GrapH: Heterogeneity-Aware Graph Computation with Adaptive Partitioning

Christian Mayer Muhammad Adnan Tariq Chen Li Kurt Rothermel
Institute of Parallel and Distributed Systems, University of Stuttgart, Germany
Email: {christian.mayer, adnan.tariq, chen.li, kurt.rothermel}@ipvs.uni-stuttgart.de

***Abstract*—Vertex-centric graph processing systems such as Pregel, PowerGraph, or GraphX recently gained popularity due to their superior performance of data analytics on graph-structured data. These systems exploit the graph-structure to improve data access locality during computation using graph partitioning algorithms. Recent partitioning techniques assume a uniform and constant amount of data exchanged between graph vertices (i.e., uniform vertex traffic) and homogeneous underlying network costs. However, in real-world scenarios vertex traffic and network costs are heterogeneous. This leads to sub-optimal partitioning decisions and inefficient graph processing. To this end, we designed GrapH, the first graph processing system using vertex-cut graph partitioning that considers both diverse vertex traffic and heterogeneous network, to minimize overall communication costs. The main idea is to avoid frequent communication over expensive network links using an adaptive edge migration strategy. Our evaluations show an improvement of 60% in communication costs, compared to state-of-the-art partitioning approaches.**

I. Introduction

In recent years, a strong demand to perform complex data analytics on graph-structured data sets, such as the web graph, simulation grids, Bayesian networks, and social networks [1]–[4] has lead to the advent of distributed graph processing systems, such as Pregel, PowerGraph, and GraphX [5]–[7]. These systems adopt a user-friendly programming paradigm, where users specify vertex functions to be executed in parallel on vertices distributed across machines ("think-like-a-vertex"). During execution, vertices iteratively compute their local state based on the state of neighboring vertices, therefore efficient communication across vertices is vital in building highly-efficient graph processing systems. In fact, recent work on data analytics frameworks suggests that network-related costs are often the bottleneck for overall computation [8]–[11].

To overcome these inefficiencies, graph processing systems require suitable partitioning methods improving the locality of vertex communication. Mainly, there are two types of partitioning strategies: edge-cut and vertex-cut. These strategies minimize the number of times an edge or vertex spans multiple machines (*cut-size*). The idea is that a decreased cut-size leads to lower communication costs due to less inter-machine traffic [6], [7]. But this holds only under two assumptions: *vertex traffic homogeneity*, i.e., processing each vertex involves the same amount of communication overhead, and *network homogeneity*, i.e., the underlying network links between each pair of machines have the same usage costs (e.g., [6], [11]). However, these assumptions oversimplify the target objective, i.e., **minimize overall communication costs**, for two reasons.

First, real-world vertex traffic is rarely homogeneous. This is due to computational hotspots causing processing to be unevenly distributed across graph areas and vertices. Hotspots arise mainly for three reasons: i) the vertices process different amounts of data, ii) the graph system executes vertices a different number of times, and iii) the graph analytic algorithms concentrate on specific graph areas. Examples of the first group are large-scale simulations of heart cells [12], liquids or gases in motion [13], and car traffic in cities [1], where each vertex is responsible for a small part of the overall simulation. Vertices simulating real-world hotspots (e.g., the Times Square in NY) have to process more data. The second group consists of algorithms defining a convergence criteria for vertices. The graph system skips execution of converged vertices (*dynamic scheduling* [6]) leading to inactive graph areas and therefore different frequencies of vertex execution. Concerning this, a popular example is the PageRank algorithm [6]. The third group include user centric graph analytic algorithms such as k-hop random walk and graph pattern matching. A prominent example is Facebook Graph Search, where users pose search queries to the system ("find friends who tried this restaurant"). In general, our evaluations show that vertex traffic often resembles a Pareto distribution, whereby a higher percentage of the total traffic is contributed by a much lower percentage of the vertices (cf. Fig. 2). We argue, that the one-size-fits-all approaches for vertex traffic misfit real-world, heterogeneous and dynamic traffic conditions in modern graph processing systems.

Second, network-related costs, such as bandwidth, latency, or monetary costs, are subject to large variations. Today, it is common to run graph analytics in the cloud, because of low deployment costs and high scalability [6], [14], [15]. Network heterogeneity exits even in a single data center because the machines are connected via a tree-structured switch topology, where machines connected to the same switch experience high-speed communication, while distant machines suffer from degraded performance due to multi-hop forwarding of network packages [16]. Nevertheless, modern cloud infrastructures are *geo-distributed* [17]. Cloud providers are deploying data centers globally to provide low latency user-access. For instance, Amazon, Google, and Microsoft maintain dozens of data centers world-wide. Global services, such as Twitter and Facebook, are deployed on these data centers and

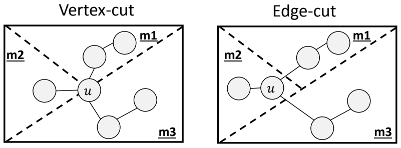


Fig. 1. Vertex-cut and Edge-cut.

produce large amounts of data (e.g., user friendship relations) that need to be analyzed. The standard solution to copy all the data to a single data center before performing data analytics is inefficient due to a large overhead [18], [19]. It is preferable to shift computation to the data, and select data to be moved carefully, leading to more efficient geo-distributed execution [19]. But here, network link costs can differ by orders of magnitudes (cf. Fig. 2d). These heterogeneous network costs should be considered when partitioning the graph.

To overcome these limitations, we developed GrapH, a graph processing system for distributed, in-memory data analytics on graph-structured data. GrapH is aware of both dynamic vertex traffic during execution and underlying network link costs. Considering this information, it adaptively partitions the graph during runtime to minimize overall communication costs by systematically *avoiding frequent communication over expensive network links*. In particular, the contributions of this paper are as follows:

- A fast partitioning algorithm, named H-load, and a fully distributed edge migration strategy for runtime refinement, named H-move, solving the dynamic vertex traffic- and network-aware partitioning problem.
- A graph processing system named GrapH enabling network-aware, geo-distributed execution of graph algorithms. In contrast to most state-of-the-art graph processing systems, we improve efficiency of multi-query execution by keeping the graph in memory across graph processing tasks.
- Evaluations on PageRank, and two important classes of graph algorithms: subgraph isomorphism to find arbitrary subgraphs in the graph, and cellular automaton to simulate social movement patterns of people in Beijing. We show, that GrapH reduces communication costs by up to 60% compared to state-of-the-art partitioning methods.

II. Preliminaries and Problem Formulation

In this section, we provide background information about the graph execution model and standard vertex-cut partitioning. Then, we present the network- and traffic-aware dynamic partitioning problem to be addressed in this paper.

