the offline process is much smaller than those of some online graph algorithms.

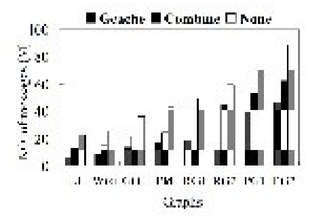
Exp-3: Figure 9 plots the percentages of the saved communication costs of GCache on 8 graphs with respect to various cache sizes. Four typical graph algorithms (BFS, SI,



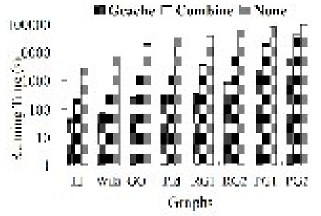
(a) Communication costs of *SSSP*



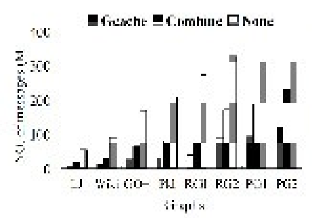
(b) Running time of *SSSP*



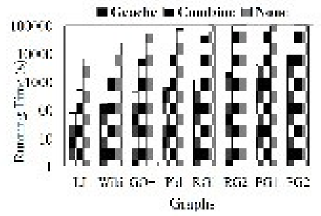
(c) Communication costs of CC



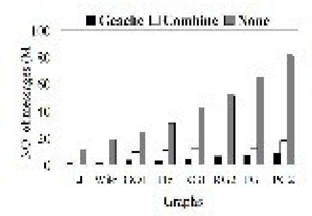
(d) Running time of CC



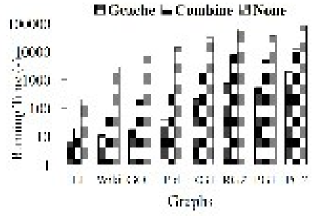
(e) Communication costs of PR



(f) Running time of PR

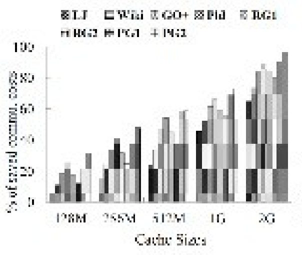


(g) Communication costs of SI

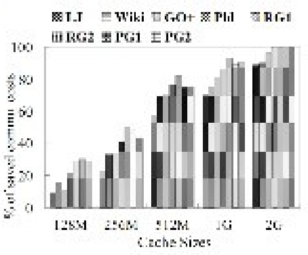


(h) Running time of SI

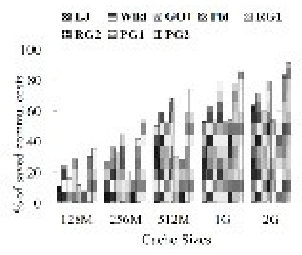
Fig. 8: Communication costs and running time of graph algorithms after the caching schemes are applied.



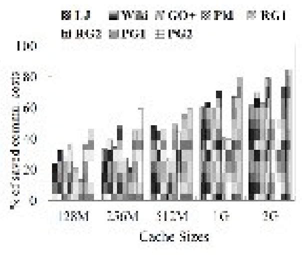
(a) Breadth-First Search



(b) Subgraph Isomorphism

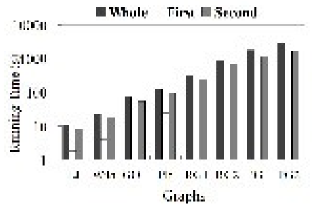


(c) PageRank

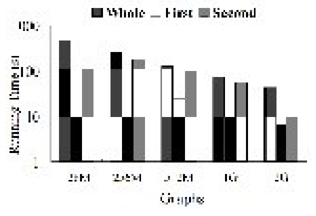


(d) Connected Component

Fig. 9: Percentage of the saved communication costs of GCache on various cache sizes.

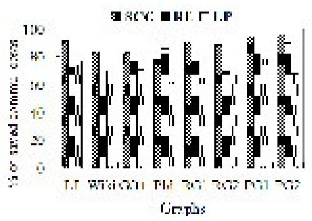


(a) Various Datasets

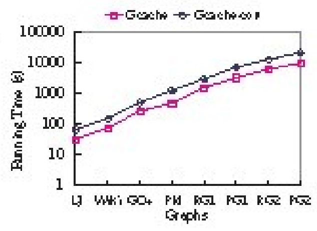


(b) Various Cache Sizes

Fig. 10: Running time of GCache on various datasets and cache sizes.



