GrapH: Traffic-Aware Graph Processing

Christian Mayer, Muhammad Adnan Tariq, Ruben Mayer, and Kurt Rothermel

Abstract—Distributed graph processing systems such as Pregel, PowerGraph, or GraphX gained popularity due to their superior performance of data analytics on graph-structured data. These systems employ partitioning algorithms to parallelize graph analytics while minimizing inter-partition communication. Recent partitioning algorithms, however, unrealistically assume a uniform and constant amount of data exchanged between graph vertices (i.e., uniform *vertex traffic*) and homogeneous network costs between workers hosting the graph partitions. This leads to suboptimal partitioning decisions and inefficient graph processing. To this end, we developed GrapH, the first graph processing system using vertex-cut graph partitioning that considers both, diverse vertex traffic and heterogeneous network 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 10 percent in graph processing latency and 60 percent in communication costs compared to state-of-the-art partitioning approaches.

Index Terms—Distributed graph processing, partitioning, vertex-cut, heterogeneous, traffic, geo-distributed, adaptive, workload, architecture, network, streaming partitioning, communication

1 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], [2], [3] has led to the advent of distributed graph processing systems, such as Pregel, PowerGraph, and GraphX [4], [5], [6]. These systems adopt a user-friendly programming paradigm, where users specify vertex functions to be executed in parallel on vertices distributed across workers ("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 [7], [8], [9], [10].

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 partitions (*cut-size*). The idea is that a decreased cut-size leads to lower communication costs due to less inter-worker traffic [5], [6]. But this holds only under two assumptions: *vertex traffic homogeneity*, i.e., processing each vertex involves the same amount of communication overhead, and *network costs homogeneity*, i.e., the underlying network links between each pair of workers have the same usage costs (e.g., [5], [10]). 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, where processing is unevenly distributed across graph areas and vertices. Hotspots mainly arise for three causes. I) Vertices process different amounts of data. Examples are large-scale simulations of heart cells, liquids or gases in motion, and car traffic in cities [1], [11], 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. II) Graph systems execute vertices a different number of times. This can be observed for algorithms defining a convergence criteria for vertices. The graph system skips execution of converged vertices (*dynamic scheduling* [5]) leading to inactive graph areas and therefore different frequencies of vertex execution. Concerning this, a popular example is the PageRank algorithm [5]. III) Graph algorithms inherently focus on specific graph areas. This includes user-centric graph 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. 1). Hence, we argue that assuming homogeneous vertex traffic misfits 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 [5], [12], [13]. Network heterogeneity exists even in a single data center where worker machines are connected via a tree-structured switch topology.

- *C. Mayer, R. Mayer, and K. Rothermel are with the Institute of Parallel and Distributed Systems, University of Stuttgart, Stuttgart 70174, Germany. E-mail: {christian.mayer, ruben.mayer, kurt.rothermel}@ipvs.uni-stuttgart.de.*
- *M.A. Tariq is with the Department of Computer Science, FAST - National University of Computer and Emerging Sciences. E-mail: muhammad.adnan@nu.edu.pk.*

Manuscript received 23 Dec. 2016; revised 11 Jan. 2018; accepted 13 Jan. 2018. Date of publication 18 Jan. 2018; date of current version 11 May 2018. (Corresponding author: Christian Mayer.)
Recommended for acceptance by M. Steinder.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2794989

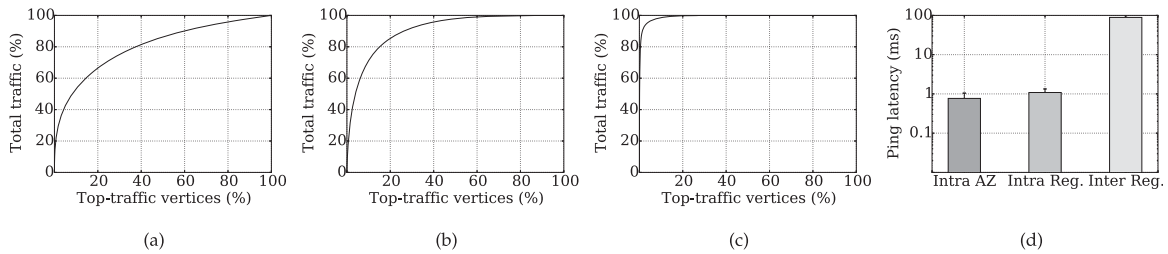


Fig. 1. (a)-(c) Distribution functions of vertex traffic for PageRank, subgraph isomorphism, and cellular automaton. (d) Latency between worker machines intra-Availability Zone (AZ), inter-AZ and intra-region, and inter-region.

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 [14]. Nevertheless, modern cloud infrastructures are *geo-distributed* [15]. Cloud providers such as Amazon, Google, and Microsoft are deploying dozens of data centers world-wide to provide low latency user-access. These data centers host global services such as Twitter that need to analyze large amounts of data (e.g., user friendship relations). Geo-distributed data analytics spanning *multiple* data centers is often the only option [16]. For instance, data should be stored close to the geo-distributed users to reduce access latency, but replication may be prohibitive for legal reasons or efficiency considerations [17], [18]. In these scenarios, network link costs can differ by orders of magnitudes (cf. Figs. 1d, 12a, and Table 2). These heterogeneous network costs significantly influence overall communication costs and therefore should be considered when partitioning the graph.

To overcome these limitations, we developed GrapH, a distributed graph processing system for in-memory data analytics on graph-structured data (*https://github.com/GrapH2/GrapH2.0*). GrapH is aware of both, dynamic vertex traffic and underlying network link costs. Considering this information, it adaptively partitions the graph at runtime to minimize overall communication costs by systematically *avoiding frequent communication over expensive network links*. This paper is an improved version of [19] and provides the following extended contributions:

- A fast single-pass partitioning algorithm, named *H-load*, and a fully distributed edge migration algorithm for runtime refinement, named *H-adapt*, solving the dynamic vertex traffic- and network-aware partitioning problem. H-adapt extends previous work [19] by i) regulating the migration frequency between workers (*constant back-off migration*), and ii) eliminating the need to exchange locking messages (*lock-free migration*).
- A method for online vertex traffic prediction, named *Adaptive-α*, that treats *each individual vertex as an independent learner* and thereby is able to predict diverse and dynamic vertex traffic patterns. Compared to standard methods for time series prediction, Adaptive-α reduces the prediction error by 10-30 percent.
- Two novel distributed algorithms plus implementation in the vertex-centric API solving two important graph problems: subgraph isomorphism to find arbitrary subgraphs in the graph, and agent-based cellular automaton to simulate physical phenomena such as mobility of citizens.
- Extensive evaluations on PageRank, subgraph isomorphism, and cellular automaton—for multiple graph data sets with up to 1.4 billion edges—showing that GrapH reduces communication costs by up to 60 percent and end-to-end graph processing latency (including overhead of dynamic repartitioning) by more than 10 percent compared to state-of-the-art partitioners.

The paper is structured as follows. We formulate the network- and traffic-aware dynamic vertex-cut partitioning problem in Section 2, present the partitioning algorithms in Section 3, introduce the vertex functions for subgraph isomorphism and cellular automaton in Section 4, evaluate in Section 5, discuss related work in Section 6, and conclude in Section 7.

2 Preliminaries and Problem Formulation

In this section, we provide preliminaries and present the network- and traffic-aware dynamic partitioning problem.

2.1 Preliminaries

We assume the widely-used vertex-centric, iterative graph computation model from PowerGraph [5]. In each *iteration*, the system executes the user-defined vertex function for all *active* vertices and waits until they terminate (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. In the apply phase, a vertex changes its local data using σ. In the scatter phase, a vertex activates neighboring vertices for the next iteration. For example, in PageRank each vertex has vertex data $rank \in \mathbb{R}$, gathers the sum σ over all neighbors' $rank$ values, changes its vertex data $rank$ using σ (i.e., $rank = 0.15 + 0.85 * \sigma$), and activates neighbors, if $rank$ has changed more than a threshold.

In order to parallelize GAS execution, a graph is distributed onto multiple workers by cutting it through edges or vertices (*edge-cut* or *vertex-cut*) [5]. Edge-cuts distribute vertices to partitions, therefore an edge can connect vertices on different partitions. Vertex-cuts distribute edges to partitions, therefore a vertex can span multiple partitions, each having its own *vertex replica*. The set of partitions, where vertex u is replicated, is denoted as *replica set* R_u. The *replication degree* is defined as the total number of vertex replicas. Vertex-cut has superior partitioning properties for real-world graphs with power-law degree distribution such as the Twitter or Facebook graph [5]. Thus, we use vertex-cut in this paper.

