On Achieving Efficient Data Transfer for Graph Processing in Geo-Distributed Datacenters

Amelie Chi Zhou, Shadi Ibrahim
Inria Rennes - Bretagne Atlantique Research Center, France

Bingsheng He
National University of Singapore, Singapore

***Abstract*—Graph partitioning is important for optimizing the performance and communication cost of large graph processing jobs. Recently, many graph applications such as social networks store their data on geo-distributed datacenters (DCs) to provide services worldwide with low latency. This raises new challenges to existing graph partitioning methods, due to the costly Wide Area Network (WAN) usage and the multi-levels of network heterogeneities in geo-distributed DCs. In this paper, we propose a geo-aware graph partitioning method named *G-Cut*, which aims at minimizing the inter-DC data transfer time of graph processing jobs in geo-distributed DCs while satisfying the WAN usage budget. G-Cut adopts two novel optimization phases which address the two challenges in WAN usage and network heterogeneities separately. G-Cut can be also applied to partition dynamic graphs thanks to its light-weight runtime overhead. We evaluate the effectiveness and efficiency of G-Cut using real-world graphs with both real geo-distributed DCs and simulations. Evaluation results show that G-Cut can reduce the inter-DC data transfer time by up to 58% and reduce the WAN usage by up to 70% compared to state-of-the-art graph partitioning methods with a low runtime overhead.**

***Keywords*-Graph partitioning; Heterogeneous network; Geo-distributed datacenters**

I. Introduction

Graph processing is an emerging computation model for a wide range of applications, such as social network analysis [1], [2], natural language processing [3] and web information retrieval [4]. Graph processing systems such as Pregel [5], GraphLab [6] and PowerGraph [7] follow the "think-as-a-vertex" philosophy and encode graph computation as vertex programs which run in parallel and communicate through edges of the graphs. Vertices iteratively update their states according to the messages received from neighboring vertices. Thus, efficient communication between vertices and their neighbors is important to improving the performance of graph processing algorithms. Graph partitioning plays a vital role in reducing the data communication cost and ensuring load balance of graph processing jobs [7], [8].

Many graph applications, such as social networks, involve large sets of data spread in multiple geographically distributed (geo-distributed) datacenters (DCs). For example, Facebook receives terabytes of text, image and video data everyday from users around the world [9]. In order to provide reliable and low-latency services to the users, Facebook has built four geo-distributed DCs to maintain and manage those data. Also, it is sometimes impossible to move data out of their DC due to privacy and government regulation reasons [10]. Therefore, it is inevitable to process those data in a geo-distributed way. We identify a number of technical challenges for partitioning and processing graph data across geo-distributed DCs.

First, the data communication between graph partitions in the geo-distributed DCs goes through the Wide Area Network (WAN), which is usually much more expensive than intra-DC data communication [11]. Most existing cloud providers, such as Amazon EC2, charge higher prices on inter-region network traffic than on intra-region network traffic [12]. Traditional graph partitioning methods which try to balance the workload among different partitions while reducing the vertex replication rate [7] may end up with large inter-DC data transfer size and hence large WAN cost. Another extreme example is to replicate the entire graph data in every DC in advance, which although greatly reduces the inter-DC data transfer size during graph processing, causes huge WAN cost for data replications. Hence, a more efficient approach to reduce the WAN usage cost during graph partitioning in geo-distributed DCs is needed.

Second, the geo-distributed DCs have highly heterogeneous network bandwidths on multiple levels. On the one hand, the uplink and downlink bandwidths of a DC can be highly heterogeneous due to the different link capacities and resource sharing among multiple applications [11]. We have found more than four times difference between the uplink and downlink bandwidths of the cc2.8xlarge Amazon EC2 instances. On the other hand, the network bandwidths of the same type of link in different DCs can also be heterogeneous due to different hardwares and workload patterns in the DCs. For example, it has been observed that the network bandwidth of the EU region is both faster and more stable than the network bandwidth of the US region in Amazon EC2 [13]. What makes it worse is that many graphs have heterogeneous traffic patterns in different vertices due to the power-law distribution of vertex degrees [7], [8]. Thus, even if the amount of data transfered across DCs is minimized, it can still result in long inter-DC data transfer time if not considering the multiple levels of network heterogeneities.

To address the above challenges, we propose a geo-aware graph partitioning method named *G-Cut* for geo-distributed DCs. Compared to other resources such as CPU and memory, WAN bandwidth is more scarce in the geo-distributed environment. Thus, *our goal in this paper is to optimize the performance of graph processing jobs by minimizing the inter-DC data transfer time while satisfying user-defined WAN usage*

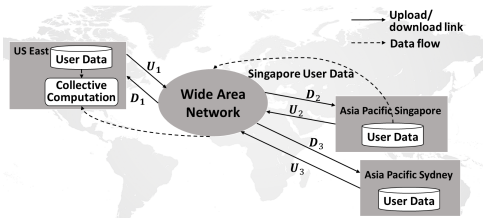

Fig. 1: Data communication in geo-distributed datacenters.

budget constraint. However, considering the heterogeneities in graph traffic and network bandwidths, and given the large sizes of many geo-distributed graphs, obtaining a good partitioning result is non-trivial. Thus, G-Cut adopts two optimization phases which address the two challenges in WAN usage and network heterogeneities separately. In the first phase, we propose a streaming heuristic which aims at minimizing the inter-DC data transfer size (i.e., runtime WAN usage) and utilize the one-pass streaming partitioning method to quickly assign the edges onto different DCs. In the second phase, we propose two partition refinement heuristics which identify the network performance bottlenecks and refine the graph partitioning generated in the first phase to reduce the inter-DC data transfer time. Thanks to the two-phases optimization, G-Cut exhibits a light-weight runtime overhead and can be easily and efficiently extended to support the partitioning of dynamic graphs. We evaluate the effectiveness and efficiency of G-Cut with five real-world graphs and three different graph algorithms, and compare it with three state-of-the-art graph partitioning methods [7], [8]. Evaluation results show that G-Cut can reduce the inter-DC data transfer time by up to 58% and reduce the WAN usage by up to 70% compared to the other methods. Regarding the overhead, it takes less than 8 seconds for G-Cut to partition one million edges, which is comparable to existing methods.

The following of this paper is organized as below. Section II introduces the background and related work. We formulate the graph partitioning problem in Section III and introduce our proposed techniques in Section IV and V. We evaluate G-Cut in Section VI and conclude this paper in Section VII.

II. Background and Related Work

A. Geo-distributed Datacenters

Many cloud providers and large companies are deploying their services globally to guarantee low latency to users around the world. For example, Amazon EC2 currently has 14 geographically distributed service regions [15] and Google has tens of DCs distributed in four different regions [17].

Figure 1 uses Amazon EC2 as a case study to show the scenario of providing geo-distributed services. In this example, user data are stored and updated in local DCs to provide low-latency services to local users and are transfered through the WAN to other DCs for collective computations. U_i (D_i) stands for the uplink (downlink) bandwidth from a DC i to an endpoint of the WAN. We measure the uplink/downlink bandwidths from the US East, Asia Pacific Singapore and Sydney regions of Amazon EC2 to/from the WAN using cc2.8xlarge instances and have the following observations.

TABLE I: Uplink/downlink bandwidths of cc2.8xlarge instances from three Amazon EC2 regions to the Internet. Prices are for uploading data out of the regions to the Internet.