A. Preliminaries

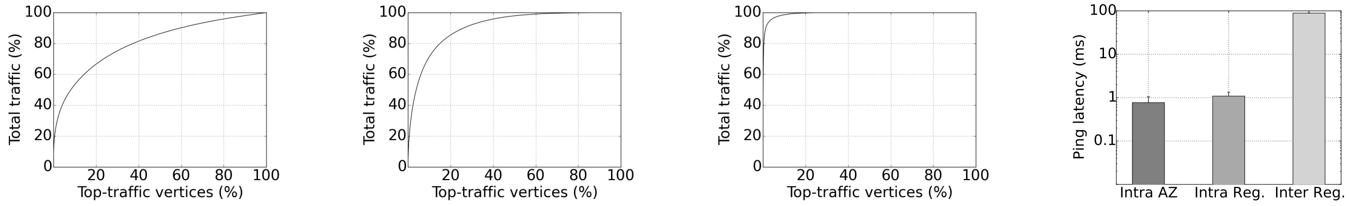
We assume a widely-used distributed vertex computation model similar to PowerGraph [6], where computation is organized in *iterations*. In each iteration, the system executes the user-defined vertex function for all *active* vertices and waits for termination (synchronized model). The vertex function operates on user-defined vertex data and consists of three phases, **G**ather, **A**pply and **S**catter (GAS). In the gather phase, each vertex aggregates data from its neighbors into a *gathered sum* σ (e.g., a union of all neighboring vertex data). In the apply phase, a vertex changes its local data according to σ. In the scatter phase, a vertex activates neighboring vertices for future execution in the next iteration. For example, in PageRank each vertex has vertex data $rank \in \mathbb{R}$. The gathered sum σ is the sum over all neighboring vertices' $rank$ values. A vertex changes its vertex data according to $rank = 0.15 + 0.85 * \sigma$ and activates all neighbors, if $rank$ has changed more than a certain threshold.

Large real-world graphs have billions of vertices and the sequential execution of all vertex functions on a single machine is not scalable. In order to parallelize execution, the graph has to be distributed onto multiple machines by cutting it through edges or vertices (*edge-cut* or *vertex-cut*). Vertex-cut has superior partitioning properties for real-world graphs with power-law degree distribution such as the Twitter or Facebook graph [6]. Thus, we use vertex-cut in this paper. In Fig. 1, we divided the graph into three parts using both strategies. As we can see, vertex-cuts distribute edges across machines and make vertices span multiple machines, each having its own *vertex replica*. The set of machines, where vertex u is replicated, is denoted as *replica set* R_u (e.g., the set $\{m1, m2, m3\}$ for vertex u). With this, we can define the *cut-size* as the total number of vertex replicas.

Inter-machine communication happens only in the form of *vertex traffic* between replicas. For instance, if all neighbors of vertex v are on the same machine, no inter-machine communication is needed because all neighboring data can be accessed locally. However, if vertex v is distributed, replicas have to communicate to access neighboring data by exchanging the gather, apply, and scatter messages. One dedicated replica, the *master* $\mathcal{M}_v$, initiates the distributed vertex function execution and keeps vertex data consistent on other replicas denoted as *mirrors*. More precisely, there are three types of communication.

First, a master sends a *gather request* to each mirror; in reply each mirror sends back a *gather response* containing an aggregation of local neighboring data (e.g., a sum of all local $rank$s for PageRank). We denote the number of bytes, exchanged in the gather phase between the master of vertex v and a mirror r in iteration i as $g_r^v(i)$. Second, after computing the new vertex data in the apply phase, the master sends a *vertex data update* to all mirrors (e.g., the new $rank$). The size of this message, $a^v(i)$, depends on the local vertex data on the master and can vary significantly. Third, in order to schedule neighbors of v for future execution, *scatter requests* of constant size s are exchanged between master and mirrors. With this, we can define average vertex traffic $t^v(i)$ per replica of vertex v in iteration i (having replica set $R_v(i)$).

$$t^v(i) = \frac{1}{|R_v(i)|} \sum_{r \in R_v(i)} (g_r^v(i) + a^v(i) + s) \qquad (1)$$



(a) (b) (c) (d)

Fig. 2. (a)-(c) Distribution functions of vertex traffic for PageRank, subgraph isomorphism, and cellular automaton. (d) Latency between machines intra-Availability Zone (AZ), inter-AZ and intra-region, and inter-region.

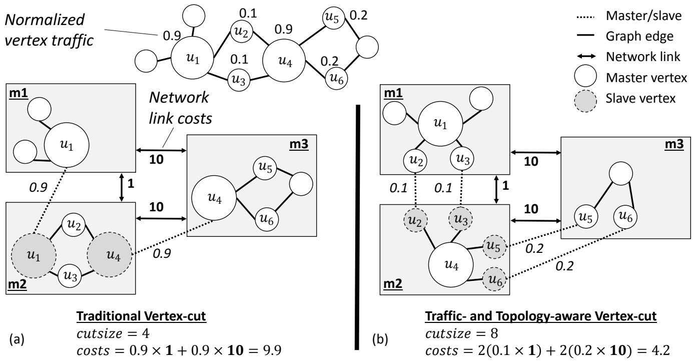


Fig. 3. (a) Vertex-cut minimizing replication degree. (b) Network- and traffic-aware vertex-cut minimizing costs.

B. Network- and Traffic-aware Vertex-cut

We now show heterogeneity of vertex traffic and the network and explain how we incorporated those into the vertex-cut partitioning problem.

In general, vertex traffic and network costs are heterogeneous. In Fig. 2(a)-(c), we show vertex traffic heterogeneity for three algorithms: PageRank, subgraph isomorphism and cellular automaton (cf. Sec. IV for details about the algorithms). The graph shows the x/y distribution: x% of the top-traffic vertices are responsible for y% overall traffic. In our evaluations, PageRank is 20/65 distributed, because of different convergence behaviors of vertices as mentioned previously in Sec. I. Subgraph isomorphism is more extreme with a 20/84 distribution, because some vertices match more patterns than others. Cellular automaton is highly imbalanced (20/100), because vertices simulating unpopular regions in Beijing have almost zero traffic (cf. Sec. IV). Besides vertex traffic, machine communication is also subject to significant variations in terms of bandwidth and latency, even in a single datacenter [9], [14], [20]. For Amazon EC2 machines, we show orders-of-magnitude variations of latency (cf. Fig. 2d). Likewise, many cloud providers charge variable prices for intra and inter data center communication. For instance, Amazon charges nothing for communication within the same availability zone (AZ), while communication across AZs is 0.01$ and across regions is 0.02$ per GB outgoing traffic.