Inter-partition communication happens only in the form of *vertex traffic* between replicas. For instance, if all neighbors

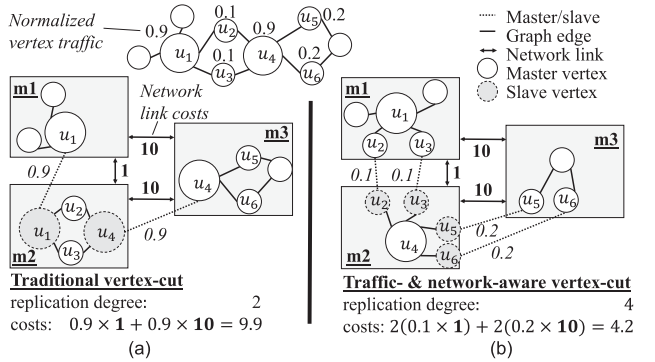


Fig. 2. (a) Vertex-cut minimizing replication degree. (b) Network- and traffic-aware vertex-cut minimizing communication costs.

of vertex v are on the same partition, 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 *master replica* $\mathcal{M}_v$, initiates the distributed vertex function execution and keeps vertex data consistent on the other *mirror replicas*. There are three types of network communication. First, a master sends a *gather request* to each mirror which replies a *gather response* containing an aggregation of local neighboring data (e.g., summed page $rank$s). 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. Third, in order to schedule neighbors of v for future execution, *scatter requests* of constant size s are exchanged between master and mirrors.

In general, vertex traffic between the master and the replicas of vertex v can vary due to a different size of the vertex data and the different amount of gathered data (e.g., gather sum being a union operation, cf. Section 4). In Eq. (1), we define vertex traffic $t^v(i)$ of vertex v in iteration i as the summed byte size of gather, apply, and scatter messages. In order to make vertex traffic independent from the number of replicas to prevent partitioning oscillations (i.e., due to the wrong assumption that a vertex with good locality induces low traffic), we averaged vertex traffic $t^v(i)$ of vertex v in iteration i over all replicas in the replica set $R_v(i)$.

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

2.2 Network- and Traffic-Aware Dynamic Vertex-Cut

Vertex traffic and network costs are heterogeneous. In Figs. 1a, b, and c, we show vertex traffic heterogeneity for three algorithms: PageRank, subgraph isomorphism and cellular automaton (cf. Section 5 for details). The graph shows the x/y distribution: x percent of the top-traffic vertices are responsible for y percent of overall traffic. In our evaluations, PageRank is 20/65 distributed, because of different convergence behaviors of vertices as mentioned in Section 1. Subgraph isomorphism is more extreme with a 20/84 distribution, because some vertices match more subgraphs than others. Cellular automaton is highly imbalanced (20/100), because vertices simulating unpopular regions in Beijing have almost zero traffic (cf. Section 5). Besides vertex traffic, network communication is also subject to significant variations in terms of bandwidth and latency, even in a single data center [8], [12], [20]. For Amazon EC2 workers, we show orders-of-magnitude variations of latency (cf. Fig. 1d). 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), but for communication across different AZs and regions, i.e., respectively 0.01$/GB and 0.02$/GB (cf. Table 2). We model this heterogeneity using a *static, weighted worker topology*, where each pair of workers is associated with specific network-related costs. In Section 5, we state precisely how the weighted worker topology can be determined in practical scenarios with very low overhead.

TABLE 1
Notation Overview

M	The set of workers.
k	The number of workers.
$G = (V, E)$	The graph with vertex set V and edge set E.
V_m	The set of vertices replicated on worker m.
I	The set of all iterations.
$T_{m,m'}$	The network costs between workers m and m'
$\mathbb{A}$	Function mapping edges to workers.
R_v	Replica set of vertex v.
$\mathcal{M}_v$	Master of replica set of vertex v.
$t^v(i)$	Vertex traffic of vertex v in iteration i.
$\hat{t}^v(i)$	Vertex traffic estimation of vertex v for iteration i.
$L_m(i)$	Load (*summed vertex traffic*) of worker m in iteration i.
C	Capacity of exchange partner worker.
$\beta(x)$	Byte size of serialized information x (e.g., vertex).
μ	Aggressiveness parameter specifying *willingness-to-move*.
c_+	Investment costs of migrating edges.
c_-	Payback costs in terms of saved future traffic.

Efficient graph partitioning should utilize these diverse costs. For example in Fig. 2a, vertices are annotated with their (normalized) vertex traffic, also indicated by the vertex size. Workers $m1 - m3$ communicate via network links with different costs, given by the weights in bold. The vertex-cut distributes vertices u_1 and u_4, both having traffic 0.9. *Communication costs* via a network link are the costs 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 replication degree, but high communication costs, because high-traffic vertices u_1 and u_4 induce more network overhead. 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 inter-partition communication. In Fig. 2b, we minimize communication by cutting the graph on vertices u_2, u_3 and u_5, u_6. This increases the replication degree, 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 $2(0.2 * 1) + 2(0.1 * 10) = 2.4$. In Table 1, we summarize notation used in the paper.

TABLE 2
Heterogeneous Network Link Costs for AWS (Jan 2018)

Worker placement	Incoming traffic	Outgoing traffic
Same AZ	0.00-0.01 $/GB	0.00-0.01 $/GB
Different AZ, same region	0.01 $/GB	0.01 $/GB
Different region	0.00 $/GB	0.02 $/GB
Internet	0.00 $/GB	0.00-0.09 $/GB

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

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

$$\mathbb{A}_{opt} = \underset{\mathbb{A}}{\operatorname{argmin}} \sum_i \sum_{v \in V} \sum_{m \in R_v^{\mathbb{A}}(i)} t^v(i) \; T_{m,\mathcal{M}_v}. \tag{2}$$

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

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

Theorem 1. *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, $\mathbb{A}_{opt} = \operatorname{argmin}_{\mathbb{A}} \sum_i \sum_{v \in V} \sum_{m \in R_v^{\mathbb{A}}(i)} 1 * 1 \; = \; \operatorname{argmin}_{\mathbb{A}} \sum_{v \in V} |R_v^{\mathbb{A}}|$, Eq. (2) becomes the network- and traffic- *unaware* vertex-cut problem, which is NP-hard (e.g., [21]). □

3 Partitioning Algorithms

In this section, we present two algorithms addressing the network- and traffic-aware partitioning problem: *H-load* for pre-partitioning the graph and *H-adapt* for runtime refinement using dynamic migration of edges.

3.1 H-load: Initial Partitioning

Graph processing systems pre-partition the graph, so that each worker can load its partition into local memory. To this end, we developed H-load, a fast pre-partitioning algorithm that consists of two phases (cf. Fig. 3). First, it divides

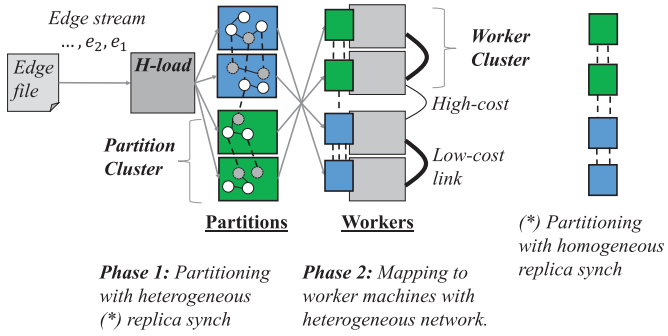


Fig. 3. Approach overview H-load.

the graph into k partitions using a vertex-cut algorithm. Second, it determines a cost-efficient mapping from partitions to workers. We describe these two phases in the following.

1) The goal of the first phase is to find a reasonable partitioning of the graph into k balanced parts, ignoring concrete mapping of partitions to workers. In order to improve partitioning performance of billion-scale graphs, we assume a streaming setting: the graph is given as a stream of edges $e_1, e_2, \ldots, e_{|E|}$ with $e_i \in E$ and we consecutively read and assign one edge at a time to a partition until all edges are assigned. For instance, PowerGraph [5] greedily reduces replication degree by assigning edges to the partitions where incident vertices are already replicated. However, the PowerGraph partitioning leads to homogeneous total replica synchronization between each pair of partitions as they share a similar amount of replicas (cf. Fig. 3 right).

In order to exploit heterogeneity of network link costs, inter-partition traffic must be heterogeneous as well: partitions exchanging more traffic should be mapped to workers with low-cost network links (cf. Fig. 3 center). Therefore, our aim in the first phase of H-load is to produce a clustered partitioning consisting of *partition clusters* such that two partitions with many shared replicas are likely to reside in the same partition cluster. Hence, the partition clusters are designed to have low intra- and high inter-cluster traffic, i.e., more traffic will be exchanged between partitions in the same partition cluster.

How can we ensure that a mapping from these partition clusters to the workers exists such that communication costs during graph processing are minimal? H-load addresses this problem by assuming that the network cost matrix T consists of several *worker clusters* with low intra-cluster and high inter-cluster costs (e.g., EC2 instances running in different availability zones). The number of worker clusters c can be determined from the matrix T using well-established clustering methods [22]. We use the number of worker clusters c in the first phase of H-load to generate c partition clusters by grouping the partitions into equal-sized clusters and assigning edges to partitions such that replicas preferentially lie in the same partition cluster. Hence, the relation between the worker clusters and the partition clusters is the following: we use the number of worker clusters c to produce c partition clusters. Therefore, the first phase operates independently of the concrete mapping of partitions to workers.