	US East	AP Singapore	AP Sydney
Uplink Bandwidth (GB/s)	0.52	0.55	0.48
Downlink Bandwidth (GB/s)	2.8	3.5	2.5
Price ($/GB)	0.09	0.12	0.14

Observation 1: the uplink/downlink bandwidths of a single DC can be heterogeneous. As shown in Table I, the downlink bandwidths of all the three regions are several times higher than their uplink bandwidths. This is mainly due to the different link capacities and resource sharing among multiple applications [11]. Our evaluation on Amazon EC2 shows that many instance types, such as c4.4xlarge, cc2.8xlarge and m4.10xlarge, show significant differences between their uplink and downlink bandwidths.

Observation 2: the bandwidths of different DCs are also heterogeneous. For example, the uplink and downlink bandwidths of the Singapore region are 17% and 40% higher than those of the Sydney region, respectively. This is mainly due to the differences of hardware and the amount of workloads between different cloud regions [13].

Observation 3: using WAN bandwidth is pricy. Using network bandwidth within a single DC is usually fast and cheap. However, it is not the case when using WAN in geo-distributed DCs. For example, data transfer within the same region of Amazon EC2 is usually free of charge, while sending data to the Internet can be very pricy and region-dependent as shown in Table I. This is because the providers of geo-distributed DCs have to rent WAN bandwidth from Internet Service Providers and pay accordingly for the WAN usage.

B. Graph Processing Systems

Many graph processing systems, such as Gemini [18], Pregel [5], GraphLab [6] and PowerGraph [7], follow the "think-as-a-vertex" philosophy and provide user-friendly vertex-centric abstractions for users to efficiently implement graph processing algorithms.

In this paper, we consider the widely-used GAS graph processing model proposed in PowerGraph [7], which iteratively executes user-defined vertex computations until convergence. It is our future work to extend this study to other graph processing models. There are three computation stages in each GAS iteration, namely **G**ather (Sum), **A**pply and **S**catter. In the gather stage, each active vertex receives data from all gathering neighbors and a sum function is defined to aggregate the received data into a *gathered sum*. In the apply stage, each active vertex updates its data using the gathered sum. In the scatter stage, each active vertex activates its scattering neighbors for executions in the next iteration. A global barrier is defined to make sure all vertices have completed their computations before proceeding to the next iteration (the synchronized mode). For example, with PageRank algorithm, the vertex data is a numeric value indicating the *rank* of a webpage. The gather and scatter functions are executed on the in and out neighbors of a vertex, respectively. The sum function aggregates neighboring data using weighted sum.

Due to different user-defined vertex data and computation functions, the traffic pattern in each GAS iteration can be highly heterogeneous [8]. First, the data size transferred by different vertices can be highly variant, mainly due to the different degrees of vertices. In natural graphs, the degrees of vertices usually follow power-law distribution [7], which causes a small portion of vertices consuming most of the traffic during graph processing [8]. Second, the sent and received data sizes of a single vertex can also be different, mainly depending on the different features of graph processing algorithms and the number of in/out neighbors of the vertex. For example, in PageRank, the data sizes sent from and received by a vertex are often proportional to the number of out and in neighbors of the vertex, respectively. Constructing a synthetic power-law graph [7] with $\lambda = 2.1$, the difference between the numbers of in and out neighbors of a vertex can be up to 500x.

C. Graph Partitioning Methods

Existing graph processing engines adopt either edge-cut [5], [6] or vertex-cut [7], [19] method to partition graphs onto multiple machines by cutting them through edges or vertices, respectively. Edge-cut replicates cross-partition edges which cause data communication between partitions. Vertex-cut on the other hand replicates cross-partition vertices. Although it also causes communication between vertex replicas, it has the benefit of maintaining data locality for vertices. Recent studies find that vertex-cut is more efficient than edge-cut for graphs following power-law degree distribution [7]. As many real graphs usually follow power-law degree distribution, we focus on vertex-cut other than edge-cut in this paper.

The *Greedy* approach used in PowerGraph [7] is a vertex-cut partitioning method which is adopted by many graph processing engines aiming at minimizing the number of vertex replicas while achieving load balancing. Recently, a hybrid-cut method [20] is proposed to use edge-cut for low-degree vertices and vertex-cut for high-degree vertices for skewed graphs. However, both methods are not aware of the heterogeneity in network bandwidths and thus are not suitable for graph partitioning in geo-distributed DCs.

To provide light-weight partitioning for large-scale graphs, the one-pass streaming-based partitioning method has been utilized to generate fast and comparable graph partitioning results [21], [22]. Hereafter, we discuss several graph partitioning methods which are most relevant to this paper. Xu et. al [23] propose to consider the heterogeneous computing (e.g., CPU frequency) and communication (e.g., network bandwidth) capabilities when placing graph vertices to different machines, in order to minimize the execution time of graph processing jobs. However, their method cannot be applied directly to the graph partitioning problem in geo-distributed DCs, due to both inter- and intra-DC network heterogeneities in the geo-distributed environment. Mayer et. al [8] propose an adaptive streaming graph partitioning method named GrapH. It considers the heterogeneity in vertex traffic and networking prices during edge assignment (i.e., vertex-cut), aiming at minimizing the communication costs for graph processing jobs. However,

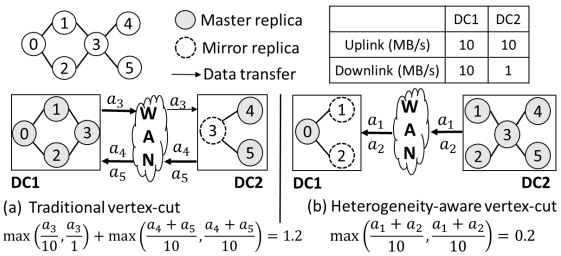

Fig. 2: A comparison between two graph partitioning methods: (a) traditional vertex-cut and (b) heterogeneity-aware vertex-cut. Widths of the up/down links indicate network bandwidths.

GrapH is not aware of the multi-level network bandwidth heterogeneities in geo-distributed DCs, and thus can lead to large inter-DC data transfer time. Chen at al. [24] studied the graph partitioning and placement on the heterogeneous network in a single data center. Thus, their study does not capture the feature of geo-distributed environments.

D. Motivation

We give a simple example to demonstrate the importance of a heterogeneity-aware graph partitioning method for geo-distributed DCs. Figure 2 shows a graph with six vertices, where the input data of vertex 0, 1 and 2 are stored in DC 1 and those of the rest vertices are stored in DC 2. The network bandwidths of the two DCs are given in the figure, where we can find that the downlink of DC 2 is the bottlenecked link.

With the vertex-centric abstraction of the GAS model, we can easily partition graphs using vertex-cut to parallelize vertex computations. For each vertex, one of the replicas is selected as the *master* and all the other replicas are called *mirrors*. In the gather stage, each replica gathers data from their local neighbors and sends the gathered data to the master. The master then aggregates all received data into the gathered sum. In the apply stage, the master updates the vertex data using the gathered sum and sends the updated data to all mirrors. In the scatter stage, each replica is responsible for activating its local neighbors. In this example, we assume the $Sum()$ function of the graph processing algorithm aggregates two messages by simply binding them together. For simplicity, we assume the data sizes of all vertices are the same ($\forall i = 0, \ldots, 5$, $a_i = 1$ MB). The inter-DC data transfer time can be calculated as the sum of the time spent in the gather stage (mirrors to master) and apply stage (master to mirrors).