Therefore, efficient partitioning techniques should utilize these diverse costs. For example in Fig. 3(a), vertices are annotated with their (normalized) vertex traffic, also indicated by the vertex size. Machines $m1 - m3$ communicate via network links with different costs, given by the weights in bold. The vertex-cut leads to distributed vertices u_1 and u_4, both having traffic 0.9. We define *communication costs* via a network link as the cost of the link multiplied by the traffic sent over this link. The summed communication costs over all network links are the *total communication costs*. In the example, total communication costs are $(0.9*1)+(0.9*10) = 9.9$. Here, traditional vertex-cut leads to minimal cut-size, but high communication costs, because high-traffic vertices u_1 and u_4 send many messages over the network. To this end, we introduce the network- and traffic-aware dynamic vertex-cut partitioning. The idea is to cut the graph on the low-traffic vertices to decrease across-machine communication. In Fig. 3(b), we minimize communication by cutting the graph on vertices u_2,u_3 and u_5,u_6. This increases the cut-size, but decreases overall communication costs. Note that this partitioning could be improved even further by exploiting heterogeneous network link costs. Suppose, the subgraph assignments of m1 and m3 were swapped. Then, the (relative) high traffic vertices u_5,u_6 communicate over the inexpensive link (m1,m2), decreasing overall communication costs to $2(0.2*1)+2(0.1*10) = 2.4$.

Problem Formulation: Let $G = (V, E)$ be a directed graph with the vertex set V and edge set $E \in V \times V$. Let $M = \{m_1, ..., m_k\}$ be the set of all participating machines. The network cost matrix $T \in \mathbb{R}^{k \times k}$ assigns a cost value to each pair of machines (e.g., monetary costs for sending one byte of data). The set of all iterations needed for the graph processing task be $I = \{0, 1, 2, ...\}$. Vertex traffic for all iterations $i \in I$ and vertices $v \in V$ is denoted as $t^v(i)$. The assignment function $a : E, I \rightarrow M$ specifies the mapping of edges to machines in a given iteration. The *replica set* of vertex v in iteration i based on assignment function a is denoted as $R^a_v(i)$. It represents the set of machines maintaining a replica of v: $R^a_v(i) = \{m | a((u, v), i) = m \vee a((v, u), i) = m\}$. In the following, we denote R_v to be v's replica set under the current assignment.

Our goal is to *find an optimal dynamic assignment of edges to machines minimizing overall communication costs*:

$$a_{opt} = \underset{a}{\operatorname{argmin}} \sum_{i} \sum_{v \in V} \sum_{m \in R^a_v(i)} t^v(i) \; T_{m,\mathcal{M}_v} \qquad (2)$$

The load $L_m(i)$ of machine m in iteration i is defined as the summed vertex traffic over all vertices replicated on m in iteration i. To balance machine load, we require for each iteration i and machine m (having vertices V_m) that load deviation is bounded by a small balancing factor $\lambda > 1$:

$$L_m(i) = \sum_{v \in V_m} t^v(i) < \lambda \frac{\sum_{v \in V} t^v(i)}{|M|}. \quad (3)$$

Hardness: The dynamic network- and traffic-aware partitioning problem is NP-hard.

Proof sketch: Reduce the NP-hard balanced vertex-cut problem to Eq. 2. Set input: $I = \{1\}$, $t^v(i) = 1$, $T_{m_1,m_2} = 1$. By, $a_{opt} = \text{argmin}_a \sum_i \sum_{v \in V} \sum_{m \in R_v^a(i)} 1*1 = \text{argmin}_a \sum_{v \in V} |R_v^a|$, Eq. 2 becomes the network- and traffic-*unaware* vertex-cut problem, which is NP-hard (e.g., [21]). □

III. Algorithms for Network- and Traffic-aware Partitioning

In this section, we present our novel algorithms addressing the network- and traffic-aware partitioning problem. We developed two methods: a partitioning algorithm called **H-load** for pre-partitioning the graph, and a dynamic algorithm called **H-move** for runtime refinement using migration of edges.

A. H-load: Initial Partitioning

Graph processing systems have to pre-partition the graph, so that each machine can load its partition into local memory. To this end, we developed H-load, a fast pre-partitioning algorithm that consists of two phases. First, it partitions the graph using a vertex-cut algorithm ignoring network heterogeneity. Second, it determines a cost-efficient mapping from partitions to machines. We describe these two phases in the following.

1) Initially, our goal is to find a reasonable partitioning of the graph into k balanced parts, ignoring concrete mapping of partitions to machines. We assume a streaming setting: the graph is given as a stream of edges $e_1, e_2, ..., e_{|E|}$ with $e_i \in E$ and we consecutively read and assign one edge at a time to a partition until there are no more edges to read. Our method is similar to the vertex-cut algorithm of PowerGraph [6], which greedily reduces replication degree of vertices. However, the PowerGraph pre-partitioning leads to relatively homogeneous total traffic between each pair of partitions. But to exploit heterogeneity of network costs, we also require heterogeneity of inter-partition traffic: partitions exchanging more traffic should be mapped to machines with low-cost network links. More precisely, the network cost matrix T often consists of several *clusters* of machines that have low intra-cluster and high inter-cluster costs (e.g., EC2 machines running in different availability zones). The number of clusters c can be determined from the matrix T using well-established clustering methods (e.g., [22]).

We assign each edge (u, v) to a partition p to be determined as follows. If there exists no replica of u or v on any partition, assign (u, v) to the least loaded partition. If there are partitions containing replicas of u *and* v, assign (u, v) to the least loaded of those partitions. Otherwise, a new replica has to be created, because there is no partition containing both replicas of u and v. For example, if we place the edge on a partition p that already has a replica of u, we have to create a new replica of v. We choose partition p, such that the new replica preferentially lies in the same cluster as already existing replicas. With this method, our algorithm ensures a clustered traffic behavior: partitions in the same cluster share the same replicas and thus are expected to exchange more traffic than partitions in different clusters. In the next phase, we try to find a good mapping of partitions to machines.

2) Now, we try to find a mapping of the $|M|$ partitions to $|M|$ machines while minimizing overall communication costs. In order to minimize these costs, an optimal mapping of partitions to machines would assign two partitions with higher inter-partition traffic to machines connected via a low-cost network link. This is an instance of the well-known quadratic assignment problem: map $|M|$ factories (i.e., partitions) to $|M|$ locations (i.e., machines), so that the mapping has minimal costs of factories sending their goods to other factories. Each two factories exchange a certain amount of goods and a cost function associates a cost value to each pair of locations (i.e., the network link matrix T). The assignment should have minimal overall costs of factories sending their goods to other factories (i.e., minimal communication costs). We used the iterated local search algorithm of Stützle et al. [23] which greedily minimizes overall costs. Initially, partitions are randomly mapped to machines. Then the algorithm iteratively improves the total costs using the following method. Find two machines, such that an exchanging of partition assignments would result in lower total communication costs. For example in Fig. 3, exchanging partition assignments of machine $m1$ and $m3$ results in lower total communication costs. If an improvement is found, it is applied immediately. In order to address convergence to local minima, we perturb a local optimal solution by randomly exchanging two assignments. Note, that this algorithm is computationally feasible, because it runs on a relatively small problem set with size $|M| << |V|$. Clearly, the above method assumes that the traffic exchanged between each pair of partitions is known (i.e., cumulative traffic exchanged between vertex replicas shared by each pair of partitions). This information can be determined from the previous executions of the GAS algorithm. Otherwise, homogeneous traffic between vertex replicas is assumed.