Each edge (u, v) is assigned to a partition p 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

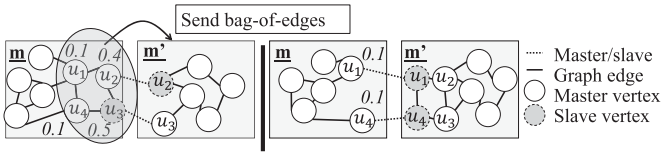


Fig. 4. Example: Bag-of-edges migration to reduce inter-partition traffic.

those partitions. Otherwise, a new replica has to be created (say, of vertex v). We choose partition p, such that the new replica of v preferentially lies in the same partition cluster as already existing replicas of v—simply by counting the number of replicas in each partition cluster. With this method, our algorithm ensures a clustered traffic behavior: partitions in the same partition cluster share the same replicas and thus are expected to exchange more traffic than partitions in different partition clusters.

2) The second phase retrieves a mapping from the $|M|$ partitions to $|M|$ workers while minimizing overall communication costs. A minimal-cost mapping of partitions to workers would assign two partitions with high inter-partition traffic to workers connected via a low-cost network link (cf. Fig. 3). This is an instance of the well-known quadratic assignment problem: map $|M|$ factories (i.e., partitions) to $|M|$ locations (i.e., workers), so that the mapping has minimal costs of factories sending their goods to other factories (i.e., communication costs). We used the iterated local search algorithm of Stützle et al. [23] to minimize (communication) costs. Initially, partitions are randomly mapped to workers. Then the algorithm iteratively improves the total costs using the following method. Find two workers such that exchanging partition assignments results in lower total communication costs. For example in Fig. 2, exchanging partition assignments of workers $m1$ 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 the problem set is relatively small 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 previous executions of the GAS algorithm. Otherwise, homogeneous traffic between vertex replicas is assumed.

3.2 H-adapt: 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-adapt* solving the dynamic heterogeneity-aware partitioning problem (note that this algorithm extends [19] by three optimizations (i) adaptive-α, (ii) constant back-off migration, and (iii) lock-free migration). The idea is that each worker locally reduces the communication costs by migrating edges to distant workers. To this end, we define the term *bag-of-edges* as the set of edges to be migrated. Workers migrate bag-of-edges in parallel after each GAS iteration.

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

Algorithm 1. Migration Algorithm on Worker m

```
1:  waitForActivation()
2:  m' ← selectPartner()
3:  b ← bagOfEdges(m')
4:  lock(b)
5:  b ← updateLocked(b)
6:  Δc ← c+ − c−
7:  if Δc < 0 then
8:    migrateBag(b)
9:  end if
10: releaseLocks(b)
```

We give an example of this procedure in Fig. 4. Two workers 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'. Worker 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. Here, low-traffic vertices u_1 and u_4 are cut leading to less inter-partition traffic. In the following, we describe the proposed H-adapt algorithm (cf. Algorithm 1) in more details.

3.2.1 Selection of Partner and Bag-of-Edges

Which worker to select as exchange partner? Intuitively, two workers sharing high-traffic replicas are strong candidates for exchanging bag-of-edges, because improving their partitioning can greatly reduce the overall communication costs. On the other hand, two workers sharing no or only low-traffic replicas have low potential to improve overall costs. Hence, some workers are more promising partners than other workers for the edge migration. To exploit this, each worker m maintains a candidate list of potential exchange partners (with decreasing priority). Worker m computes the candidate list by sorting neighboring workers w.r.t. the total amount of exchanged traffic. In each round of Algorithm 1, we iteratively select the top-most worker from the list as exchange partner and remove it from the list. Once the list is empty, it is recomputed using the most recent traffic statistics.

However, this strategy leads to suboptimal or redundant selections of exchange partners in the following three cases. First, worker m has already selected worker m' in one of the previous i' iterations and can only find minor improvements of the partitioning. Second, worker m has less load than worker m' and the load balancing requirement

prohibits any migration from m to m'. Third, workers m and m' share no common replicas. In these cases, we do not expect to find significant improvements of the current partitioning. Therefore, we remove worker m' from the candidate list for a constant number of iterations. During this time period, worker m' can not be selected as exchange partner (hence, denoted as *constant back-off migration*). We set the parameter for back-off migration to half the number of partitions, i.e., $k/2$, and set it to infinity as soon the algorithm detects a convergence that is assumed when the bag-of-edges for an exchange partner is empty.

Next, we determine the maximal size of the bag-of-edges to be sent to m' in order to ensure balanced worker load (cf. Eq. (3)). Therefore, we introduce the notion of *capacity* of a worker m', i.e., the maximum amount of additional load, worker m' can carry. Capacity is defined as half the difference of loads $L_{m'}$ and L_m of the receiving and the sending worker: $C = (L_{m'} - L_m)/2$. To learn about the current load $L_{m'}$ of worker m', worker m sends a request to m'. Using the capacity, worker m can control the size of the bag-of-edges, such that load deviation 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', worker m will not include v into the bag-of-edges.

Once the exchange partner is selected and we know its capacity, we determine a bag-of-edges (bag) to be send. Selecting a suitable bag is crucial for optimizing communication costs and migration overhead. Theoretically, the perfect bag could be any subset out of p edges on a worker (i.e., 2^p subsets). In order to keep the migration phase lean, we developed a fast heuristic to find a bag improving communication costs (cf. Algorithm 2). Initially, worker m determines the set of candidate vertices, those replicated on both workers, because they are responsible for all the traffic between m and m'. Worker m sorts the candidates by descending vertex 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 (lines 6-7, cf Section 3.2.2). If the total communication costs would decrease when sending the edges, worker m adds them to the bag (lines 8-9).

Algorithm 2. Determining the Bag-of-Edges to Exchange

```
1: function BAGOFEDGES(m'):
2:   bag ← []
3:   candidates ← sort(adjacent(m'))
4:   while hasCapacity(m', bag) do
5:     v ← candidates.removeFirst()
6:     b ← {(u, v)|u ≠ v}
7:     Δc ← c+ − c−
8:     if Δc < 0 then
9:       bag ← bag + b
   RETURN bag
```

3.2.2 Calculation of Costs

Clearly, migrating bag b from one worker to another is only beneficial, if it results in lower overall costs (i.e., line 6 in Algorithm 1, and line 7 in Algorithm 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 and should be avoided. Payback costs are the saved costs after migrating bag b in the form of less future inter-partition 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 worker 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 worker 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). Because of this, m has to send an update to all workers in R_u with the new vertex replica set, denoted as R'_u. Additionally, when creating a new replica on m', worker 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. 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 workers 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 (i) the function $\beta(x)$ returns the number of bytes needed to encode x (to be sent over the network), (ii) the indication function $\delta(u)$ returns 1, if worker m' has no local replica of u, otherwise 0, (iii) V_b is the set of all vertices in bag b, and (iv) V'_b is the set of all vertices whose replica sets will change when sending the bag b to m'.

Payback costs: we can also save costs when sending bag b 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. Theoretically, 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.

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

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

3.2.3 Vertex Traffic Prediction

To estimate payback costs, we first need to predict vertex traffic in future iterations. More formally, given vertex

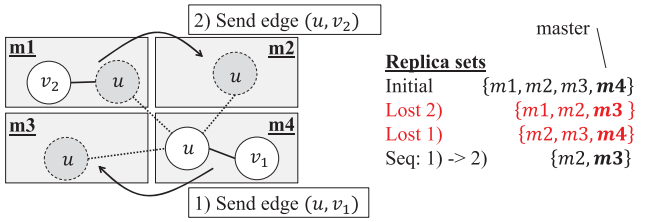


Fig. 5. Lost update problem for parallel edge migration.

traffic $t^v(0), t^v(1), \ldots, t^v(i)$ of vertex v, we estimate traffic values $t^v(i+1), \ldots, t^v(|I|)$. The prediction should be fast with 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 (cf. [24]) for time series prediction of the next traffic value $t^v(i+1)$ that fit to our requirements. The first method is *most recent value* (denoted as *Last*) 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* (abbreviated as *MA*) with the idea of using 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* (abbreviated as *EA*) calculating the prediction based on the previous prediction and the last observed 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 recently observed traffic.