(a) % of Saved Costs



(b) Running Time

Fig. 11: Running time and % of saved communication costs of optimization for graph connectivity algorithms.

PR and CC) are selected in the testing since they behave various algorithmic properties. As expected, GCache saves more communication costs as the cache size increases. As shown in Figure 9(b), SI hardly incurs traffic costs at all 8 datasets, when the cache size reaches 2G. The reason is that SI is a localizable algorithm. All graph algorithms save fewer communication costs on RG1 and RG2, since the random graph is denser and the algorithms converge fast.

Exp-4: In this experiment, we report the running time of the offline phase on different datasets and cache sizes. Recall that the offline phase consists of two steps, i.e., a random partitioning and a relabeling step. We use Whole to denote the whole process; First to denote the first step; Second to denote the second step. Figure 10 reports results from which we know the following. (a) All the bars increase as the datasets become larger. All the bars decrease as the cache size increases. This is because a large cache size leads to a small number of clusters. (b) Whole is efficient, e.g. less then 1000s on the graph with 100M nodes. Second dominates the whole costs, since First employs a random partitioning which is very efficient compared to Second.

Exp-5: In this experiment, we examine the optimization mechanisms of GCache for graph connectivity algorithms. We refer to the optimization mechanism as GCache-con. The used graph connectivity algorithms include connected components (CC) [25], reachability (RE) [32] and label propagation (LP) [26]. Figure 11(a) shows the percentages of the saved communication costs of GCache-con on

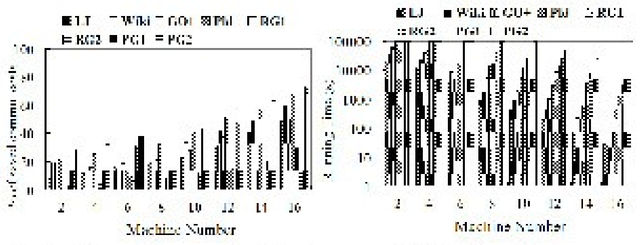

(a) % of Saved Costs on PageRank (b) Running Time

Fig. 12: Performance of GCache on various machine numbers.

CC, RE and LP with respect to 8 datasets. As shown in the figure, GCache-con saves more communication costs based on the optimization. Compared to GCache, the saved percentage of GCache-con increases by 45% on average. This is because GCache-con only needs cache a very small portion of edges to guarantee the connectivity. GCache-con saves more traffics than GCache at the cost of longer time to perform the optimization mechanism. However the cost is not large as reported in Figure 11(b), from which we know that GCache-con runs less twice as much as GCache.

Exp-6: Finally, we examine the scalability of GCache with respect to machine number. Figure 12(a) reports the percentages of the saved communication costs of GCache by running the PageRank on 8 datasets. The percentage grows with the rising of machine number, owing to that more machines incur lager network overhead and certainly leads to a better caching effect. Figure 12(b) plots the processing time of GCache with respect to machine number. Generally, the running time is reduced by adding more machines due to a higher parallelism. However, the decrease of time cost is not completely proportional to the increase of machine number. The main reason is that the clustering tasks are not balanced on all machines. This examination concludes that the effectiveness and efficiency of our caching algorithm, GCache, are well scalable to the number of computational machines.

7 Related Work

We review pervious works on the distributed graph processing and graph caching methods.

Offline Graph Caching. Current works of offline graph caching concentrate upon the CPU-based caching algorithms which have been studied in recent years [10], [11], [12], [13], [14], [33]. There are works on specific graph algorithms to improve the CPU cache performance. Park et al. propose several schemes to optimize the cache performance of Prim, Floyd-Warshall, and Bipartite matching algorithms in [33]. However, these strategies are designed for some specific graph algorithms, and can not be used to support any graph algorithms in general. Some existing works improve the ability of graph caching by graph ordering and node clustering. Auroux et al. propose to reorder the node IDs by BFS [12]. Banerjee et al. propose a node ordering method by the depth-first traversal method to improve the efficiency of the graph exploring [13]. Zhang et al. [14] improve the cache utilization for graph analytics by breaking the vertices into segments that fit in last level cache. The techniques of graph ordering proposed in [11] and [10] are to find the optimal permutation among all nodes in a given graph G by keeping nodes that will be frequently accessed together locally. Though the approaches of graph ordering can handel many graph algorithms, their distributed versions have lower caching capability than our proposed algorithm as shown in the experiments.