B. H-move: Distributed Migration of Edges

The H-load algorithm is suitable for a static network-aware and traffic-aware partitioning. However, often the vertex traffic changes dynamically at runtime. To this end, we developed the distributed edge-migration algorithm *H-move* solving the dynamic heterogeneity-aware partitioning problem. The idea is that each machine locally reduces the costs of graph processing by migrating edges to distant machines. In view of this, we define the term *bag-of-edges* as the set of edges to be migrated. Machines exchange bag-of-edges in parallel after each GAS iteration. Finally, if no further improvements can be performed, migration is switched off.

问题
1.数据迁移真的就不可行吗（对于迭代计算呢）
2.traffic 是怎么计算出来的？
3.容量的设置capacity

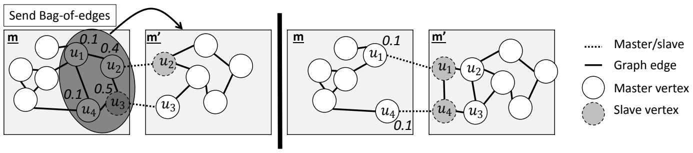


Fig. 4. Example of bag-of-edges migration to reduce inter-machine traffic.

Approach overview: The overall migration strategy is given by Alg. 1. After activation of the migration algorithm (line 1), machine m first selects partner machine m' (line 2) and then calculates the bag-of-edges to be send to m' (line 3). In order to prevent inconsistencies due to parallel updates on the distributed graph, machine m requests locks for all vertices in the bag (line 4). Afterwards, m updates the bag to contain only those edges, whose endpoint vertices could be locked (line 5) and determines, whether sending the updated bag results in lower total communication costs (line 6). Recap, that communication costs are defined as the global sum over all vertex traffic values multiplied by the network costs between vertex replicas (cf. Eq. 2). When sending the bag-of-edges, these communication costs change due to modifications of the vertex replica sets. To calculate Δc in line 6, machine m considers both: the migration overhead c_+ of sending the bag, as well as the decrease of graph processing costs c_- when improving the partitioning. If Δc is negative, the bag-of-edges is migrated to m'. Finally, machine m releases all held locks in line 9.

Algorithm 1 Migration algorithm on machine m.

```
1: $waitForActivation()$
2: $m' \leftarrow selectPartner()$
3: $b \leftarrow bagOfEdges(m')$
4: $lock(b)$
5: $b \leftarrow updateLocked(b)$
6: $\Delta c \leftarrow c_+ - c_-$
7: if $\Delta c < 0$ then
8:     $migrateBag(b)$
9: $releaseLocks(b)$
```

We give an example of this procedure in Fig. 4. Two machines m and m' have replicas of high-traffic vertices u_2 and u_3. In order to reduce communication costs, m decides to send the bag-of-edges $b = \{(u_1, u_2), (u_2, u_3), (u_3, u_4), (u_4, u_1)\}$ to m'. Machine m' receives b and adds all edges in b to the local subgraph. The right side of the Fig. 4 shows the final state after migration of b. Now, low-traffic vertices u_1 and u_4 are cut leading to less inter machine traffic. In the following, we describe the proposed migration approach (cf. Alg. 1) in more details.

1) Selection of partner and bag-of-edges: Which machine is suitable as exchange partner? Intuitively, two machines sharing high-traffic replicas are strong candidates for exchanging bag-of-edges, because improving their partitioning can potentially reduce the overall communication costs. To this end, each machine m maintains a list of potential exchange partners (with decreasing priority). This list is computed by sorting neighboring machines w.r.t. the total amount of exchanged traffic. On each round of Alg. 1, the top-most machine m' is selected as an exchange partner and removed from the list. Once the list is empty, it is recomputed as mentioned above.

Now, we determine the maximal size of the bag-of-edges to be sent to m' in order to ensure balanced machine load (cf. Eq. 3). Therefore, we introduce the notion of *capacity* of a machine m', i.e., the maximum amount of additional load, machine m' can carry. Capacity is defined as half the difference of loads L_m and $L_{m'}$ of the sending and the receiving machine: $C = (L_{m'} - L_m)/2$. To learn about the current load $L_{m'}$ of machine m', machine m sends a request to m'. Using the capacity, machine m can balance the loads by only sending edges, such that the deviation of the two machines traffic values is still bounded. For example, if sending the bag-of-edges results in a new replica of vertex v on m', this increases load of m' by the vertex traffic of v. If this violates load balancing between m and m', machine m will not include v into the bag.

Once the exchange partner is selected and we know its capacity, we determine a bag-of-edges to be send. Selecting a suitable bag-of-edges is crucial for optimizing communication costs and migration overhead. Theoretically, the perfect bag-of-edges could be any subset out of p edges on a machine (i.e., 2^p subsets). In order to keep the migration phase lean, we developed a fast heuristic to find a bag-of-edges improving communication costs (cf. Alg. 2). Initially, machine m determines the set of candidate vertices, those replicated on both machines, because they are responsible for all the traffic between m and m'. Machine m sorts the candidates by descending traffic in order to focus on the high-traffic vertices first (line 3). Then, m iterates the following steps until m' has no more capacity. It checks for the top-most candidate vertex (line 5), whether sending all adjacent edges results in lower total communication costs of the overall graph processing (line 6-8, cf Sec. III-B2). If the total communication costs would decrease when sending the edges, machine m adds them to the bag (line 9).

Algorithm 2 Determining the bag-of-edges to exchange.

```
1: bagOfEdges($m'$)
2: $bag \leftarrow []$
3: $candidates \leftarrow sort(adjacent(m'))$
4: while $hasCapacity(m', bag)$ do
5:     $v \leftarrow candidates[0]$
6:     $b \leftarrow \{(u, v), (v, u) | u \neq v\}$
7:     $\Delta c \leftarrow c_+ - c_-$
8:     if $\Delta c < 0$ then
9:         $bag \leftarrow bag + b$
10: return $bag$
```

2) Calculation of costs: Clearly, migrating bag-of-edges b from one machine to another is only beneficial, if it results in lower overall costs (i.e., line 6 in Alg. 1, and line 7 in Alg. 2). In general, two types of costs have to be considered in calculating the resulting overall costs: *investment costs* and *payback costs*. Investment costs represents the overhead for migrating the bag-of-edges and should be avoided. Payback costs are the saved costs after migrating b in the form of less future inter-machine traffic. In the following, we formulate both costs.