Which prediction method performs best? In Fig. 13, we compare overall system performance for these methods with different parameter choices, i.e., window sizes w and decay parameters α. In summary, the third method, i.e., exponential averaging, leads to the best overall system performance for certain choices of the parameter α. The reason is that exponential averaging is able to express both rapid (i.e., short term) and gradual (i.e., long term) changes in vertex traffic by varying the parameter α. For instance, setting α to a high value (e.g., $\alpha = 0.9$) is better for predicting rapidly changing vertex traffic whereas setting α to a low value (e.g., $\alpha = 0.1$) leads to more accurate long term traffic pattern prediction. A major challenge is to select the correct parameter α according to the concrete traffic pattern of the graph algorithm. Furthermore, assuming a *global and static* parameter value α for all vertices is unrealistic for many applications such as social simulations, where some vertices are responsible for simulating long-term, repeating movement patterns (e.g., in a work environment) and others for simulating short-term, unique movement patterns (e.g., a flash mob). In such cases, the graph system should select the parameter α automatically and specifically for each vertex.

Adaptive-α: To this end, we developed our method Adaptive-α where each vertex learns its individual prediction model by dynamically optimizing α at runtime. This releases the system administrator from the burden of choosing the optimal parameter value and empowers vertices to catch their individual diverse and dynamic traffic patterns. In more details, each vertex continuously monitors the accuracy of its predictions by comparing its previous vertex traffic predictions with the actually observed vertex traffic. For a given α, the prediction accuracy of a vertex v in iteration i can be quantified by using the error function $e_{v,i}(\alpha)$ as shown in Eq. (6).

$$e_{v,i}(\alpha) = (t^v(i) - \hat{t}^v(i))^2. \tag{6}$$

In order to find the value of α that minimizes the error function, we set the first order derivative of Eq. (6) to zero, i.e., $\frac{\partial e_{v,i}}{\partial \alpha} = 0$, and calculate the extremum of $e_{v,i}(\alpha)$ as follows:

$$\begin{aligned} &\frac{\partial}{\partial \alpha}(t^v(i) - (\alpha t^v(i-1) + (1-\alpha)\hat{t}^v(i-1)))^2 = 0 \Leftrightarrow \\ &\alpha_{min} := \alpha = \frac{\hat{t}^v(i-1) - t^v(i)}{\hat{t}^v(i-1) - t^v(i-1)}. \end{aligned} \tag{7}$$

If the extremum is a local minimum (i.e., the second order derivative is greater than zero) and $\alpha_{min} \in [0,1]$, we set $\alpha = \alpha_{min}$. Note that each vertex can quickly calculate α_{min} by simply applying the closed formula in Eq. (7) (cf. Fig. 11c).

With the above mentioned methods, we can determine the vertex traffic estimation for the next iteration. 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 representing the minimum number of iterations we expect to save communication costs as a result of migrating bag b. This parameter specifies the aggressiveness with which migration should be performed. We set it to the total number of iterations specified by the GAS algorithm minus the current iteration, i.e., to the (expected) number of iterations left in the algorithm. Together, our estimated payback costs are the following (denoting the new master of vertex v as $\mathcal{M}'_v$ and the old master as $\mathcal{M}_v$—cf. Eq. (5)).

$$c_- = \mu \sum_{v \in V_b} \left(\sum_{r \in R'_v} \hat{t}^v(i+1) T_{r,\mathcal{M}'_v} - \sum_{r \in R_v} \hat{t}^v(i+1) T_{r,\mathcal{M}_v} \right). \tag{8}$$

3.2.4 Lock-Free Migration

When two workers independently migrate edges and change replica sets of vertices, inconsistencies of the data graph can arise. In Fig. 5, we give an example. Workers $m1$-$m4$ maintain a replica of vertex u. Initially, all workers have the same view of the replica set $\{m1, m2, m3, m4\}$. Now, suppose worker $m4$ migrates edge (u, v_1) to worker $m3$. At the same time, worker $m1$ migrates edge (u, v_2) to $m2$. Both workers $m1$ and $m4$ remove the local replica of vertex u, if no incident edge exists on the respective worker. In order to synchronize the replica sets, worker $m4$ ($m1$) has to update all other workers having a replica of u with the new replica set. Worker $m4$ sends the new replica set $\{m1, m2, m3\}$ to all workers, while worker $m1$ sends $\{m2, m3, m4\}$. However, different workers can receive these updates in different orders leading to inconsistent views on the replica sets (lost update problems). Instead, sequential updates would result in a consistent state (e.g., worker $m4$ changes the replica set *before* worker $m1$).

To guarantee sequential updates during edge migration, the standard procedure is to send locking requests to the masters of the vertices to be migrated [19]. More precisely, a

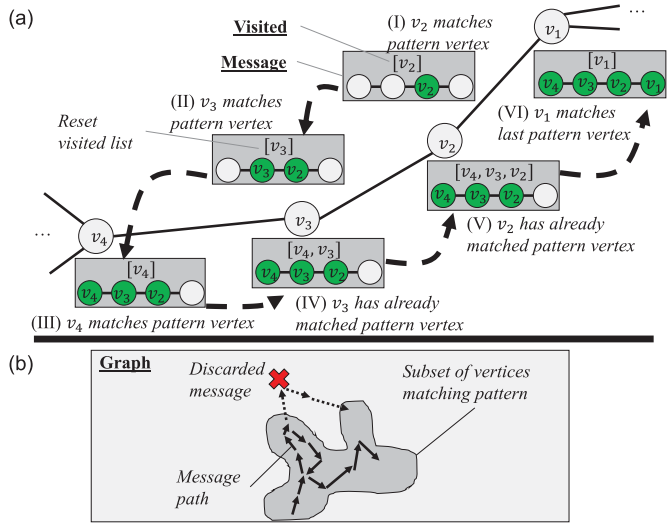


Fig. 6. (a) Subgraph isomorphism example – perspective of vertex v_2. (b) The algorithm considers only messages that travel *within the matching subgraph*.

worker m locks endpoint vertices in the bag-of-edges to be sent to the partner worker m'. For vertex u it sends a locking request to the master $\mathcal{M}_u$. If vertex u is already locked, the master worker returns $false$, otherwise it locks u and returns $true$. If a worker holds a lock, no other worker can change the replica set of a vertex and the distributed graph is always in a consistent state.

However, acquiring all locks via locking messages leads to significant latency penalty (at least one round trip time per migration phase) and message overhead (one locking message per vertex in the bag-of-edges). To save this additional overhead, we designed a so called *implicit locking scheme* that enables workers to acquire locks for vertices without the need for explicitly exchanging lock messages. In Algorithm 3, we describe our implicit locking scheme. The given function returns true, if worker m owns the lock for vertex v. In order to enable each worker m to migrate each combination of vertices eventually, worker m possesses locks for all vertices every kth iteration (line 2)—the value k being the number of participating workers. This ensures fairness because each worker has an equal chance of getting all locks. However, a worker does not need the lock for vertex v, if it does not possess a vertex replica of vertex v. Therefore, the lock of this vertex v is given to another worker that contains a replica of vertex v and therefore might actually need it for migration (line 4). This simple technique allocates vertex locks without inducing any locking message.

Algorithm 3. Implicit Locking Scheme. Does Worker m have the Lock for Vertex v?

```
1: function hasLock m, v
2:     m̂ ← iteration() mod k
3:     if m̂ ∉ R_v then
4:         m̂ ← R_v.sort()[iteration() mod |R_v|]
5:     return m == m̂
```

4 Graph Algorithms

In order to evaluate our partitioning methods and heterogeneity of vertex traffic, we have implemented three important graph algorithms: PageRank (cf. [5]), subgraph isomorphism, and social simulations via agent-based cellular automaton, denoted as PR, SI, and CA, respectively. For SI and CA there is, to the best of our knowledge, no vertex-centric algorithm, so we have designed novel algorithms and implemented them in the GAS API.

4.1 Subgraph Isomorphism

The NP-complete subgraph isomorphism problem addresses the question, whether a graph contains a subgraph that is isomorphic to a specified subgraph [25]. More precisely, given an undirected 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 (cf. [25]). Each graph vertex can have an optional label (e.g., a category to which the graph vertex belongs). In this case, the labeled SI additionally requires both vertices in V_{sub} and V_P to have matching labels.

The main idea of our GAS algorithm is to solve SI using a vertex-centric message passing scheme. Conceptually, a message represents a partially matched graph pattern, i.e., we associate graph vertices with pattern vertices (assuming they have matching labels). We denote a message as *partially matched* if we have not yet associated all pattern vertices with graph vertices. In contrast, a *fully matched* message associates one unique graph vertex with *each* pattern vertex in the message. If a message is fully matched, we have found a matching subgraph. Eventually, we are guaranteed to retrieve matching subgraphs (if they exist) by repeatedly attempting to match a pattern vertex in the message and sending the message to all neighbors, because this creates a message for each possible path in the graph and one of these paths leads to a fully matched message.