Online Graph Caching. There are works designed for specific graph queries, e.g., [34], [35] for subgraph/supergraph queries and [36], [37], [38] for SPARQL queries. Their methods are not universal like GCache for any graph algorithm. Other typical online graph caching includes the works such as [7], [8] and [9]. The authors of [7] and [8] advocate the traditional replacement strategy to improve the cache hit ratio, such as LRU and MRU. The work [9] not only combines LRU or MRU into its algorithm, but also considers graph partition-aware strategies. The common principle of these online works is to discard the nodes that was not needed for the longest time. Hence, their adopted caching schemes tend to use the previous access patterns to predict whether a data item will be accessed in the future. This fashion makes replacement decision according to the most recently accessed information. GCache extends the LRU and MRU schemes in the online phase on the basis of the offline phase. Therefore, GCache is more effective than the mere online algorithm.

Graph Clustering. The goal of graph clustering, also called community detection or graph partitioning, is to identify clusters of densely linked nodes given only the graph itself [39]. The vast majority of algorithms optimize a function that captures the edge density of the cluster (a set of nodes), for instance, conductance or modularity. There have been extensive works to study graph clustering and there are many classical works, e.g., METIS [19]. A detailed introduction to the algorithms of graph clustering can be found in a survey [40]. The technique of label propagation (LP) has been applied to community detection and graph graph partitioning [17], [26]. However, the algorithms in these works cannot directly solve our model, since there are several differences between our model and the general partitioning problem. In our model, we consider the maximum size restriction and the unfixed partitioning number that are not defined for the general partitioning problem. Even though we adapt METIS to our problem, as shown in the experiment, our algorithm GCache needs 240 seconds to process the GO+ graph, whereas METIS takes more than 100,000 seconds (=27 hours) at the same graph.

Parallel and Distributed Graph Processing. Several parallel models have been studied for graphs, e.g., PRAM [41], BSP and MapReduce [42]. [43] proposes a hypergraph partitioning to model communication volume. However, this model only suits to parallel sparse matrix vector multiplication and is not designed for general graph applications.

Due to the challenges posed by parallel graph processing, PRAM and MapReduce are not suitable for large graph processing as discussed in [4]. BSP is widely regarded as the most popular model to handle large graphs. Pregel [4] implements BSP with node-centric programming, where a superstep executes a user-defined function at each node in parallel. Popular Pregel-like systems also include GraphX [5], GRACE [18], GPS [15], etc. GRACE [18] provides an operator-level, iterative programming model to enhance synchronous BSP with asynchronous execution. GraphX [5] recasts graph computation in its distributed dataflow framework as a sequence of join and group-by stages punctuated by map operations over Spark platform. GPS [15] implements Pregel with extended APIs and partition strategies. Mizan [44], a middleware for Pregel, between the users' code and the computing infrastructure can reduce graph computations and communications. [45] optimized the Pregel-like systems by transferring computation loads from heavy machines to light machines. [46] shows the algorithmic behaviours of parallel graph analytics. A variety of parallel algorithms exist for the graph problems such as SCC [47], minimum spanning tree [48], maximum matching [49], and graph coloring [50]. These algorithms are designed for the PRAM model or are implemented in the Pregel-like systems. Our proposed caching scheme, GCache, can be seamlessly integrated into the above graph systems and can speed up the above parallel graph algorithms.

8 Conclusion

This paper studies distributed graph caching to reduce communication costs of graph algorithms. Different from traditional online and offline caching algorithms, we propose a hybrid caching scheme (GCache) consisted of an offline and online phase. We model the offline phase by the local clustering on a bipartite graph. We employ a greedy labeling algorithm to solve the local clustering. We also integrate a maximum-cost maximum-flow model in the local clustering to solve the problem of the cache size constraint. Moreover we optimize the offline phase for graph connectivity problems. In the online phase, we cache and process a cluster as an unit by extending traditional eviction policies. Extensive experimental results demonstrate that our methods can scale to web-scale graphs for basic graph algorithms. In the future, we will study an incremental caching mechanism for distributed graphs in the dynamic world.