Investment costs: After sending b to m', m can remove isolated replicas that have no local edges anymore (Fig. 4 vertices u_2, u_3). If machine m is the master $\mathcal{M}_v$ of a vertex v to be removed, i.e., $\mathcal{M}_v = m$, we have to select a new master after removing v from m. We set the partner machine m' to be the new master of v: $\mathcal{M}'_v = m'$. On the other hand, some vertices may not exist on m' leading to creation of new replicas (Fig. 4 vertices u_1, u_4). In both cases, the replica set R_u of a vertex u might have changed (i.e., remove m or add m' to R_u). Then m has to send an update to all machines in R_u with the new vertex replica set, denoted as R'_u. Additionally, when creating a new replica on m', machine m has to send the state of v, i.e., vertex data and meta information such as the vertex id. This can be very expensive for large vertex data, and should be taken into account when deciding whether to migrate a bag-of-edges. Together, the investment costs are the sum of three terms. The first term calculates the costs of sending the bag b to m'. The second term calculates the costs of sending new replicas to m', if needed. The third term calculates the costs of updating machines in all replica sets that have changed.

$$c_+ = \sum_{e=(u,v)\in b} \beta(e) T_{m,m'} + \sum_{u\in V_b} \delta(u)\beta(u) T_{m,m'} + \sum_{u\in V'_b, r\in R_u \cup R'_u} \beta(R_u) T_{m,r}, \tag{4}$$

where the function $\beta(x)$ returns the number of bytes needed to encode x (to be sent over the network). And the indication function $\delta(u)$ returns 1, if machine m' has no local replica of u, otherwise 0. V_b is the set of all vertices in bag b, while V'_b is the set of all vertices whose replica sets will change when sending the bag-of-edges b to m'.

Payback costs: we can also save costs when sending a bag from m to m'. Suppose the replication degree decreases because of sending (u,v), i.e., $|R'_u| < |R_u|$ or $|R'_v| < |R_v|$. Then, we save *for each iteration* (starting from the current iteration i_0) the costs of exchanging gather, apply, and scatter messages across replicas, i.e., the vertex traffic $t^v(i)$ of vertex v in iteration i. The theoretical exact payback costs are given by the following formula that calculates for all future iterations and each vertex in the bag the difference of the new costs and the old costs of v's replica set.

$$c^*_- = \sum_{i>i_0} \sum_{v\in V_b} \Big(\sum_{r\in R'_v} t^v(i) T_{r,\mathcal{M}'_v} - \sum_{r\in R_v} t^v(i) T_{r,\mathcal{M}_v} \Big) \tag{5}$$

Here, it is assumed that vertex traffic is known for all future iterations. This is not the case in real systems. Therefore, we describe next, how we calculated the payback costs in uncertainty about the real future traffic values.

In order to estimate payback costs, we have to predict vertex traffic for future iterations. More formally, given vertex traffic $t^v(0), t^v(1), ..., t^v(i)$, we estimate traffic values $t^v(i+1), ..., t^v(|I|)$. The prediction should be quick, have low computational overhead, and low memory requirements, because we have to predict vertex traffic in each migration phase for millions of vertices. We investigate three well-known methods for time series prediction of the next traffic value $t^v(i+1)$, that fit to our requirements [24]. The first method is *most recent value* (naive) taking the last traffic value as prediction for the next traffic value: $\hat{t}^v(i+1) = t^v(i)$. The second method is *incremental moving average* (MA) with the idea to use the moving average of the last w observations, while not storing the values in the window: $\hat{t}^v(i+1) = \frac{\hat{t}^v(i)(w-1)+t^v(i)}{w}$. The third method is *incremental exponential average* (EA) that calculates the estimation based on the old estimation and the last traffic value: $\hat{t}^v(i+1) = \alpha t^v(i) + (1-\alpha)\hat{t}^v(i)$. The parameter $\alpha \in [0,1]$ specifies the amount of decaying older traffic values and thus, the importance of recent traffic. We assume $t^v(0) = 0$ for all methods. In Sec. IV, we have compared overall system performance for these methods with different parameter choices, i.e., window sizes w and decay parameters α.

With the previous methods, we can determine the vertex traffic estimation for the next iterations. However, Eq. 5 expects a vertex traffic value for all future iterations. In general, accuracy of the predicted vertex traffic $\hat{t}^v(i)$ can decrease with increasing i, because vertex traffic patterns may change over time. Therefore, we introduce a factor μ, which represents the minimum number of iterations, we expect to save those costs. This parameter can be used to specify the aggressiveness with which migration should be performed. Together, our estimated payback costs are the following.

$$c_- = \mu \sum_{v\in V_b} \Big(\sum_{r\in R'_v} \hat{t}^v(i+1) T_{r,\mathcal{M}'_v} - \sum_{r\in R_v} \hat{t}^v(i+1) T_{r,\mathcal{M}_v} \Big) \tag{6}$$

3) *Graph consistency:* When two machines independently send edges and change replica sets of vertices, inconsistencies of the data graph can arise. In Fig. 5, we give an example. Machines $m1$-$m4$ maintain a replica of vertex u. Suppose machine $m4$ wants to send edge (u, v_1) to machine $m3$ and machine $m1$ edge (u, v_2) to $m2$. After sending the edge, machines $m4$ ($m1$) removes the vertex replicas of u with no further local edges. Therefore, machine $m4$ ($m1$) has to update all other machines having a replica of v with the new replica set. Machine $m4$ sends the new replica set $\{m1, m2, m3\}$ to all machines, while machine $m1$ sends $\{m2, m3, m4\}$. The machines can receive these updates in different orders leading to inconsistent views on the replica sets (lost update problems). Only a sequential update would result in a consistent state (machine $m4$ changes the replica set *before* machine $m1$). To guarantee sequential updates during edge migration, we lock endpoint vertices in the bag-of-edges to be sent to the partner machine. We use a simple master locking scheme, i.e., each machine intending to change vertex u sends a locking request to $\mathcal{M}_u$. If vertex u is already locked, the master machine returns $false$, otherwise it locks u and returns $true$. However, this also implies that not all locks may be acquired. Therefore,

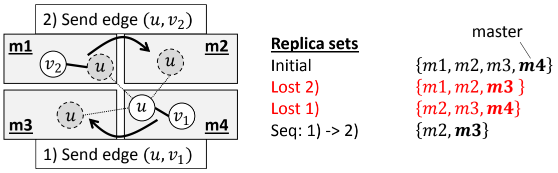


Fig. 5. Lost update problem for parallel edge migration.

we exclude edges from the bag-of-edges for which not both endpoints have been locked successfully (see line 5 in Alg. 1). Only after the locking machine has successfully implemented the bag-of-edge exchange, it releases all locks.

IV. Implementation and Evaluations

In this section, we present implementation details and evaluations of our graph processing framework $GrapH$.

A. System Architecture

We have implemented GraphH in the Java programming language (with 10,000 lines of code). GrapH consists of a master machine and multiple client machines that perform the graph analytics. The master receives a sequence of graph processing queries $q_1, q_2, q_3, ...$ consisting of user specified GAS algorithms. All machines communicate with each other directly via TCP/IP. In contrast to many traditional graph processing frameworks, GrapH allows the graph to stay in memory for multiple graph processing queries (PowerGraph and GraphLab reload the whole graph into distributed memory for each GAS algorithm). This improves efficiency of executing multiple short algorithms in sequence due to less reloading overhead. To initialize the vertex data after each GAS execution, the user can define a vertex initialization function, specifying whether the vertices should be re-initialized with fresh data, or take the most recent data. Hence, graph algorithms can build upon past algorithms. For example, we can execute the PageRank algorithm and use the ranks in a link prediction algorithm executed afterwards [25]. Another advantage of leaving the graph in memory is, that subsequent queries can make use of the improved graph partitioning. As our migration strategy adapts to changing vertex traffic, a possible abrupt change in vertex traffic patterns can be handled smoothly by our system.