Comparing the traditional vertex-cut approach [7] and our approach, we find that the traditional vertex-cut method has a smaller vertex replication rate and better load-balancing. However, due to the ignorance of network heterogeneity, the inter-DC networking of the traditional vertex-cut method is bounded by the downlink of DC 2. On the contrary, the heterogeneity-aware vertex-cut avoids using the downlink of DC 2 and thus can greatly reduce the inter-DC data transfer time from 1.2s to 0.2s.

III. Problem Formulation

In this paper, we study the graph partitioning problem in geo-distributed DCs. We first present the assumptions of the

TABLE II: Notation overview.

Symbol	Meaning
M	The number of geo-distributed DCs
$R(v)$	The set of DCs containing at least one replica of v
I_v^r	A boolean value indicating whether the replica of vertex v in DC r is the master ($I_v^r = 1$) or not ($I_v^r = 0$)
$g_v^r(i)$	The aggregated data size transfered from the mirror in DC r to the master of vertex v during the gather stage in iteration i
$a_v(i)$	The combined data size sent from the master of vertex v to each mirror in the apply stage of iteration i
U_r	The uploading bandwidth of DC r
D_r	The downloading bandwidth of DC r
B	The WAN usage budget

problem. Second, we mathematically formulate the problem as a constrained optimization problem. Table II summarizes the notations used in problem formulation.

A. Model Overview

We study how to partition a large graph onto multiple geo-distributed DCs. The graph data are generated and stored in geo-distributed locations. For simplicity, we assume that the graph data are not replicated across DCs initially and each machine executes only one vertex replica at a time. We assume that there are unlimited computation resources in each single DC, and the inter-DC data communication is the bottleneck to graph processing in geo-distributed DCs. This assumption is valid in geo-distributed environments since the WAN bandwidth is much more scarce than the computation resources such as CPU and memory. We also assume that the DCs are connected with a congestion-free network and the bottlenecks of the network are only from the uplinks/downlinks of DCs [11]. This assumption is based on the observation that many datacenter owners are expanding their services world wide and are very likely to build their own private WAN infrastructure [25]. The graph partitioning algorithm is applied before executing graph processing jobs.

B. Problem Definition

Consider a graph $G(V,E)$ with input data stored in M geo-distributed DCs, where V is the set of vertices and E is the set of edges in the graph. Each vertex v ($v = 0,1,\ldots,|V|-1$) has an initial location L_v ($L_v \in [0,1,\ldots,M-1]$), indicating at which DC the input data of vertex v is stored.

With the distributed GAS model, the inter-DC network traffic mainly comes from the gather stage and the apply stage. For a given iteration i and a vertex v, each mirror in DC r sends aggregated data of size $g_v^r(i)$ to the master of v in the gather stage and the master sends the combined data of size $a_v(i)$ to each mirror to update the vertex data in the apply stage. To simplify the calculation of data transfer time, we assume there is a global barrier between the gather stage and the apply stage. Thus, the data transfer time in iteration i can be formulated as the sum of the data transfer times in gather and apply stages. In each DC, the data transfer finishes when the data transfer on both up and down links are finished. Thus, we have:

$$T(i) = T_G(i) + T_A(i) = \max_r T_G^r(i) + \max_r T_A^r(i) \tag{1}$$

$$T_G^r(i) = max(\frac{\sum_v I_v^r \cdot \sum_{k \in R(v), k \neq r} g_v^k(i)}{D_r}, \frac{\sum_v (1 - I_v^r) \cdot g_v^r(i)}{U_r}) \tag{2}$$

$$T_A^r(i) = max(\frac{\sum_v I_v^r \cdot a_v(i) \cdot (|R(v)| - 1)}{U_r}, \frac{\sum_v (1 - I_v^r) \cdot a_v(i)}{D_r}) \tag{3}$$

where I_v^r is a boolean indicator showing whether the replica of vertex v in DC r is the master ($I_v^r = 1$) or not ($I_v^r = 0$). $R(v)$ is the set of DCs containing at least one replica of v and initially contains only L_v. U_r and D_r are the uplink and downlink bandwidths of DC r, respectively.

Our goal is to find an optimal assignment of edges to DCs, in order to minimize the inter-DC data transfer time while satisfying WAN usage budget constraint B. The WAN usage includes the cost of replicating vertex input data across DCs W_{rep} and the runtime inter-DC network traffic. We formulate the problem as a constrained optimization problem as below.

$$\min T(i) \tag{4}$$

$$\text{s.t.} \quad W_{rep} + \sum_i \sum_v \sum_{r \in R(v)} a_v(i) + g_v^r(i) \leq B \tag{5}$$

IV. Geo-Aware Graph Partitioning

Obtaining a good edge assignment in geo-distributed DCs is a challenging problem, mainly due to the heterogeneous graph traffic and multi-levels of network bandwidth heterogeneities. In this paper, we propose a geo-aware graph partitioning algorithm named *G-Cut*, which incorporates two optimization phases to address the two challenges separately. In the first phase, we study the traffic pattern in graphs and propose a streaming heuristic to minimize the inter-DC data transfer size. We then utilize the one-pass streaming partitioning method to quickly assign graph edges onto different DCs. In the second phase, we propose two partition refinement heuristics which identify the bottlenecks of network performance and refine the graph partitioning generated in the first phase accordingly to reduce the inter-DC data transfer time. In the following, we introduce the two optimization phases in details.

A. Traffic-Aware Graph Partitioning

The initial locations of vertices can greatly affect the graph partitioning results and the resulted inter-DC network traffic. Thus, we first discuss how to partition a graph given the initial locations of vertices and then consider moving the input data of vertices to further reduce the inter-DC network traffic size.

1) Streaming Graph Partitioning: Given the initial locations of vertices (i.e., where the input data of vertices are located), we adopt the streaming graph partitioning approach to quickly partition a graph. We view a graph as a stream of edges $e_0,\ldots,e_{|E|-1}$ to be assigned to a graph partition. The number of partitions is the same as the number of DCs. The order of edges in the set can be decided randomly or using breadth- or depth-first traversals. Existing studies have shown that the random order of edges can produce nearly the same result as that produced by optimized edge orders [21]. Thus, we randomly order the edges in the stream. One important design parameter in streaming partitioning is the heuristic which decides where to place an incoming edge. In this paper, we design a traffic-aware streaming heuristic.

When placing an edge (u,v) in a DC r, it can cause two types of additional network traffic to DC r. First, the

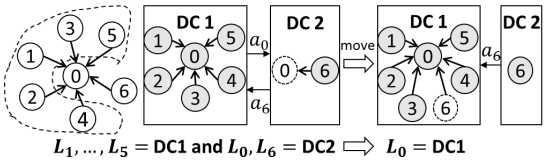

Fig. 3: Changing the initial location of vertex 0 can reduce the inter-DC traffic size.

replication of vertex u (v) in DC r increases the network traffic of the DC in iteration i by $a_u(i)$ ($a_v(i)$), which is used to synchronize the updated vertex data between the master and the added mirror replica of u (v) in the apply stage of the GAS model. Second, the data passed along the edge (u,v) increases the local aggregated data size of u and/or v during the communication between master and mirror replicas in the gather stage. Denote the aggregated data size transfered from the mirror of a vertex v in DC r to the master replica during the gather stage in iteration i as $g_v^r(i)$. We calculate the increased gathering traffic size caused by placing an edge (u,v) in DC r in iteration i as follows. Without loss of generosity, we assume graphs are directed and data are gathered from in-edges.

$$\Delta g_{u,v}^r(i) = \begin{cases} Sum(g_v^r(i), a_u(i-1)) - g_v^r(i) & , \text{ if } r \in R(v) \\ a_u(i-1) & , \text{ otherwise} \end{cases} \quad (6)$$

Based on the above analysis of graph traffic pattern, we derive the following streaming heuristic which always tries to place an incoming edge (u,v) to the DC with the lowest increased inter-DC network traffic.