Acknowledgments

Ye Yuan is supported by the NSFC (Grant Nos. 61572119 and 61622202) and the Fundamental Research Funds for the Central Universities (Grant No. N181605012). Lian Xiang is supported by the NSF OAC No. 1739491 and Lian Start Up No. 220981, Kent State University. Guoren Wang is supported by the NSFC (Grant Nos. 61672145 and 61732003). Lei Chen is supported by Hong Kong RGC GRF Project 16214716, the NSFC (Grant No. 61729201) and Hong Kong ITC ITF grants ITS/391/15FX. Jeffrey Xu Yu is supported by the Research Grants Council of Hong Kong, China under No. 14203618 and No. 14221716.

References

[1] S. Muthukrishnan, "One trillion edges: graph processing at facebook-scale," *PVLDB*, vol. 8, no. 12, pp. 1804–1815, 2015.

[2] http://www.worldwidewebsize.com/.

[3] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs." *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–146.

[5] 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 *OSDI*, 2014, pp. 599–613.

[6] C. Martella, R. Shaposhnik, D. Logothetis, and S. Harenberg, *Practical graph analytics with apache giraph*. Springer, 2015.

[7] D. Yan, H. Chen, J. Cheng, M. T. Özsu, Q. Zhang, and J. Lui, "G-thinker: big graph mining made easier and faster," *arXiv preprint arXiv:1709.03110*, 2017.

[8] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, and H. Li, "Tao: Facebook's distributed data store for the social graph," in *Usenix Conference on Technical Conference*, 2013, pp. 49–60.

[9] H. Aksu, M. Canim, Y. Chang, I. Korpeoglu, and Ö. Ulusoy, "Graph aware caching policy for distributed graph stores," in *2015 IEEE International Conference on Cloud Engineering, IC2E 2015, Tempe, AZ, USA, March 9-13, 2015*, 2015, pp. 6–15.

[10] H. Wei, J. X. Yu, C. Lu, and X. Lin, "Speedup graph processing by graph ordering," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1813–1828.

[11] I. Safro, D. Ron, and A. Brandt, "Multilevel algorithms for linear ordering problems," *Journal of Experimental Algorithmics (JEA)*, vol. 13, pp. 4:1.4–4:1.20, 2009.

[12] L. Auroux, M. Burelle, and R. Erra, "Reordering very large graphs for fun & profit," in *International Symposium on Web AlGorithms*, 2015.

[13] J. Banerjee, W. Kim, S.-J. Kim, and J. F. Garza, "Clustering a dag for cad databases," *TSE*, vol. 14, no. 11, pp. 1684–1699, 1988.

[14] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 293–302.

[15] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the SSDM*, 2013, p. 22.

[16] D. Stauffer and A. Aharony, *Introduction to percolation theory*. Taylor and Francis, 2014.

[17] M. J. Barber and J. W. Clark, "Detecting network communities by propagating labels under constraints," *Physical Review E*, vol. 80, no. 2, p. 026129, 2009.

[18] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *KDD*. ACM, 2012, pp. 1222–1230.

[19] G. Karypis and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed computing*, vol. 48, no. 1, pp. 96–129, 1998.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2001.

[21] G. A. Mendes and L. R. D. Silva, "Generating more realistic complex networks from power-law distribution of fitness," *Brazilian Journal of Physics*, vol. 39, no. 2, pp. 423–427, 2009.

[22] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *PVLDB*, vol. 7, no. 12, pp. 1047–1058, 2014.

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

[24] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using banks," in *Proc. of ICDE*, 2002, pp. 431–440.

[25] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers & Mathematics with Applications*, vol. 7, no. 1, pp. 67–72, 1981.

[26] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proceedings of the WSDM*, 2013, pp. 507–516.

[27] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[28] G. Jeh and J. Widom, "Simrank: a measure of structural-context similarity," in *Proceedings of KDD*, 2002, pp. 538–543.

[29] L. P. Cordellaand, P. Foggia, and C. Sansone, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 10, pp. 1367–1372, 2004.

[30] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theoretical Computer Science*, vol. 363, no. 1, pp. 28–42, 2006.

[31] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 1307–1317.

[32] W. Fan, X. Wang, and Y. Wu, "Performance guarantees for distributed reachability queries," in *VLDB*, 2012, pp. 1304–1316.

[33] J.-S. Park, M. Penner, and V. K. Prasanna, "Optimizing graph algorithms for improved cache performance," *TPDS*, vol. 15, no. 9, pp. 769–782, 2004.

[34] J. Wang, N. Ntarmos, and P. Triantafillou, "Graphcache: A caching system for graph queries," in *Proceedings of the 20th International Conference on Extending Database Technology (EDBT)*, 2017, pp. 13–24.