However, the number of paths and therefore the number of messages grows exponentially—considering that a message is gathered by *all* neighbors in each iteration. To limit the number of concurrent messages traversing the graph, we focus on these messages that traverse only graph vertices that can match exactly one pattern vertex and discard all other messages. In Fig. 6a, we exemplify the path of a single message (I) to (VI) that subsequently traverses graph vertices attempting to match a pattern vertex in the message. Vertices that have already matched a pattern vertex (IV)-(V) simply forward the message to their neighbors. The example shows that the message traverses some vertices twice, if they have already matched a pattern vertex. In Fig. 6b, we sketch a graph containing a subgraph that matches the pattern. Also, we give a specific message traversing the graph to find the matching subgraph. A message that has traversed all vertices in the subgraph would successfully return the found subgraph. Messages that leave the subgraph of vertices matching a pattern vertex are discarded immediately (in contrast to the naive approach given above). This already reduces the number of concurrent messages traversing inefficient paths. Furthermore, we break infinite message forwarding cycles if there is no progress in terms of newly matched pattern vertices. To this end, we maintain a list of vertices, denoted as $visited$, that have already processed this message (cf. line 14).

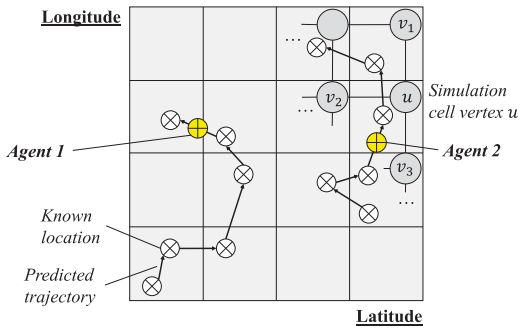


Fig. 7. Agent-based social simulations via cellular automaton.

In Algorithm 4, we specify the vertex data and the gather, apply, and scatter phases of the vertex function. The vertex data consists of the graph pattern $P = (V_P, E_P)$ and the set of messages ω to be processed by graph vertex u. We define a message as a tuple $(\mathcal{X}, visited)$ where $\mathcal{X} \subset V \times V_p$ specifies the set of current matches between graph vertices and pattern vertices. In the gather phase (line 4), vertex u collects the messages stored at neighboring vertex v and performs a union operation over all gathered messages from neighboring vertices (line 6). In the apply phase, vertex u processes all gathered messages by matching pattern vertices if possible (lines 8-23). If a graph vertex u has already matched a pattern vertex, it forwards the message to neighbors by adding them to the message set $D_u.\omega$ (cf. line 15). Note that the function $match(\ldots)$ (line 18) returns true, if for all edges in the pattern there are corresponding edges in the graph (i.e., $\forall(u_1, u_1'), (u_2, u_2') \in \mathcal{X} : (u_1', u_2') \in E_p \Rightarrow (u_1, u_2) \in E$) and the labels of associated graph and pattern vertices match. Graph vertices that can not match a pattern vertex (or that detect a message loop) drop the message. In the scatter phase (lines 24-26), vertex u schedules neighboring vertex v, if u has partially matched messages to be processed by vertex v.

4.2 Cellular Automaton

The powerful and well-established model of *cellular automaton* is able to express various problems in several research areas [26] such as simulations of complex systems. A cellular automaton models the problem space as a grid of cells, each with a finite number of states. A cell iteratively calculates its own state based on the states of neighboring cells. We believe, that cellular automata fit extremely well to recent graph processing systems due to the powerful and generic abstraction and the simplicity of expressing complex problems within the vertex-centric programming model.

We implemented an agent-based variant for simulating real-world movements of people in Beijing (*http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13/*). In Fig. 7, we give an example of agent-based simulations. Given are two moving agents in an area described as a two-dimensional euclidean space using latitude and longitude coordinates. For each agent, a series of known locations at certain points in time (i.e., movement trajectory L) is given, e.g., extracted from GPS signals of user smartphones [27]. The area is divided into a grid of cells, whereby each cell is assigned to a graph vertex. A graph vertex is connected to all vertices of neighboring cells and is responsible for all agents within its assigned cell. In the example, vertex u is connected to vertices v_1, v_2, v_3 and simulates movement of agent 2. More precisely, the vertex data consists of a set of agents A, each agent maintains its current location loc and the set of known locations L, both defined by (latitude,longitude,time) coordinates (cf. Algorithm 5). In the gather phase, vertex u collects agents from a neighboring vertex v that are in u's responsibility area (line 5). The sum function performs a simple union operation over all gathered agents. In the apply phase, vertex u adds all gathered agents to its local agents A (line 9) and removes all agents that have left the responsibility area of vertex u (line 10). Then the location of all agents is updated by incrementing time by Δt (lines 11-14) using linear interpolation to estimate the agent's current location loc. Finally, in the scatter phase, vertex u activates itself, if he possesses local agents, and activates neighboring vertices, if u has local agents in A that have left its responsibility area (line 18).

TABLE 3
Real-World Graphs for Evaluations

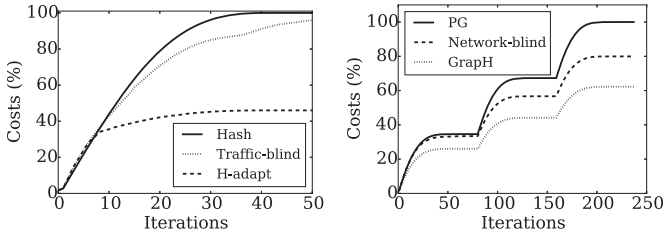
Name	$\|V\|$	$\|E\|$
Twitter	81,308	1,768,149
GoogleWeb	875,713	5,105,039
Web	41,291,594	1,150,725,436
TwitterLarge	41,652,230	1,468,365,182

Algorithm 4. Subgraph Isomorphism Algorithm

```
1:  Vertex data D_u:
2:  P = (V_P, E_P)
3:  ω                    ▷The set of partially matched pattern messages
4:  function GATHER(u, v)
5:    return D_v.ω
6:  function SUM(S_1, S_2)
7:    return S_1 ∪ S_2
8:  function APPLY(D_u, S)
9:    if First execution of apply then
10:     D_u.ω ← {({(u, v_P)}, [u]) | u matches v_P ∈ V_P}
                 ▷Create message for each matching of graph
                    vertex u and a pattern vertex
11:   else
12:     D_u.ω = ∅
13:     for all (X, visited) ∈ S do
14:       if ∃(u, v_P) ∈ X ∧ u ∉ visited then
15:         D_u.ω ← D_u.ω ∪ {(X, visited + [u])}
16:       else if not ∃(u, v_P) ∈ X then
17:         for all c ∈ neighbors(visited[−1]) do
18:           if match(X ∪ {(u, c)}) then
19:             X ← X ∪ {(u, c)}
20:             visited ← [u]
21:             D_u.ω ← D_u.ω ∪ {(X, visited)}
22:             if |X| == |V_P| then
23:               X is a correct matching
24: function SCATTER D_u, D_v
25:   if D_u.ω ≠ ∅ then
26:     Activate({u, v})
```

5 Evaluations

In the following, we present evaluations for GrapH on two computing clusters for three different algorithms, i.e., PageRank, subgraph isomorphism, and cellular automaton, on several real-world graphs given in Table 3 with



(a) PR on *Twitter*. (b) PR on *GoogleWeb*.

Fig. 8. (a) Traffic- and (b) network-awareness reduce communication costs.

up to 1.4 billion edges (*http://konect.uni-koblenz.de/networks/twitter*, *http://snap.stanford.edu/data*). We compare H-load and H-adapt against both existing static vertex-cut approaches, i.e., hashing of edges (Hash) and PowerGraph (PG) [5], [6], as well as traffic- and network-blind algorithmic variants.

Evaluation Setup. We have implemented GrapH in the Java programming language ($> 10,000$ lines of code). GrapH consists of a master worker and multiple client workers performing graph analytics. The master receives a sequence of graph processing queries $q_1, q_2, q_3, \ldots$ consisting of user specified GAS algorithms. All workers communicate with each other directly via TCP/IP.