1) If $R(v)$ and $R(u)$ intersect, place edge (u,v) into one of the intersected DCs r with the lowest $\Delta g_{u,v}^r(i)$.
2) If both $R(v)$ and $R(u)$ are not empty but do not intersect, place (u,v) in a DC $m \in R(v)$ or a DC $n \in R(u)$ with the lowest $a_u(i) + \Delta g_{u,v}^m(i)$ or $a_v(i) + \Delta g_{u,v}^n(i)$.
3) If only $R(v)$ or $R(u)$ is not empty, choose the DC r from the non-empty set with the lowest $\Delta g_{u,v}^r(i)$.
4) If both $R(v)$ and $R(u)$ are empty, place (u,v) in any DC $r \in [0, M-1]$ with the lowest $\Delta g_{u,v}^r(i)$.

After the edge assignment, we build local subgraphs in each DC and create vertex replicas as needed. The time complexity of the streaming graph partitioning is $O(|E|)$. As our streaming heuristic prioritizes DCs which do not require replicating vertices, our partitioning method can result in a low vertex replication rate and hence low W_{rep} size as shown in our evaluations. We select the replica with the largest number of local neighbors in each DC as the master of a vertex, in order to further reduce the inter-DC network traffic.

2) Moving Input Data of Vertices: We *move* the input data of a vertex by replicating the input data to another DC and set this new DC as the initial location of the vertex. Changing the initial locations of vertices can affect the streaming partitioning results and potentially lead to lower inter-DC network traffic. For example, as shown in Figure 3, moving the input data of vertex 0 from DC 2 to DC 1 can reduce the inter-DC data transfer size by $a_0(i)$.

In many graphs, the input data sizes of vertices are much larger than the intermediate data size transferred during graph processing. For example, when we query for users who have twitted on the same topic in the Twitter graph, the input data of each vertex may contain the past tweets of the user while the intermediate data can be only the IDs of the users matching the query. Thus, we mainly apply the input data movement to high-degree vertices, which are usually the hotspots of data processing in graph algorithms and have large sizes of intermediate data. We choose the top *threshold* high-degree vertices as moving candidates and order them in a queue according to their scores. The score of a candidate is calculated as the difference between the reduced inter-DC data transfer size caused by moving the candidate and its input data size. Iteratively, we move the input data of the candidates in the queue to the DCs which lead to the largest reduction to inter-DC data transfer size until no further improvement can be obtained or the WAN usage budget constraint is violated.

B. Network-Aware Partition Refinement

After the traffic-aware partitioning, we obtain graph partitions with small inter-DC data traffic size. In this phase, we consider the multi-levels of network heterogeneities in geo-distributed DCs and propose two heuristics to address them. First, we have a partition mapping heuristic which takes the heterogeneous uplink and downlink bandwidths into consideration, and maps the partitions to different DCs in order to reduce the inter-DC data transfer time. Second, we consider the network heterogeneities between different DCs and use edge migrations to diminish the data traffic in bottlenecked DCs to further reduce the inter-DC data transfer time.

1) Partition Mapping: Mapping the M graph partitions to M geo-distributed DCs is a classic combinatorial NP-hard problem and has a solution space of $O(M!)$. For small values of M, we can simply adopt BFS/DFS-based search algorithms to find the optimal mapping solution. For a large number of DCs, we provide the following heuristic: starting from an initial mapping, we iteratively identify the bottlenecks of the inter-DC data communication and relocate the graph partitions in the bottlenecked DCs. We adopt the graph partitioning result generated in the first phase as our initial partition mapping. Relocating graph partitions involves large size of data movement. Thus, in order to optimize inter-DC data transfer time while satisfying WAN usage budget, we consider the following two subproblems: 1) how to identify the bottlenecks of inter-DC data communication and 2) where to relocate the graph partitions in the bottlenecked DCs.

According to Equations 1–3, we can identify the bottlenecks of inter-DC data communication by estimating the data transfer times in the gather and apply stages for each DC r, i.e., $T_G^r(i)$ and $T_A^r(i)$, respectively. The bottlenecks for the gather/apply stage are the DCs which have the same data transfer time as $T_G(i)/T_A(i)$. The bottleneck can be bounded either on the uplink or the downlink of the DC. Denote the bottlenecked DC in the gather and apply stages as d_1 and d_2, respectively. As described in Algorithm 1, if $d_1 == d_2$, we adopt Algorithm 2 to choose a DC and switch the graph partition in the bottlenecked DC with it (Lines 6–10). Otherwise,

we estimate the possible gains obtained by switching the graph partitions in d_1 and d_2 with other DCs individually (Lines 12–15). We choose one from d_1 and d_2 with larger gain to do partition switching if the WAN budget allows (Lines 16–19). After the partition switching, we identify the new bottlenecks and repeat the above optimizations until no further gain can be obtained or a fixed number of iterations have been reached.

Algorithm 1 Graph partition mapping heuristic.

Require: P_{init}: the initial partition mapping solution;
Ensure: P_{opt}: the optimized partition mapping solution;
1: $P_{opt} = P_{init}$;
2: **repeat**
3: $continue = false$;
4: Identify d_1/d_2 as the bottlenecked DC in the gather/apply stage;
5: **if** $d_1 == d_2$ **then**
6: $(G_b, d_{opt}) = \text{EstimateGain}(P_{opt}, d_1)$;
7: U is the sum of the current WAN usage and the data size in d_1 and d_{opt};
8: **if** $U \leq B$ **then**
9: Switch the partition in d_1 with that in d_{opt} for plan P_{opt};
10: $continue = true$;
11: **else**
12: Find a DC d_{opt1} from $\{0,\ldots,M-1 \setminus d_2\}$ for d_1 to switch with, using $(G_1, d_{opt1}) = \text{EstimateGain}(P_{opt}, d_1)$;
13: Find a DC d_{opt2} from $\{0,\ldots,M-1 \setminus d_1\}$ for d_2 to switch with, using $(G_2, d_{opt2}) = \text{EstimateGain}(P_{opt}, d_2)$;
14: $d = G_1 > G_2 ? d_1 : d_2$;
15: $d_{opt} = G_1 > G_2 ? d_{opt1} : d_{opt2}$;
16: U is the sum of the current WAN usage and the data size in d and d_{opt};
17: **if** $U \leq B$ **then**
18: Switch the partition in d with that in d_{opt} for plan P_{opt};
19: $continue = true$;
20: **until** $!continue$
21: **return** P_{opt};

We adopt Algorithm 2 to make the partition relocation decisions, in which we iteratively compare the gain of switching the graph partition in the bottlenecked DC with any other DCs and choose the one with the best gain to switch to. The gain of a partition switching is calculated as $\frac{\Delta T}{C}$, where ΔT is the reduction to the estimated data transfer time after partition switching calculated using Equation 4 (Line 5) and C is the sum of input data sizes in the bottlenecked DC and the DC to be switched with, i.e., the data movement cost (Line 6).