B. Graph Algorithms

For our evalutations, we have implemented three important graph algorithms: PageRank (cf. [6]), subgraph isomorphism, and social simulations via agent-based cellular automaton. We denote these three algorithms as PR, SI, and CA for the rest of the paper. For SI and CA there is, to the best of our knowledge, no implementation using the GAS API, so we have designed our own algorithms. For implementation details on these algorithms, we refer the reader to [26].

Subgraph isomorphism is an NP-complete graph problem. It can be used to query a graph for certain patterns (*subgraphs*). Examples are simple queries in Facebook Graph Search ("Has a friend of mine been in that restaurant?") or complex queries in community detection ("Is there a k-clique in the graph?"). More precisely, given a graph $G = (V, E)$ and a graph pattern $P = (V_P, E_P)$, subgraph isomorphism is the problem of finding subgraphs $G_{sub} = (V_{sub}, E_{sub})$, with $V_{sub} \subseteq V, E_{sub} \subseteq E$, that are isomorphic to the graph pattern P. Each graph vertex can have an optional label (e.g., an importance weight such as a PageRank value). In this case, the labeled subgraph isomorphism additionally requires both vertices in V_{sub} and V_P to have matching labels (cf. [4]).

Name	$\lvert V\rvert$	$\lvert E\rvert$
Gnutella	8,000	26,000
Facebook	4,000	88,000
WikiVote	7,000	103,000
Twitter	81,000	1,700,000
GoogleWeb	800,000	5,000,000
TwitterLarge	41,000,000	1,400,000,000

TABLE I
Real-world graphs for evaluations.

Cellular automaton is a powerful and well-establish model of simulation, where the problem space is expressed as a grid of cells, each cell having a finite number of states. A cell iteratively calculates its own state based on the states of neighboring cells. Many complex simulation problems can be modeled as cellular automata and computation is easily parallelizable using the GAS programming abstraction for recent graph processing systems. We implemented an agent-based variant for simulating movements of people in Beijing using real-world movement data[1] [1].

C. Evaluations

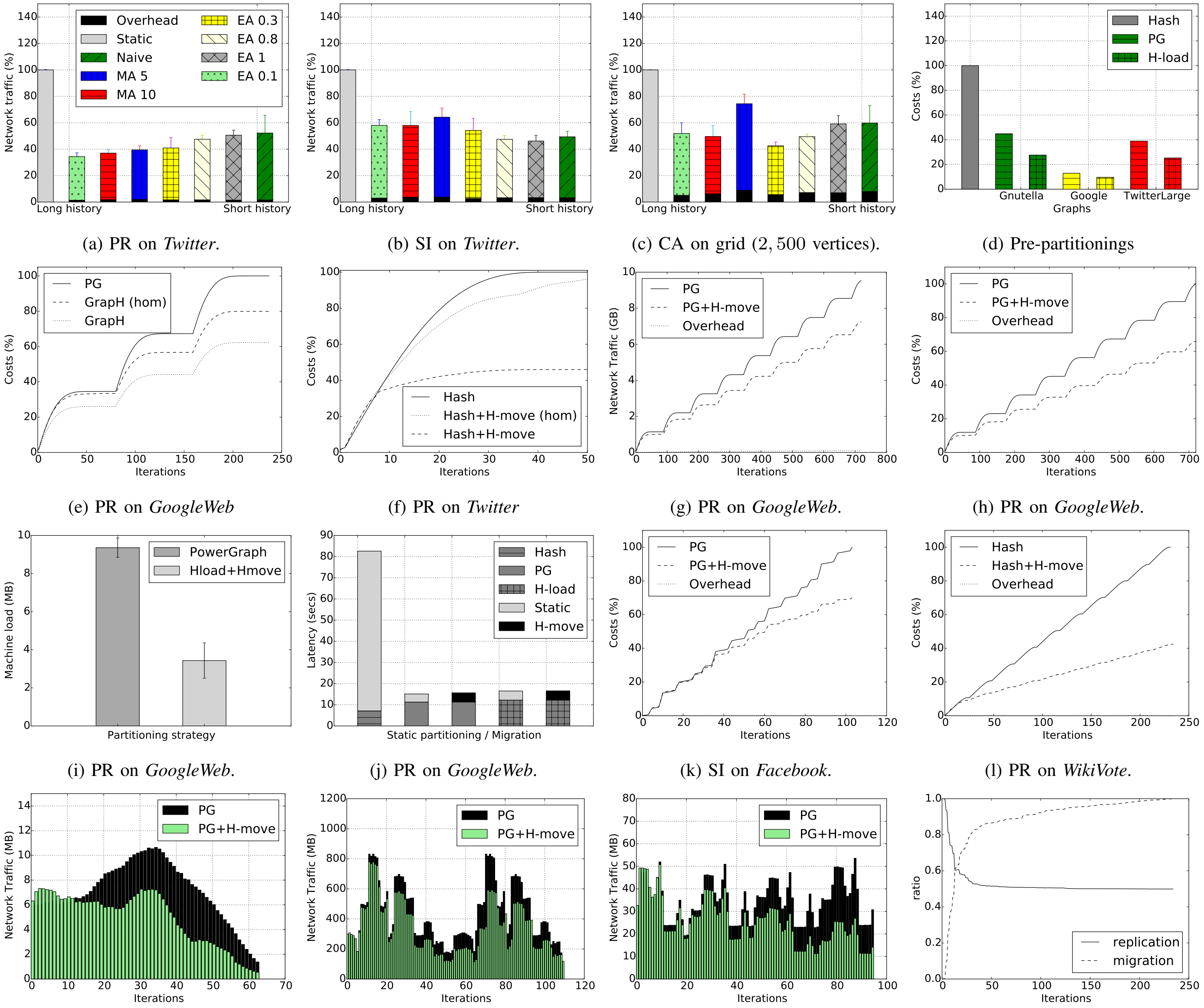
In the following, we present evaluations for GraphH on two computing clusters for three different algorithms, i.e., PageRank, subgraph isomorphism, and cellular automaton, on several real-world graphs[2] with up to 1.4 billion edges given in Tab. I. We compared our migration strategies against state-of-the-art static vertex-cut partitioning approaches: hashing of edges (Hash) and PowerGraph (PG) [6], [7].

Evaluation setup: The homogeneous computing cluster consists of 12 machines, each with 8 cores (3.0GHZ) and 32GB RAM, interconnected with 1 Gbps ethernet. We have also performed experiments in the (heterogeneous) cloud using 8 geographically distributed Amazon EC2 instances (1 virtual CPU with 3.3 GHz and 1 GB RAM) that are distributed across two regions, US East (Virginia) and EU (Frankfurt), and four different availability zones. As network costs between these instances, we used the real monetary costs charged by Amazon (cf. Sec. II).