Algorithm 5. Cellular Automaton Algorithm of Vertex u

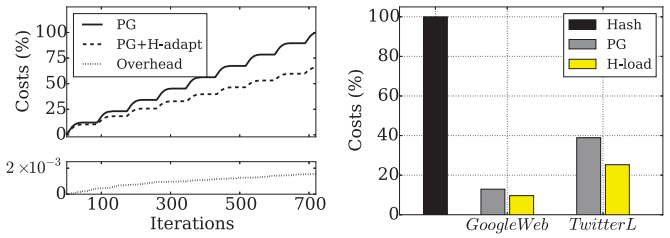
Vertex data D_u:
A ▷The set of local agents
$Cell$ ▷Responsibility area
function GATHER(u, v)
  return $\{((x, y, t), L) \in D_v.A | (x, y) \in Cell\}$
function SUM(S_1, S_2)
  return $S_1 \cup S_2$
function APPLYu, S
  $D_u.A \leftarrow D_u.A \cup S$
  $D_u.A \leftarrow D_u.A \setminus \{((x, y, t), L) \in D_u.A | (x, y) \notin Cell\}$
  Remove the agents that have left the cell
  for all $(loc, L) \in D_u.A$ **do**
    $last \leftarrow$ *previous location in* L
    $next \leftarrow$ *next location in* L
    $loc \leftarrow interpolateLocation(last, next, \Delta t)$
function SCATTER(u, v)
  if $D_u.A \neq \emptyset$ **then**
    $Activate(u)$
  if $|\{((x, y, t), L) \in D_u.A | (x, y) \notin Cell\}| > 0$ **then**
    $Activate(v)$

We used four computing infrastructures with homogeneous and heterogeneous network costs. (i) The homogeneous computing cluster (ComputeC) consists of 12 machines, each with 8 cores (3.0 GHz) and 32 GB RAM, interconnected with 1 Gbps Ethernet. (ii) Furthermore, we performed evaluations on an in-house shared memory machine (ComputeM) with 32 cores (2.3 GHz) and 280 GB RAM. Unless stated otherwise, we modeled network costs in ComputeC and ComputeM as follows, $\forall m, m' \in M: T_{m,m} = 0 \wedge m \neq m' \rightarrow T_{m,m'} = 1$. (iii) The heterogeneous computing cluster (CloudC) is deployed in the Amazon cloud using 8 geographically distributed 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 monetary costs charged by Amazon (cf. Table 2). If not mentioned otherwise, the experiments are performed on CloudC. (iv) The heterogeneous computing infrastructure (CloudM) consists of two powerful EC2 multi-core machines in the same availability zone (m4.16xlarge) with 64 virtual CPUs and 256 GB RAM. Scaling up computation on a small number of powerful machines becomes more and more common in recent years [28], [29]. GrapH supports scaling up by simply creating multiple workers running on the same machine. However, as *distributed* graph processing mainly focuses on scaling out, our default setting is ComputeC or CloudC with one worker per machine. The network costs matrix T is calculated based on packet round-trip times and presented in Fig. 12a. Note that for large-scale deployments, even with hundreds of thousands of servers, there are scalable methods with low overhead (i.e., $< 1\%$ CPU overhead, < 45 MB RAM footprint, and ≈ 50 KB/s probing traffic) to determine the estimated round-trip time between any two servers in a dynamic environment [30].

Communication Costs. The main idea of this paper is to consider network- and traffic-heterogeneity while constantly repartitioning the graph during computation. In the following, we evaluate the effect of traffic- and network-awareness on total communication costs as defined in Eq. (2): *total traffic sent via each network link, weighted by the costs of the network link*. In our first experiment (cf. Fig 8a), we compared three different partitioning methods: (i) hashing of edges to partitions without dynamic migration (Hash), (ii) our dynamic migration strategy assuming *homogeneous* vertex traffic on the hash partitioned graph (Traffic-blind), and (iii) our dynamic migration strategy on the hash partitioned graph considering heterogeneous vertex traffic (H-adapt). We accumulated communication costs over 50 iterations of PR on *Twitter*. The results indicate that considering heterogeneous vertex traffic greatly improves total communication costs by up to 50 percent compared to Traffic-blind. The reason is that H-adapt implicitly prioritizes high-traffic vertices, that are the major sources of total traffic (cf. Section 2, Fig. 1).

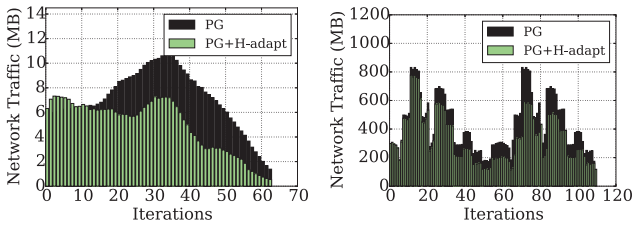
How does network-awareness improve total communication costs? In Fig. 8b, we compare three different partitioning methods: (i) PowerGraph partitioning without dynamic migration, (ii) our strategy H-adapt on the H-load partitioned graph, while both strategies assume homogeneous network costs (Network-blind), and (iii) our dynamic migration strategy H-adapt on the H-load partitioned graph considering heterogeneous network costs (GrapH). We performed 250 iterations of PageRank on *GoogleWeb* (and restarted computation after termination of one PageRank instance). It can be seen, that Network-blind already reduces communication costs by 20 percent compared to PG. However, taking heterogeneous network into account reduces total costs by additional 20 percent. Our experiments therefore indicate, that the awareness of traffic and network heterogeneity improves partitioning quality significantly.

What is the overhead of migrating edges in terms of additional communication costs? In Fig. 9a, we show total communication costs and migration overhead of 700 iterations of PageRank on *GoogleWeb*. While communication costs decreased by 33 percent compared to PG, the costs for



(a) PR on *GoogleWeb*. (b) Pre-partitionings.

Fig. 9. (a) H-adapt reduces communication costs with low overhead. (b) Pre-partitioning with H-load reduces communication costs.



(a) CA on grid (10^5 vertices) (b) SI on *Twitter*.

Fig. 10. (a)-(b) H-adapt reduces network traffic.

migration itself (investment costs, cf. Eq. (4)) are very low compared to the saved costs. The total costs of migration approaches only 2×10^{-3} percent of the overall communication costs due to the following three reasons: (i) optimizing the placement of a vertex replica *once* saves replica communication *in each iteration*, (ii) optimizing the placement of a master replica can reduce replica communication over *multiple* high-costs network links, and (iii) traffic awareness allows for more focused optimizations, i.e., H-adapt automatically targets optimal placement of the high-traffic vertices dominating overall network usage (cf. Figs. 1a, 1b, and 1c). Similar results for SI and CA are omitted due to space constraints.

Finally, we evaluated communication costs improvements of H-load compared with PowerGraph and Hashing for two different graphs: *GoogleWeb*, and *TwitterLarge* (Fig 9b). We assume homogeneous vertex traffic and heterogeneous network costs. H-load greatly reduces total graph processing costs by 70-90 percent compared to Hash and by 25-38 percent compared to PowerGraph partitioning even if no vertex traffic statistics are known (e.g., from previous executions).

Considering Network Heterogeneity when Scaling Up. Many real-world computing clusters such as ComputeC and CloudM consist of a number of multi-core workers connected via Ethernet. These types of infrastructures require both operations, scaling up and scaling out. Similar to other graph systems (e.g., [31]), we support both scaling operations by assigning multiple workers to the same machine. This introduces another source of network heterogeneity: loop-back versus plain network communication.

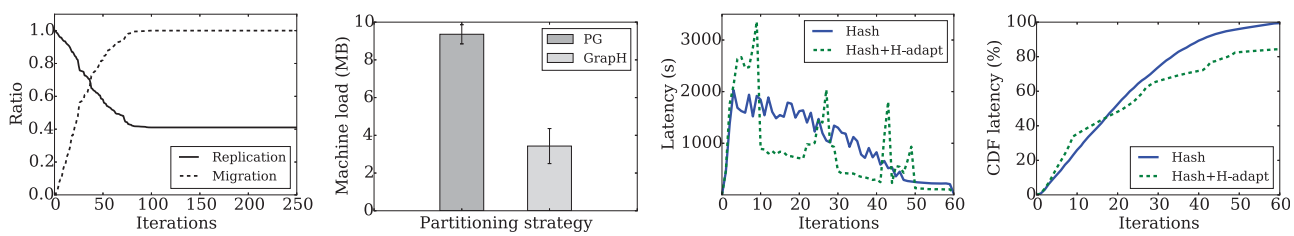
In Fig. 12a, we plot the round-trip time of 8 workers that are evenly divided among two machines in CloudM. Hence, we used four workers per machine for the following experiment. Clearly, the round-trip time differs by an order of magnitude: workers on the same machine can communicate with very low delay of only 0.06 milliseconds—compared to workers on different machines with up to 0.2 milliseconds. We set the network costs matrix $T_{m,m'}$ for workers m and m' according to these round-trip time values.

In Fig. 12b, we attempt to exploit these heterogeneities in CloudM in order to show that single data center applications can benefit from network awareness as well. We performed 250 iterations of PageRank on the web graph [32] and took the round-trip times between the workers as network costs T as described previously. We measured the total accumulated costs (i.e., the product of traffic sent over a link and network link costs) for H-adapt when switching network awareness on and off. The result shows that being network aware reduces graph processing costs by 29% while introducing neglectable costs overhead.

Network Traffic. Sometimes network costs are unknown or relatively homogeneous. In this case, H-adapt dynamically reduces the total amount of worker communication, i.e., *network traffic*. In Figs. 10a and 10b, we show the extent of this improvement for SI and CA (similar results for PR are omitted). We show the current network traffic (averaged over a sliding window of 10 iterations) for these algorithms on ComputeC. The SI algorithm searched for a series of ten randomly chosen patterns of sizes 3-6, such as triangles and circles. Our migration strategy reduces total network traffic by up to 50 percent compared to PowerGraph partitioning.