Algorithm 2 EstimateGain(P_{init}, d_0)

Require:
P_{init}: the partition mapping solution to be optimized;
d_0: the bottlenecked DC;
Ensure:
G_b: the best gain obtained by replacing the graph partition in d_0;
d_1: the DC chosen to switch graph partition with d_0;
1: $T_0 = \text{EstimateTime}(P_{init})$;
2: $G_b = 0$;
3: **for each** DC $d \in \{0,\ldots,M-1 \setminus d_0\}$ **do**
4: P is P_{init} after switching the graph partition in d_0 with that in d;
5: $\Delta T = T_0 - \text{EstimateTime}(P)$;
6: C is the sum of the input data size in DC d_0 and d;
7: **if** $\frac{\Delta T}{C} > G_b$ **then**
8: Let $G_b = \frac{\Delta T}{C}$ and $d_1 = d$;
9: **return** G_b and d_1;

Complexity Analysis: The worst-case time complexity of Algorithm 2 is $O(M)$. Assuming the pre-defined maximum number of iterations to check for new bottlenecks in the inter-DC data communications is *MaxIter*, the worst-case time complexity of our partition mapping heuristic is $O(MaxIter \times M)$, which is much faster than the naive search algorithms.

2) Edge Migration: Network bandwidth heterogeneity between different DCs can cause performance bottlenecks even with the optimal partition mapping. For example, as shown in Table I, both the uplink and downlink bandwidths of Sydney are lower than the other two cloud DCs. Thus, an equal partition of the graph workload will lead to performance bottleneck on the Sydney DC. To mitigate such performance bottlenecks, we propose to migrate edges out of the bottlenecked DCs after obtaining the optimal partition mapping.

We identify the bottlenecked DCs and *links* in the same way as the partition mapping step. There can be four types of bottlenecked links, namely uplink/downlink bounded in the gather stage and uplink/downlink bounded in the apply stage. If the gather and apply stages are bounded on the same link of the same DC, we migrate edges out of that DC considering the improvement to the inter-DC data transfer time. Otherwise, we select the link with more possible reduction to the data transfer time of the gather and apply stages to migrate.

Based on Equations 1–3, we address a bottlenecked link l_r of DC r as follows. If l_r is uplink and bounds the gather stage, the network traffic on l_r is mainly caused by mirror replicas in the DC sending gathering data to the masters. Thus, we order the mirror replicas according to their gathering data sizes (i.e., $g_v^r(i)$ for the mirror of vertex v in DC r) in a priority queue Q. We iteratively remove the vertices in Q until l_r is no longer the bottleneck or Q is empty. If l_r is downlink and also bounds the gather stage, the network traffic on l_r is mainly caused by master replicas receiving the gathering data from mirrors. Thus, we order the master replicas in DC r according to their non-local gathering data sizes. After removing the master replicas, we choose new master replicas for each vertex using the same policy as introduced in the streaming graph partitioning step. Similarly, for bottlenecks of the apply stage, if l_r is uplink (downlink), we order the master (mirror) replicas in DC r according to the vertex data sizes. After removing a vertex replica v, we migrate the edges connected to v to a DC which results in the minimum inter-DC data transfer time. We calculate the inter-DC data transfer time after migrating an edge using Equation 4 with the updated inter-DC network traffic. By default, we migrate the edges one at a time.

Complexity Analysis: The average time complexity of the edge migration can be calculated as $O(|Q| \times \frac{|E|}{|V|} \times M)$. As $|Q|$ is on average $\frac{|V|}{M}$, we have the time complexity as $O(|E|)$. To balance the trade-off between the effectiveness and efficiency of edge migrations, we provide two optimizations. First, as introduced in Section II, it is common in power-law graphs that a small portion of vertices are contributing to most of the data traffic. Thus, we can limit the length of Q to a small value L_Q while achieving similar migration results. Second, for each vertex candidate v in Q, we group the edges connected to v into C groups, where edges in the same group will be migrated at the same time to the same DC. We adopt the clustering method to group edges, and the distance between two edges is defined as the intersection size between the replication locations of the other end of the edges. For example, for two edges (u,v) and (w,v), the distance between them is defined as $|R(u) \cap R(w)|$. In this way, we can minimize the number of additional vertex replications and hence the size of additional

inter-DC data transfer caused by edge migrations. With the two optimizations, we are able to reduce the time complexity of edge migration to $O(L_Q \times C \times M)$.

V. Partition Refinement for Dynamic Graphs

With G-Cut, we are able to partition a static graph in $O(|E|)$ time, assuming $|E| \gg M$. However, many real-world graphs such as social network graphs are dynamic, with frequent vertex and edge insertions and deletions (e.g., newly registered/deactivated users and following/unfollowing operations). These changes to a graph can greatly affect graph partitioning decisions. For example, a deactivated celebrity Facebook user, who usually has a large number of followers, can greatly affect the network traffic of the new social network graph. Repartitioning the entire graph once changes occur is not cost-effective due to the overhead of repartitioning. In the following, we discuss how to use G-Cut to adaptively partition dynamic graphs with a low overhead.

Since adding and removing a vertex can be represented by adding and removing edges connected to this vertex, we abstract the changes to a dynamic graph as edge insertions and edge deletions, assuming that a graph has no isolated vertex. It has been pointed out by existing studies [26] that a dynamic graph can be viewed as an intermediate state of the streaming graph partitioning. Thus, we can adopt the streaming graph partitioning technique of G-Cut to directly assign inserted edges to DCs. But the challenge is that, the inserted/deleted edges can change the data traffic of vertices and thus make the existing partitioning less effective. For example, when inserting a large number of edges connected to an originally low-degree vertex v, it is better to move v to a DC with high bandwidths.

Algorithm 3 PartitionRefine(e, *threshold*).

1: **if** e is an inserted edge to partition r **then**
2: increase Δd^r_+ by $\Delta g^r_e(i)$ and $a_{src(e)}(i)/a_{tgt(e)}(i)$, if vertex $src(e)/tgt(e)$ is inserted;
3: **if** $\Delta d^r_+ > threshold$ **then**
4: trigger partition mapping;
5: **else if** e is an deleted edge from partition r **then**
6: increase Δd^r_- by $\Delta g^r_e(i)$ and $a_{src(e)}(i)/a_{tgt(e)}(i)$, if vertex $src(e)/tgt(e)$ is deleted;
7: **if** $\Delta d^r_- > threshold$ **then**
8: trigger edge migration;

To address this challenge, we periodically apply our partition refinement technique to the updated graph partitions. Specifically, we use partition mapping to relocate the graph partitions which have new edges inserted. We use the edge migration technique to migrate edges from the current bottlenecked DC to DCs which have deleted edges, if the inter-DC data transfer time can be reduced. It is costly to perform the refinement every time an edge is inserted/deleted. Thus, we need to decide the timing of partition refinement. As shown in Algorithm 3, we calculate the increased/decreased data traffic size to a partition caused by edge insertions/deletions using the same way as introduced in streaming graph partitioning. We trigger the partition refinement whenever the amount of changes to data traffic caused by edge insertions/deletions is larger than a threshold. By default, we set *threshold* to 10% of the overall data traffic size of a partition. The time complexity of inserting a set of edges E' is $O(|E'| + MaxIter \times R)$ and that of deleting a set of edges is $O(L_Q \times C \times R)$, where R is the number of partition refinement operations applied. This is much faster than repartitioning the entire graph.

TABLE III: Experimented real-world graphs.