Performance of prediction methods: We expected that choosing the right prediction method is important for overall performance of our system. Therefore, we compared three prediction methods (cf. Sec. III): last value (naive), moving average (MA) with window sizes 5 and 10, and exponential averaging (EA) with $\alpha = 0.1, 0.3, 0.8, 1.0$. We evaluated the reduction of total network traffic for algorithms PageRank (PR), subgraph isomorphism (SI), and cellular automaton

[1] http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/

[2] http://konect.uni-koblenz.de/networks/twitter, http://snap.stanford.edu/data



(a) PR on *Twitter*.

(b) SI on *Twitter*.

(c) CA on grid (2, 500 vertices).

(d) Pre-partitionings

(e) PR on *GoogleWeb*

(f) PR on *Twitter*

(g) PR on *GoogleWeb*.

(h) PR on *GoogleWeb*.

(i) PR on *GoogleWeb*.

(j) PR on *GoogleWeb*.

(k) SI on *Facebook*.

(l) PR on *WikiVote*.

(m) CA on grid (10^5 vertices)

(n) SI on *Twitter*.

(o) SI on *Facebook*.

(p) PR on *WikiVote*.

Fig. 6. (a)-(c) Prediction methods influence migration efficiency. (d) H-load pre-partitioning improves costs. (e)-(f) Network- and traffic-awareness improves costs. (g),(m)-(o) Network traffic reduces over time. (h),(k),(l) Communication costs reduce over time. (i) Load is reduced and balanced. (j) Latency remains stable. (p) Cut-size decreases.

(CA), compared to the hash partitioned graph (here denoted as *Static*). The results are shown in Fig. 6a-c. The position of the bars from left to right reflects the size of the considered history for prediction in descending order. For example *naive* considers only the last value, so we plotted it on the right. As we can see, all prediction methods meet the goal of minimizing overall network traffic. However, no method leads to consistently better results for all algorithms. We attribute this to the different stability of vertex traffic patterns. For example in PR, considering a longer history shows better results, because vertex traffic patterns remain stable over time. Nevertheless, considering a large history in SI actually harms performance, because we issued multiple short-lived queries leading to different vertex traffic patterns. For CA, there is no such clear trend, apparently because of the sudden changes of vertex traffic, when agents move to neighboring cells. However, exponential averaging with parameter $\alpha = 0.3$ showed good results for all three algorithms, so we use this configuration in the following. A thorough study in this direction is left for future work.

Communication costs: The core idea of this paper is to consider network- and traffic-heterogeneity while constantly re-partitioning the graph during computation. In Fig 6f, we show total communication costs for one PR execution on *Twitter* in the cloud. We evaluated the accumulated graph processing costs over 50 iterations using three different partitioning methods: hashing of edges to machines (Hash), H-load and H-move ignoring heterogeneous traffic, and H-load and H-move considering heterogeneous vertex traffic. Taking heterogeneous vertex traffic into account greatly improves total communication costs by up to 50%. But how does network-awareness improve total communication costs? In Fig. 6e, we show for PageRank on *GoogleWeb*, that taking heterogeneous network into account during H-move improves total costs by further 25%. Furthermore, we show in Fig 6d, that our partitioning strategy H-load greatly improves total graph processing costs for three different graphs: *Gnutella*, *GoogleWeb*, and *TwitterLarge*. We assumed homogeneous vertex traffic and heterogeneous network costs in the cloud setting. Costs are reduced by 70-90% compared to Hash and by 25-38% compared to PowerGraph partitioning.

To learn about the generality of H-load and H-move, we have performed many evaluations on both computing infrastructures, for different graph algorithms and different real-world graphs. H-move consistently improves communication costs and network bandwidth usage. For instance, we have executed multiple PR algorithms on a slightly changing graph (remove 10^3 edges after each PR execution) in the cloud. In Fig. 6g, we plotted the total used bandwidth for this execution, while in Fig. 6h, we show the total communication costs. Bandwidth usage decreases by 25% and communication costs decreased by 33% compared to the already PowerGraph-prepartitioned setting (note, that the cloud has heterogeneous network costs, so the cost improvement can be higher than the bandwidth usage improvement). The costs and traffic for migration itself are very low and therefore not visible in the figures. We have also executed multiple queries of the complex SI algorithm on *Facebook*. In Fig. 6k, we can see the decrease of cummulative total costs for the PowerGraph partitioned system with and without migration. In Fig. 6m-o, we show the current number of bytes sent via the network for algorithms SI and CA, averaged over a sliding window of 10 iterations. Our migration strategy reduces total network traffic by up to 50% compared to PowerGraph partitioning. In Fig. 6l, we show communication costs for PR on the dynamic Wiki graph comparing our migration strategy with initial hashing. The total costs decreases by 60%, when migration is switched on. Overall, graph processing using H-move reduces total communication costs and bandwidth usage during runtime, independent from the concrete pre-partitioning. GrapH automatically decides, whether pre-partitioning is worth the additional overhead.

Cut-size: In Fig. 6p, we can see the improvement of the cut-size using H-move during PR on *WikiVote*. We plot for each iteration the current cut-size divided by the cut-size of a hash partitioned graph against the current migration overhead divided by the total migration overhead (CDF). During the first 50 iterations, there is an improvement of more than 50% in cut-size, followed by saturation. Therefore, migration is switched off after a certain amount of iterations.

Latency: The partitioning problem is computational hard and solving it during execution can be extremely expensive. However, our heuristic performs well, as we can see in Fig. 6j. We measured the total graph processing latency for one PR iteration, as well as the latency for pre-partitioning the graph using different strategies. Due to huge memory overhead, the hash-based approach leads to much higher latency in our setting. In fact, the execution lead to an insufficient-memory error (note that memory of our EC2 instances is relatively small). We can also see, H-move induces relatively little latency overhead compared to migration switched off ($0.01 - 0.13$ times). Experiments for SI on a PG partitioned graph show similar results of 0.04 times increased latency compared to static partitioning. We believe that the latency punishment of H-move can be further reduced by deciding during runtime, whether migration should take place or not.

Load balancing: GrapH maintains workload balance. In Fig. 6i, we show machine workload after one PR execution in the cloud (78 iterations). We have plotted the average machine workload for the PG partitioning and our H-load and H-move algorithms, as well as the deviation of the machines from this average workload. Using our migration algorithm, we can reduce total workload by more than 60%. Our method leads to a slightly higher workload imbalance, because we balance for (more volatile) vertex traffic, while PowerGraph balances the number of edges. However, even the machine with highest workload has still less workload than the least-loaded machine in PG partitioning.