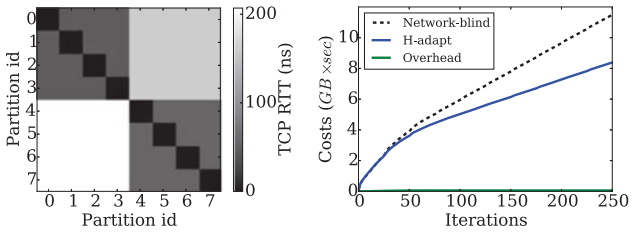
Replication Degree. A standard metric to measure the partitioning quality of vertex-cut algorithms is the replication degree, i.e., the number of vertex replicas in the system. In Fig. 11a , we performed 250 iterations of PageRank on the web graph [32] using the CloudM infrastructure. We measured replication degree in each iteration divided by the replication degree of a hash partitioned graph, as well as the current migration overhead divided by the total migration overhead (CDF). During the first 70 iterations replication degree is reduced by 60 percent, followed by saturation. Hence, H-adapt minimizes replication degree as a side effect when optimizing for total communication costs.

Latency. The partitioning problem is computationally hard and solving it during execution can hurt overall processing latency, despite the benefits of reduced communication. To evaluate how our dynamic repartitioning method H-adapt influences latency of graph processing, we



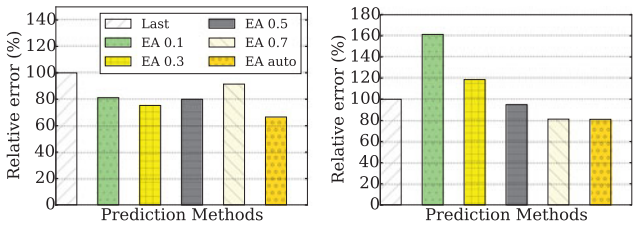
(a) PR on *Web*. (b) PR on *GoogleWeb*. (c) PR on *TwitterLarge*. (d) PR on *TwitterLarge*.

Fig. 11. (a) H-adapt reduces replication degree. (b) GrapH reduces total workload. (c)-(d) H-adapt reduces latency.



(a) Round-trip time in CloudM. (b) PR on *Web*.

Fig. 12. Heterogeneity in single data center, scale-up scenario.

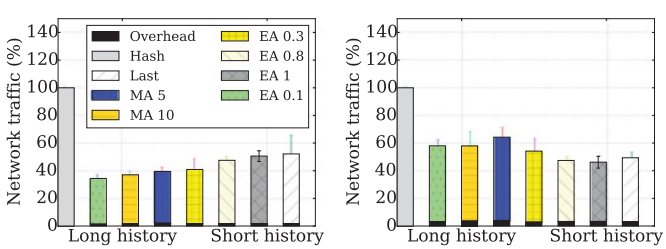


(a) SI on *Twitter*. (b) CA on grid (2,500 vertices).

Fig. 14. (a)-(b) Adaptive-α outperforms other prediction methods.

measured the latency per iteration of a single execution of PageRank on our shared memory machine. We used *Twitter-Large* with more than 1.4 billion edges. In Fig. 11c, we plot the latency per iteration for the Hash pre-partitioned graph (Hash) and for the Hash repartitioned graph using H-adapt (8 partitions). The figure highlights the trade-off between migration latency overhead and gain: there are four instances where H-adapt temporarily invests more time (approximately at iterations 3,25,41,48). However, investing time to compute and implement an improved partitioning results in large reductions of latency up to 60 percent. In Fig. 11d, we show the cumulated distribution function for latency to compare end-to-end latency of both methods. Surprisingly, the end-to-end latency for a single execution of PageRank improves by 10 percent using our dynamic repartitioning method H-adapt. Thus, compared to the predecessor method H-move [19], H-adapt reduces end-to-end latency by approximately 20 percent due to the novel methods of constant back-off and lock-free migration. Although included into the total runtime, the time spent on partitioning is 12%, averaged over all iterations. Note that we have executed only one instance of the PageRank algorithm, subsequent executions and long-running graph algorithms would benefit even more from the improved partitioning.

We also tested the impact of network-awareness on graph processing latency for the PageRank algorithm in CloudM. However, we observed a non-significant reduction of graph processing latency by less than 1 percent for H-adapt compared to its network-blind variant. We attribute this to the fact that our cost function minimizes the *weighted summed vertex traffic over all network links* (cf. Eq. (8)) but latency mainly depends on the bottleneck link with maximal latency because of the straggler problem [33]. In other words, at least one high-cost link between two workers residing on different machines experienced the same amount of traffic (although the *summed* high-cost network link traffic was reduced). However, tuning the cost function to minimize latency instead of communication costs is beyond the scope of this paper.



(a) PR on *Twitter*. (b) SI on *Twitter*.

Fig. 13. (a)-(b) Prediction methods influence migration efficiency.

Load Balancing. GraphH balances the *summed vertex traffic over all vertices on a partition* (cf. Eq. (3)). In Fig. 11b, we show the average worker workload after one PR execution (78 iterations) for PG and GraphH as well as the deviation of the workers from this average workload. GraphH leads to a slightly higher workload imbalance because we balance for (more volatile) vertex traffic, while PG balances the number of edges. However, because of the reduced overall communication overhead, GraphH's maximally loaded worker still has less workload than the least loaded worker in PG. Thus, GraphH reduces total workload by more than 60 percent.

Prediction Methods. Choosing the right prediction method for future vertex traffic is important for overall performance of our system. For instance, overestimating vertex traffic of vertex u leads to biased decisions towards migrating edges incident to vertex u. To learn about prediction accuracy, we compared three methods (cf. Section 3): last value (Last), moving average (MA) with window sizes 5 and 10, and exponential averaging (EA) with decay parameter $\alpha = 0.1, 0.3, 0.8, 1.0$ in Figs. 13a and 13b. We evaluated the reduction of total network traffic for PR and SI, compared to Hash. The position of the bars from left to right reflects the size of the considered history for prediction in descending order. For example, since *Last* considers only the last value, we plotted it on the right. Clearly, all prediction methods meet the goal of reducing overall network traffic. However, none of these methods leads to consistently better results for all algorithms, because of 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 the subsequent short-lived queries lead to diverse vertex traffic patterns.

We have seen that there is no single prediction method that is optimal for all algorithms. Thus, determining the best parameter α is a crucial task for effective vertex traffic prediction. In the following, we show that our method for automatically selecting α individually for each vertex consistently outperforms all other static (and global) parameter choices. In Figs. 14a and 14b, we compared the *relative prediction error* w.r.t. decay parameter α. We define the relative prediction error as the ratio of the absolute prediction errors of the tested method and the benchmark method *Last*. The absolute error of a prediction method is determined by predicting vertex traffic for the next w iterations and accumulating the difference between predicted and observed vertex traffic value, averaged over 5 randomly selected vertices. We performed evaluations for SI on *Twitter* and CA on a grid with 2500 vertices (and window sizes w of 10,5, and 3 respectively). On the right, we plotted the relative error for

our method Adaptive-α. Our experiments lead to two interesting conclusions. First, the extremely simple method $Last$ results in relatively robust vertex traffic predictions with high accuracy and low computational overhead. Second, Adaptive-α consistently outperformed all other prediction methods in terms of prediction accuracy while introducing little computational overhead (cf. Eq. (7)).

6 Related Work

Several distributed graph processing systems inspired our work. PowerGraph [5] suggests the GAS API based on vertex-cut partitioning. They provide a greedy streaming heuristic (*Coordinated*) for static vertex-cut partitioning, which we used for comparison. Petroni et al. (*HDRF* [21]) consider the vertex degree in order to find a minimum vertex-cut in the streaming setting arguing that optimal placement of low-degree vertices should be preferred. A similar argumentation is used in degree-based hashing (*DBH* [34]), where an edge is assigned to a partition based on the hash value of the edge's low-degree vertex. PowerLyra [35] extends PowerGraph with hybrid-cuts: cutting vertices with high degree and edges of low-degree vertices decreasing expensive replica communication overhead. These strategies minimize the replication degree, but ignore diverse and dynamic vertex traffic.

On the other hand, the graph systems Mizan [36] and GPS [10] propose adaptive edge-cut partitioning using vertex migration. Mizan considers the traffic sent via each edge to balance workload at runtime. Vaquero et al. [37] apply edge-cut to changing graphs using a decentralized algorithm for iterative vertex migration to avoid costly re-execution of static partitioning algorithms. Shang et al. [38] nicely identified and categorized three types of vertex activation patterns for graph processing workload: always-active-, traversal-, and multi-phase-style. They exploit these workload patterns to dynamically adjust the partitioning during computation. Yang et al. [39] (*Sedge*) improve localization of processing small-sized queries by introducing a two-level complimentary partitioning scheme using vertex replication. While these edge-cut systems adapt to changing graphs or traffic behaviors, they do not consider network topology and migration costs. Furthermore, these approaches focus on edge-cut and optimal edge-cuts can not be transformed into close-to-optimal vertex-cuts for graphs with high-degree vertices [40].