Notation	#Vertices	#Edges	α_{in}	α_{out}
Gnutella (GN)	8,104	26,013	2.91	2.59
Facebook (FB)	4,039	88,234	4.85	3.14
WikiVote (WV)	7,115	103,689	3.63	3.80
GoogleWeb (GW)	875,713	5,105,039	2.96	3.64
LiveJournal (LJ)	3,577,166	44,913,072	3.45	2.88

VI. Evaluation

We evaluate the effectiveness and efficiency of G-Cut using real-world graph datasets on Amazon EC2 and with simulations on Grid'5000 [27]. To emulate the congestion-free network model, we limit the uplink and downlink bandwidths of the instances to be smaller than the WAN bandwidth. The limited bandwidths are proportional to their original bandwidths. We adopt the GAS-based PowerGraph [7] system to execute graph processing algorithms. The evaluated graph partitioning methods are implemented using C++ and integrated in PowerGraph to assign graph edges while loading. We adopt a multi-threaded implementation to parallelize streaming graph partitioning.

A. Experimental Setup

Graphs. We select five real-world graphs for our experiments, which are representative graphs in P2P networks, social networks and web graphs. Table III shows the number of vertices and edges in the graphs [28].

Graph algorithms. We adopt three graph algorithms which are widely used in different areas.

PageRank (PR) [29] is widely used in web information retrieval to evaluate the relative importance of webpages. The web is modeled as a graph, where each webpage is a vertex and the links between webpages are edges of the graph. Each vertex has a rank value, which gets updated according to the rank values of neighboring vertices.

Shortest Single Source Path (SSSP) [30] finds the shortest paths starting from a single source to all other vertices in the graph. It has been applied in social network analysis to study the relationships between users and in road networks to automatically find directions between two locations.

Subgraph Isomorphism (SI) [31] is used to find the subgraphs matching certain graph pattern in a large graph. It is used in diverse areas as social networks and intelligence analysis for pattern matching of graph-structured data.

Compared methods. We compare G-Cut with three state-of-the-art vertex-cut graph partitioning methods, namely *Random* [7], *Greedy* [7] and *GrapH* [8]. Random assigns an edge to a random DC (the one containing the source or target node of the edge if possible). Greedy iteratively places edges to the DCs which minimizes the expected vertex replication factor. GrapH considers the heterogeneity in vertex traffic and network pricing, and assigns edges to minimize the monetary communication costs.

Configuration details. We use both real-world experiments and simulations to evaluate the effectiveness of G-Cut.

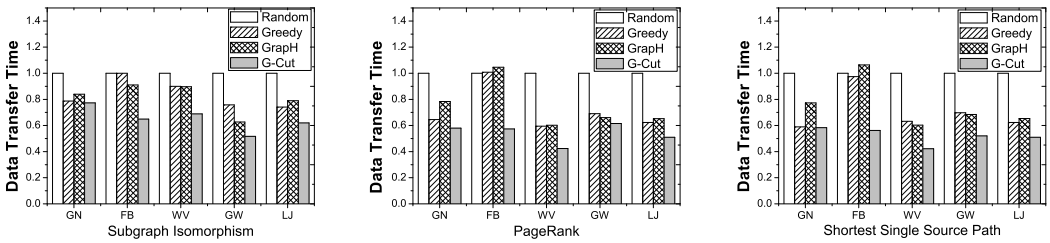

Fig. 4: Normalized results of the inter-DC data transfer time optimized by the compared methods for the three graph algorithms.

For real-world experiments, we select eight regions of Amazon EC2 as the geo-distributed DCs, namely US East (USE), US West Oregon (USW-O), US West North California (USW-NC), EU Ireland (EU), Asia Pacific Singapore (SIN), Asia Pacific Tokyo (TKY), Asia Pacific Sydney (SYD) and South America (SA). In each region, we construct a cluster of five cc2.8xlarge instances. In all experiments, we compare the performance and WAN usage of graph algorithms optimized by G-Cut and the three compared methods. It is possible that WAN bandwidths are charged differently in different geographic locations. Thus, we also present the monetary cost of inter-DC data communication optimized by the compared methods, using the real network prices charged by Amazon EC2. As GrapH is designed to optimize the data communication cost for graph algorithms, we set the WAN budget to be the WAN usage optimized by GrapH by default.

For simulations, we simulate 20 geo-distributed DCs using the network performances measured from Amazon EC2. All DCs adopt the same network pricing as the US East region. We perform three sets of simulations. First, we construct three geo-distributed environments with "Low", "Medium" and "High" network heterogeneities and study the impact of network heterogeneity on the effectiveness of G-Cut. Specifically, in Low, the uplink and downlink of all DCs have the same bandwidths. In Medium, all DCs have the same upload/download bandwidths as those measured from the US East region of Amazon EC2. In High, we randomly select five DCs from Medium and proportionally limit their upload/download bandwidths to a half of their original bandwidths. To quantitatively define the heterogeneity of a network, we use the relative standard deviation of the bandwidths of all links in the network as the metric. For example, the heterogeneity of the geo-distributed network environment shown in Figure 1 is calculated as $\frac{std(U_1,U_2,U_3,D_1,D_2,D_3)}{mean(U_1,U_2,U_3,D_1,D_2,D_3)} = 0.79$. Thus, the normalized heterogeneities of Low, Medium and High are 0, 0.68 and 0.89, respectively. Second, we study the impact of WAN budget on the effectiveness of G-Cut. We define a "'Loose" and a "Tight" budget constraint as $0.25 \times W_{min} + 0.75 \times W_{default}$ and $0.5 \times (W_{min} + W_{default})$, respectively, and evaluate the performance of G-Cut under the High network heterogeneity. $W_{default}$ is the default WAN budget and W_{min} is the WAN usage of G-Cut with the streaming graph partitioning technique only. Third, we study the effectiveness of G-Cut on supporting dynamic graphs. We take a half of the edges in a graph as the initial static graph and insert the rest of the edges. We adopt GW graph for all simulations.

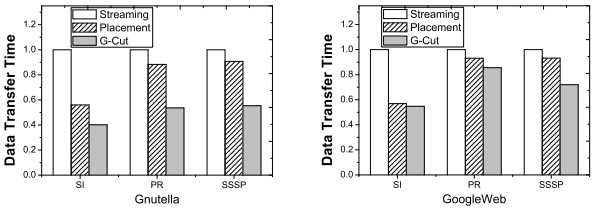

Fig. 5: Normalized data transfer time optimized by traffic-aware graph partitioning only (Streaming), both Streaming and partition mapping (Placement) and G-Cut.

B. Real Deployment Results

Inter-DC data transfer time. Figure 4 shows the normalized inter-DC data transfer time optimized by the compared partitioning methods for SI, PR and SSSP algorithms on the five real-world graphs. All results are normalized to the result of Random. We have two observations. First, G-Cut is able to obtain the lowest inter-DC data transfer time results under all settings. G-Cut reduces the execution time of SI algorithm by 2%–23% on GN, 29%–35% on FB, 23%–31% on WV, 17%–48% on GW, and 16%–38% on LJ compared to the other graph partitioning methods. Similarly, G-Cut reduces the data transfer time by 7%–58% for PR and 1%–58% for SSSP. Second, G-Cut performs better for graphs with more heterogeneous data communications. For example, the reduction in the data transfer time obtained by G-Cut is relatively small on GN than the other graphs. We use power-law distribution to fit the five evaluated graphs and present the α parameter of the fitted distributions in Table III. The α parameter of GN is smaller than the other graphs, which means the vertex degrees, hence data communications, in GN is more balanced than the other graphs. This also explains the small improvement of G-Cut on GW with PR algorithm (PR gathers from in neighbors only). This observation demonstrates that the heterogeneity-aware techniques in G-Cut are effective in reducing the inter-DC data communication size and time for graph processing jobs in geo-distributed DCs.