[35] J. Wang, Z. Liu, S. Ma, N. Ntarmos, and P. Triantafillou, "Gc: a graph caching system for subgraph/supergraph queries," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 2022–2025, 2018.

[36] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris, "Graph-aware, workload-adaptive sparql query caching," in *Proceedings of SIGMOD*, 2015, pp. 1777–1792.

[37] M. Martin, J. Unbehauen, and S. Auer, "Improving the performance of semantic web applications with sparql query caching," in *Extended Semantic Web Conference*, 2010, pp. 304–318.

[38] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[39] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3, pp. 75–174, 2010.

[40] S. E. Schaeffer, "Graph clustering," *Computer science review*, vol. 1, no. 1, pp. 27–64, 2007.

[41] L. G. Valiant, *General purpose parallel architectures*. MIT press, 1991.

[42] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[43] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *IPDPS*. IEEE, 2006, pp. 10–pp.

[44] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, W. Dan, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *ACM European Conference on Computer Systems*, 2013.

[45] Z. Shang and J. X. Yu, "Catch the wind: Graph workload balancing on cloud," in *29th IEEE International Conference on Data Engineering (ICDE)*, 2013, pp. 553–564.

[46] A. Lenharth, D. Nguyen, and K. Pingali, "Parallel graph analytics." *Commun. ACM*, vol. 59, no. 5, pp. 78–87, 2016.

[47] L. K. Fleischer, B. Hendrickson, and A. Pinar, "On identifying strongly connected components in parallel," *Lecture Notes in Computer Science*, vol. 1800, no. 4, pp. 505–511, 2000.

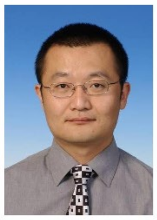
[48] F. Dehne and S. Gotz, "Practical parallel algorithms for minimum spanning trees," in *Proceedings. Seventeenth IEEE Symposium on Reliable Distributed Systems*, 1998, pp. 366–371.

[49] F. Manne and R. H. Bisseling, "A parallel approximation algorithm for the weighted maximum matching problem," in *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007, Revised Selected Papers*, 2007, pp. 708–717.

[50] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency & Computation Practice & Experience*, vol. 12, no. 12, pp. 1131–1146, 2015.

Ye Yuan received the BS, MS, and PhD degrees in computer science from Northeastern University, in 2004, 2007, and 2011, respectively. He is now a professor in the Department of Computer Science, Beijing Institute of Technology, China. His research interests include graph databases, probabilistic databases, and social network analysis.

Xiang Lian received the BS degree from the Department of Computer Science and Technology, Nanjing University, in 2003, and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong. He is now an assistant professor in the Department of Computer Science, Kent University. His research interests include data management, and streaming time series.



Lei Chen received the bachelors degree in computer science from Tianjin University, China, in 1994, the masters degree in computer science from the Asian Institute of Technology, in 1997, and the PhD degree in computer science from the University of Waterloo, Canada. He is currently a professor of computing science with the Hong Kong University of Science and Technology, China. His research interests include multimedia databases, and probabilistic databases.

Guoren Wang received his B.S., M.S. and Ph.D. degrees in computer science from Northeastern University, Shenyang, in 1988, 1991 and 1996, respectively. Currently, he is a professor in the School of Computer Science and Technology, Beijing Institute of Technology, Beijing. His research interests are query processing and optimization, high-dimensional indexing, parallel database systems, and P2P data management.

Jeffrey Xu Yu received the BE, ME, and PhD degrees in computer science from the University of Tsukuba, Japan, in 1985, 1987, and 1990, respectively. Currently, he is a professor in the Department of Systems Engineering and Engineering Management, Chinese University of Hong Kong. His research interests include graph database and query processing, graph mining, social networks, big data, and cloud computing.

Yishu Wang received B.S. in Software Engineering from the Northeastern University in 2015, then received M.S. in Computer Software and Theory in 2017. Currently, she is a Ph.D. candidate of Computer Science and Engineering College of Northeastern University. Her research interests include graph databases, uncertain graph data management, temporal graph.

Yuliang Ma received his B.S. degree in computer science from the College of Computer Science and Engineering, Northeastern University, Shenyang, in 2013. Currently, he is a Ph.D. candidate of Northeastern University, Shenyang. His main research interests include graph databases, location-based social networks, and social network analysis.