V. Related Work

Recently, many systems for distributed graph processing have emerged that inspired our work. PowerGraph [6] suggests the GAS programming model and uses a distributed vertex computation strategy based on vertex-cut partitioning, because power-law degree distributed graphs have better vertex-cuts than edge-cuts. They provide a greedy streaming heuristic for static vertex-cut partitioning, which we used for comparison. Other streaming heuristics for vertex-cut exist, for example Petroni et al. [21] consider the vertex-degree in order to find a minimum vertex-cut. PowerLyra [27] extends PowerGraph by a partitioning strategy for hybrid-cuts: high-degree vertices are cut enabling a good partitioning for power-law graphs and edges of low-degree vertices are cut decreasing replica communication overhead. These strategies minimize the replication degree, but ignore diverse and dynamic vertex traffic.

On the other hand, the graph processing systems Mizan [28] and GPS [11] propose adaptive edge-cut partitioning by migration of vertices. Mizan also considers the real traffic sent via each edge. Vaquero et al. [29] apply edge-cut to changing graphs, to avoid costly re-execution of static partitioning algorithms using a decentralized algorithm for iterative vertex migration. While these systems adapt to changing graphs or traffic behaviors, they do not consider network topology and migration costs and can not be applied to vertex-cut partitioning.

Surfer [14] tailors graph processing to the cloud by considering bandwidth unevenness across machines to map partitions with a high number of inter-partition links to machines connected via high-bandwidth networks, however they assume the number of bytes sent via each edge is homogeneous. GraphIVE [15] searches for an unbalanced k-way vertex-cut for machines with heterogeneous computation and communication capabilities, in order to put more work to more powerful machines. Therefore, they search the optimal number of edges for each machine. This approach is orthogonal to our proposed heterogeneity-aware partitioning algorithms. Xu et al. [20] consider network and vertex weights to find a static edge-cut with minimal communication costs. They do not consider adaptive vertex-cut partitioning, also the vertex weights reflect only the number of executions, not the real arising traffic. Zheng et al. [30] propose an architecture-aware graph re-partitioning method that also considers the amount of communication going over each edge and the costs of migrating a vertex. Unfortunately, it can not be used for vertex-cut partitioning and the GAS execution model.

General data processing in the geo-distributed setting is addressed by Pu et al. [18] and Jayalath et al. [19]. They argue that aggregating geographical distributed data into a single data center can significantly hurt overall data processing performance. Hence, they share our idea of saving overall costs in geo-distributed data analytics, but focus on MapReduce-like computations. Choreo [9] points out that considering network heterogeneity, even in a single data center, for an optimal task placement improves overall end-to-end data analytics performance. Their algorithms place communicating tasks, such that most data travels over fast network links. However, optimization of the graph partitioning is oblivious to task placement. Hence, it could be applied on top of our system.

VI. Conclusion

Modern graph processing systems use vertex-cut partitioning due to its superiority of partitioning real-world graphs. Those partitioning methods minimize the cut-size, which is expected to be the dominant factor for communication costs. However, the underlying assumptions of uniform vertex traffic and network costs are wrong for many applications. To this end, we proposed GrapH, a graph processing system taking dynamic vertex traffic and diverse network costs into account to adaptively minimize communication costs of the vertex-cut during runtime. GrapH outperforms PowerGraph's static and heterogeneity-blind vertex-cut algorithm by more than 60% in communication costs.

References

[1] T. Suzumura, C. Houngkaew, and H. Kanezashi, "Towards billion-scale social simulations," in *Proc. Winter Simul. Conf.*, ser. WSC, 2014.

[2] A. Pascale, M. Nicoli, and U. Spagnolini, "Cooperative bayesian estimation of vehicular traffic in large-scale networks," *Intell. Transp. Syst., IEEE Transactions on*, vol. 15, no. 5, pp. 2074–2088, 2014.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web." 1999.

[4] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *Proc. VLDB Endow.*, 2011.

[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 *Proc. ACM SIGMOD*, 2010.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, 2012.

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: graph processing in a distributed dataflow framework," in *Proc. USENIX OSDI*, 2014.

[8] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM CCR*, 2011.

[9] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware task placement for cloud applications," in *Proc. of the 2013 conf. on Internet measurement conf.* ACM, 2013, pp. 191–204.

[10] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," *SIGCOMM CCR*, 2009.

[11] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proc. Int. Conf. Scient. and Stat. Database Manag.* ACM, 2013.

[12] K. Ten Tusscher, D. Noble, P. Noble, and A. Panfilov, "A model for human ventricular tissue," *American Journal of Physiology-Heart and Circulatory Physiology*, vol. 286, no. 4, pp. H1573–H1589, 2004.

[13] A. Beck, T. Bolemann, H. Frank, F. Hindenlang, M. Staudenmaier, G. Gassner, and C.-D. Munz, "Discontinuous galerkin for high performance computational fluid dynamics," in *High Performance Computing in Science and Engineering 13*. Springer, 2013, pp. 281–294.

[14] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, "Improving large graph processing on partitioned graphs in the cloud," in *Proc. 3. ACM Symp. on Cloud Comp.* ACM, 2012.

[15] D. Kumar, A. Raj, D. Patra, and D. Janakiram, "Graphive: Heterogeneity-aware adaptive graph partitioning in graphlab," in *Par. Process. Work. (ICCPW), Int. Conf.*, 2014.

[16] C. Peng, M. Kim, Z. Zhang, and H. Lei, "Vdn: Virtual machine image distribution network for cloud data centers," in *INFOCOM 2012*.

[17] I. Narayanan, A. Kansal, A. Sivasubramaniam, B. Urgaonkar, and S. Govindan, "Towards a leaner geo-distributed cloud infrastructure," in *6th USENIX Work. on Hot Top. in Cloud Comp.*, 2014.

[18] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *SIGCOMM 2015*.

[19] C. Jayalath, J. Stephen, and P. Eugster, "From the cloud to the atmosphere: Running mapreduce across data centers," *Computers, IEEE Transactions on 2014*.

[20] N. Xu, B. Cui, L.-n. Chen, Z. Huang, and Y. Shao, "Heterogeneous environment aware streaming graph partitioning," 2015.

[21] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *ACM CIKM 2015*.

[22] G. Hamerly and C. Elkan, "Learning the k in k-means," *Advances in neural information processing systems*, vol. 16, p. 281, 2004.

[23] T. Stützle, "Iterated local search for the quadratic assignment problem," *Europ. Jour. of Oper. Research*, vol. 174, no. 3, pp. 1519 – 1539, 2006.

[24] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive workload classification and forecasting for proactive resource provisioning," *Concurr. and Comp.: Pract. and Experience 2014*.

[25] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Jour. Am. soc. for inform. science and techn. 2007*.

[26] C. Li, "Distributed data analytics using graph processing frameworks," Master's thesis, University of Stuttgart, 2015.

[27] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *EuroSys 2015*.

[28] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *EuroSys 2013*.

[29] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "Adaptive partitioning for large-scale dynamic graphs," in *ICDCS 2014*.

[30] A. Zheng, A. Labrinidis, and P. K. Chrysanthis, "Architecture-aware graph repartitioning for data-intensive scientific computing," in *Big Data, 2014 IEEE International Conference on*. IEEE, 2014, pp. 78–85.