We further breakdown the results of G-Cut to evaluate the effectiveness of partition mapping and edge migration techniques in reducing inter-DC data transfer time. Figure 5 shows the data transfer time optimized by the traffic-aware graph partitioning technique only (denoted as Streaming), both Streaming and partition mapping (Placement) and G-Cut for GN and GW graphs. All results are normalized to those of Streaming. The network-aware partition mapping is especially effective in reducing the inter-DC data transfer time of the SI

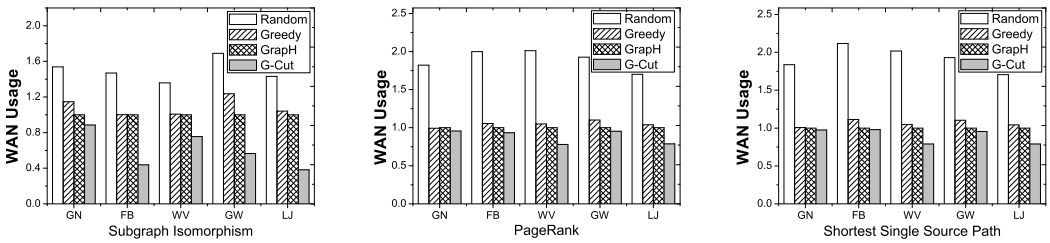

Fig. 6: Normalized results of the WAN usage optimized by the compared methods for the three graph algorithms.

TABLE IV: Load-balancing measurements of the compared methods.

	Random	GrapH	G-Cut
SI	40	38	95
PR	66	31	68
SSSP	36	26	52

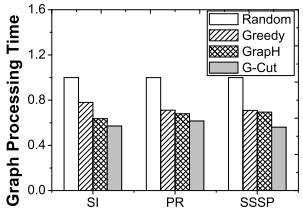

Fig. 7: Graph processing time

algorithm. This is mainly because of the high vertex traffic heterogeneity in SI as also mentioned in existing studies [8]. As a result, relocating graph partitions in bottlenecked DCs can bring more reduction to inter-DC data transfer time. For example, with GW, the time reduced by relocating a graph partition in SI is on average 6.7x and 6.6x higher than that in PR and SSSP, respectively. The edge migration technique is also effective in reducing the data communication overhead. For example, it reduces the data transfer time by 28%–39% and 2%–23% over the results of Placement for GN and GW, respectively.

WAN usage. Figure 6 shows the WAN usage obtained by the compared methods for SI, PR and SSSP algorithms on the five real-world graphs. All results are normalized to the WAN usage budget. G-Cut obtains the lowest WAN usage and is able to satisfy the WAN budget constraint under all settings. G-Cut reduces the WAN usage by 42%–70%, 3%–56% and 2%–56% compared to Random, Greedy and GrapH. Greedy and GrapH obtain similar WAN usage results, as they are both designed to achieve load-balancing among graph partitions. We measure the load-balancing results of the compared methods by calculating the variances of workloads in different graph partitions. Table IV shows the load-balancing measurements of the compared methods on GW with different graph algorithms. All results are normalized to those of Greedy. Although G-Cut has poor load-balancing results, we claim that the computation resources in geo-distributed DCs are less scarce than the WAN bandwidth and thus has a smaller impact on the overall performance of graph processing jobs. This claim can be verified with Figure 7, which shows the overall graph processing time optimized by the compared algorithms for GW graph. We find that despite the less balanced workload distribution of G-Cut, it still obtains the best overall graph processing time.

Inter-DC data transfer monetary cost. Figure 8 shows the normalized monetary cost results obtained by the compared methods for inter-DC data transfer. All results are normalized to those of GrapH. G-Cut obtains lower monetary cost than Random and Greedy in all settings. Comparing with GrapH, G-Cut is able to obtain lower monetary cost in many cases due to its low WAN usage. In some cases, such as for GN and FB graphs running PR and SSSP algorithms, the monetary cost of GrapH is lower than G-Cut. This is because the WAN usages of the two methods in those cases are very close and GrapH is able to distribute data communications to less expensive DCs. Table V shows the percentage of the overall uploading data sizes in each DC optimized by GrapH and G-Cut, using FB graph and PR algorithm. The above observations demonstrate that, although G-Cut is not specifically designed for monetary cost optimizations, it still can achieve good monetary cost results, especially for graphs with heterogeneous traffic.

TABLE V: Percentage of the overall uploading data sizes in different DCs obtained by GrapH and G-Cut, using FB graph and PR algorithm.

	USE	USW-O	USW-NC	EU	SIN	TKY	SYD	SA
price ($)	0.02	0.02	0.02	0.02	0.09	0.09	0.14	0.16
GrapH (%)	17	18	18	11	7	8	11	10
G-Cut (%)	13	13	14	13	13	13	8	12

TABLE VI: Vertex replication rates of different graphs and the SI algorithm optimized by the compared methods.

	Random	Greedy	GrapH	G-Cut
Gnutella	3.24	2.87	2.64	2.59
Facebook	6.30	4.17	4.24	2.76
WikiVote	4.28	2.88	2.75	2.60
GoogleWeb	1.64	1.47	1.33	1.30
LiveJournal	5.06	3.25	3.04	2.35

Replication rate. As shown in Table VI, G-Cut is able to obtain the lowest vertex replication rate among all compared graph partitioning methods. This is expected as our streaming heuristic in the streaming graph partitioning technique prefers to assign an edge to the partition which does not require vertex replication. The low replication rate of G-Cut is one reason that G-Cut has a small WAN usage compared to the other graph partitioning methods.

Graph partitioning overhead. To reduce edge migration overhead, we set the L_Q parameter to the number of vertices in Q which have data traffic sizes larger than 10% of that of the head of Q. The grouping parameter C is set to $\frac{L_Q}{10}$. Figure 10 shows the overhead breakdown of G-Cut for different graphs with SI algorithm. The streaming graph partitioning takes a large portion of the overall overhead. On average, it takes less than 8 seconds for G-Cut to partition one million edges, which is comparable to existing methods (usually several seconds for one million edges [8], [14]).

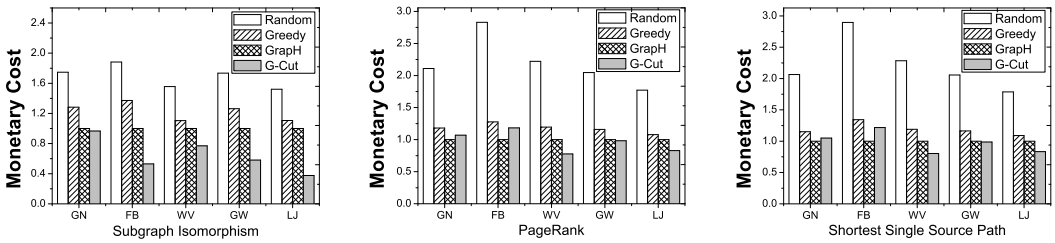

Fig. 8: Normalized monetary cost results optimized by the compared methods for the three graph algorithms.

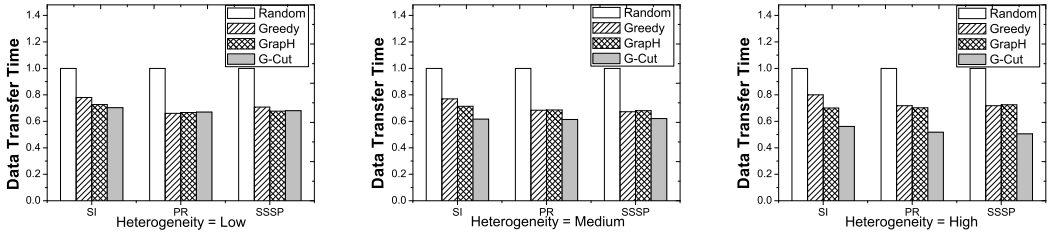

Fig. 9: Normalized inter-DC data transfer time optimized for the GW graph under different network heterogeneities.

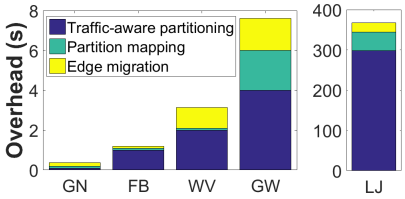

Fig. 10: Breakdown of graph partitioning overhead of G-Cut.

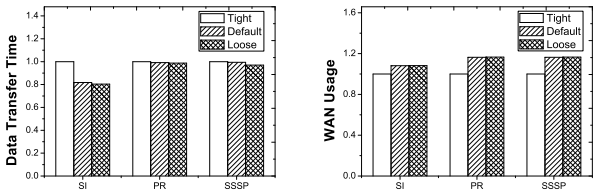

Fig. 11: Sensitivity study on the WAN usage budget constraint.

C. Simulation Results

Network heterogeneity. Figure 9 shows the normalized inter-DC data transfer time optimized by the compared algorithms under different network heterogeneities. All results are normalized to those of Random. G-Cut reduces the inter-DC data transfer time over the other three partitioning methods by 1%–33%, 8%–39% and 20%–49% for Low, Medium and High network heterogeneities, respectively. This shows that G-Cut can effectively address the network heterogeneity problem for graph processing in geo-distributed DCs.

WAN usage constraint. Figure 11 shows the normalized data transfer time and WAN usage results of G-Cut under different budget constraints. All results are normalized to those of Tight. We have two observations. First, G-Cut can further reduce the data transfer time when the WAN budget gets loose (e.g., from Tight to Loose). Second, with the same amount of increase in the WAN usage, G-Cut can reduce the inter-DC data transfer time more for SI algorithm than PR and SSSP. This is mainly because the input data sizes of PR and SSSP are much larger than their transfered message sizes.

Dynamic edge insertions. We normalize the optimization results obtained by partitioning the dynamic GW graph to the results of partitioning the entire GW graph using G-Cut. The normalized data transfer time for SI, PR and SSSP are 1.02, 1.05 and 1.05, respectively. The normalized WAN usage for SI, PR and SSSP are 1.32, 1.24, and 1.29, respectively. During edge insertions, there are 791 times of partition refinements for SI, 10 times for PR and 30 times for SSSP. This is mainly because the data traffic heterogeneity in SI is more severe than the other two graph algorithms.

VII. Conclusion

In this paper, we propose a geo-aware graph partitioning method named G-Cut, to minimize the inter-DC data transfer time of graph processing jobs in geo-distributed DCs while satisfying the WAN usage budget. G-Cut incorporates two optimization phases. While the first phase utilizes the one-pass streaming graph partitioning method to reduce inter-DC data traffic size when assigning edges to different DCs, the second phase identifies network bottlenecks and refines graph partitioning accordingly. The experiment results on both real geo-distributed DCs and with simulations have demonstrated that G-Cut is effective in reducing the inter-DC data transfer time with a low runtime overhead. As future work, we plan to extend our techniques to other graph processing models, experiment on graphs with larger sizes and heterogeneous computing environments with GPUs [32].

Acknowledgment

This work is supported by the ANR KerStream project (ANR-16-CE25-0014-01). Bingsheng's work is partially funded by a collaborative grant from Microsoft Research Asia and a NUS start-up grant in Singapore. The experiments presented in this paper were carried out using the Grid5000/ALADDIN-G5K experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/ for details).

REFERENCES

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *VLDB*, vol. 8, no. 12, pp. 1804–1815, 2015.

[2] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *WSDM '13*, pp. 507–516.

[3] L. Zhu, A. Galstyan, J. Cheng, and K. Lerman, "Tripartite graph clustering for dynamic sentiment analysis on social media," in *SIGMOD '14*, pp. 1531–1542.

[4] E. Minkov, W. W. Cohen, and A. Y. Ng, "Contextual search and name disambiguation in email using graphs," in *SIGIR '06*, 2006, pp. 27–34.

[5] 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 '10*, pp. 135–146.

[6] 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," *VLDB*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *OSDI'12*, pp. 17–30.

[8] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel, "Graph: Heterogeneity-aware graph computation with adaptive partitioning," in *Proc. of IEEE ICDCS*, 2016.

[9] Scaling the Facebook data warehouse to 300 PB, *https://goo.gl/Eyv6o3*.

[10] "The court of justice declares that the commissions us safe harbour decision is invalid," https://goo.gl/vLg6aw, 2015.

[11] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *SIGCOMM '15*, 2015, pp. 421–434.

[12] AWS Direct Connect Pricing, *https://aws.amazon.com/directconnect/pricing/*, accessed Nov 2016.

[13] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 460–471, Sep. 2010.

[14] K. Schloegel, G. Karypis, and V. Kumar, "Parallel static and dynamic multi-constraint graph partitioning," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 3, pp. 219–240, 2002.

[15] AWS Global Infrastructure, *https://aws.amazon.com/about-aws/global-infrastructure/*, accessed on Oct 2016.

[16] Windows Azure Regions, *https://azure.microsoft.com/en-us/regions/*, accessed on Oct 2016.

[17] Google Datacenter Locations, *https://www.google.com/about/datacenters/inside/locations/index.html*, accessed on Oct 2016.

[18] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *OSDI '16*, GA, Nov. 2016, pp. 301–316.

[19] 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'14*, pp. 599–613.

[20] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *EuroSys '15*, pp. 1:1–1:15.

[21] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *KDD '12*, 2012, pp. 1222–1230.

[22] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *WSDM '14*, 2014, pp. 333–342.

[23] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao, "Heterogeneous environment aware streaming graph partitioning," *IEEE TKDE*, vol. 27, no. 6, pp. 1560–1572, 2015.

[24] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, "Improving large graph processing on partitioned graphs in the cloud," in *SoCC '12*. ACM, 2012, pp. 3:1–3:13.

[25] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *SIGCOMM '13*, 2013, pp. 3–14.

[26] J. Huang and D. J. Abadi, "Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs," *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 540–551, Mar. 2016.

[27] S. Badia, A. Carpen-Amarie, A. Lèbre, and L. Nussbaum, "Enabling Large-Scale Testing of IaaS Cloud Platforms on the Grid'5000 Testbed," in *1st International Workshop on Testing The Cloud*, 2013, pp. 7–12.

[28] Stanford Large Network Dataset Collection, *https://snap.stanford.edu/data/*, accessed on Oct 2016.

[29] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *WWW7*, 1998, pp. 107–117.

[30] D. P. Bertsekas, F. Guerriero, and R. Musmanno, "Parallel asynchronous label-correcting methods for shortest paths," *J. Optim. Theory Appl.*, vol. 88, no. 2, pp. 297–320, Feb. 1996.

[31] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *VLDB*, vol. 5, no. 4, pp. 310–321, Dec. 2011.

[32] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014.