Surfer [12] tailors graph processing to the cloud by considering bandwidth unevenness to map graph partitions with a high number of inter-partition links to workers connected via high-bandwidth networks. However, they assume homogeneous traffic on each edge. GraphIVE [13] strives for a minimal *unbalanced* k-way vertex-cut for workers with heterogeneous computation and communication capabilities, in order to put more work to more powerful workers — searching for the optimal number of edges for each worker. This approach is orthogonal to heterogeneity-aware partitioning algorithms. Xu et al. [20] consider network and vertex weights to find a static minimal costs edge-cut. They do not consider adaptive vertex-cut, and vertex weights reflect only the number of executions, but not the real vertex traffic. Zheng et al. [41] propose ARGO, an architecture-aware edge-cut graph repartitioning method focusing on RDMA-enabled networks that also considers the amount of communication going over each edge and heterogeneous network costs. Similarly, GrapH is also applicable to the scenarios described by ARGO while focusing on vertex-cut partitioning and the GAS execution model.

General data processing in the geo-distributed setting is addressed by Pu et al. [17] and Jayalath et al. [18]. They argue that aggregating geographical distributed data into a single data center can significantly hurt overall data processing performance for MapReduce-like computations. LaCurts et al. [8] point out that considering network heterogeneity for an optimal task placement, improves overall end-to-end data analytics performance, even in a single data center. Therefore, they place communicating tasks in such a way that most communication flows over fast network links. However, task placement is orthogonal to graph partitioning and could be used on top of our system.

Finally, significant research efforts addressed the problem of parallelizing subgraph isomorphism (e.g., [3], [42]) and cellular automata (e.g., [1], [43]). However, we have not found any algorithm in the GAS programming interface despite the suitability of distributed graph processing systems for these kind of problems.

7 Conclusion

Modern graph processing systems use vertex-cut partitioning due to its superiority of partitioning real-world graphs. Existing partitioning methods minimize the replication degree, which is expected to be the dominant factor for communication costs. However, the underlying assumptions of uniform vertex traffic and network costs do not hold for many real-world applications. To this end, we propose GrapH, a graph processing system taking dynamic vertex traffic and diverse network costs into account to adaptively minimize communication costs of the vertex-cut at runtime. Our evaluations show, that GrapH outperforms PowerGraph's vertex-cut partitioning algorithm by up to than 60 percent w.r.t. communication costs, while improving end-to-end latency of graph computation by more than 10 percent.

References

[1] T. Suzumura, C. Houngkaew, and H. Kanezashi, "Towards billion-scale social simulations," in *Proc. Winter Simulation Conf.*, 2014, pp. 781–792.

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

[3] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *Proc. VLDB Endowment*, vol. 5, pp. 310–321, 2011.

[4] G. Malewicz, et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2010, pp. 135–146.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.

[6] 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. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[7] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 242–253.

[8] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware task placement for cloud applications," in *Proc. Conf. Internet Measurement Conf.*, 2013, pp. 191–204.

[9] A. Greenberg, et al., "VL2: A scalable and flexible data center network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 51–62, 2009.
[10] S. Salihoglu and J. Widom, "GPs: A graph processing system," in *Proc. Int. Conf. Scientific Statist. Database Manage.*, 2013, Art. no. 22.
[11] K. T. Tusscher, D. Noble, P. Noble, and A. Panfilov, "A model for human ventricular tissue," *Amer. J. Physiology-Heart Circulatory Physiology*, vol. 286, pp. H1573–H1589, 2004.
[12] 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. 3rd ACM Symp. Cloud Comput.*, 2012, Art. no. 3.
[13] D. Kumar, A. Raj, D. Patra, and D. Janakiram, "GraphIVE: Heterogeneity-aware adaptive graph partitioning in GraphLab," in *Proc. 43rd Int. Conf. Parallel Process. Workshops*, 2014, pp. 95–103.
[14] C. Peng, M. Kim, Z. Zhang, and H. Lei, "VDN: Virtual machine image distribution network for cloud data centers," in *Proc. IEEE Conf. Inf. Comput. Commun.*, 2012, pp. 181–189.
[15] I. Narayanan, A. Kansal, A. Sivasubramaniam, B. Urgaonkar, and S. Govindan, "Towards a leaner geo-distributed cloud infrastructure," in *Proc. 4th USENIX Conf. Hot Topics Cloud Comput.*, 2014, p. 3.
[16] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a geo-distributed data-intensive world," *Proc. Conf. Innovative Data Syst. Res.*, 2015, pp. 1087–1092.
[17] Q. Pu, et al., "Low latency geo-distributed data analytics," in *Proc. ACM SIGCOMM Conf.*, 2015, pp. 421–434.
[18] C. Jayalath, J. Stephen, and P. Eugster, "From the cloud to the atmosphere: Running MapReduce across data centers," *IEEE Trans. Comput.*, vol. 63, no. 1, pp. 74–87, Jan. 2014.
[19] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel, "Graph: Heterogeneity-aware graph computation with adaptive partitioning," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst. Workshops*, 2016, pp. 118–128.
[20] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao, "Heterogeneous environment aware streaming graph partitioning," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 6, pp. 1560–1572, Jun. 2015.
[21] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "HDRF: Stream-based partitioning for power-law graphs," in *Proc. 18th ACM Conf. Inf. Knowl. Manage.*, 2015, pp. 243–252.
[22] G. Hamerly and C. Elkan, "Learning the k in k-means," *Advances Neu. Inf. Process. Syst.*, pp. 281–288, 2004.
[23] T. Stützle, "Iterated local search for the quadratic assignment problem," *Eur. J. Oper. Res.*, 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," *Concurrency Comput.: Practice Exper.*, vol. 26, pp. 2053–2078, 2014.
[25] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.
[26] N. Ganguly, B. K. Sikdar, A. Deutsch, G. Canright, and P. P. Chaudhuri, "A survey on cellular automata," 2003.
[27] Z. Riaz, F. Dürr, and K. Rothermel, "Optimized location update protocols for secure and efficient position sharing," in *Int. Conf. Workshops Networked Syst.*, 2015, pp. 1–8.
[28] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *USENIX Conf. Annu. Techn. Conf.*, 2015, pp. 375–386.
[29] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *ACM Sigplan Notices*, vol. 48, pp. 135–146, 2013.
[30] C. Guo, et al., "Pingmesh: A large-scale system for data center network latency measurement and analysis," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, pp. 139–152, 2015.
[31] M. Wu, et al., "GraM: Scaling graph computation to the trillions," in *Proc. 6th ACM Cloud Comput.*, 2015, pp. 408–421.
[32] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proc. 23rd Int. Conf. World Wide Web*, 2011, pp. 587–596.
[33] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," in *Proc. 30th Int. Conf. Very Large Data Bases*, vol. 7, pp. 1047–1058, 2014.
[34] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 1673–1681.
[35] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2015, Art. no. 1.
[36] 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 *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 169–182.
[37] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "Adaptive partitioning for large-scale dynamic graphs," in *Proc. 37th IEEE Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 144–153.
[38] Z. Shang and J. X. Yu, "Catch the wind: Graph workload balancing on cloud," in *Proc. IEEE 25th Int. Conf. Data Eng.*, 2013, pp. 553–564.
[39] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proc. ACM SIGMOD Int. Conf. Manag. Data* 2012, pp. 517–528.
[40] U. Feige, M. Hajiaghayi, and J. R. Lee, "Improved approximation algorithms for minimum weight vertex separators," *SIAM J. Comput.*, vol. 38, pp. 629–657, 2008.
[41] A. Zheng, A. Labrinidis, P. K. Chrysanthis, and J. Lange, "Argo: Architecture-aware graph partitioning," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 284–293.
[42] Y. Yuan, G. Wang, J. Y. Xu, and L. Chen, "Efficient distributed subgraph similarity matching," *Very Large Data Bases J.*, vol. 24, pp. 369–394, 2015.
[43] P. B. Hansen, "Parallel cellular automata: A model program for computational science," *Concurrency: Practice Experience*, vol. 5, pp. 425–48, 1993.

Christian Mayer received the diploma degree in computer science from the University of Stuttgart. He is currently working toward the PhD degree in the Distributed Systems research group, University of Stuttgart. His research interests include distributed, parallel data processing, graph processing in the cloud, and graph partitioning.

Muhammad Adnan Tariq received the PhD degree from the University of Stuttgart, Germany. He is currently working as a postdoctoral research fellow in the Distributed Systems research group, University of Stuttgart. His research interests include event- based systems, service-aware adaptive overlay networks, QoS, and security.

Ruben Mayer received the diploma degree in computer science from the University of Stuttgart. He has recently submitted his PhD thesis and is now working as a postdoctoral researcher with the Distributed Systems research group, University of Stuttgart. His research interests include mainly in distributed event-based systems and fog computing.

Kurt Rothermel received the doctoral degree in computer science from the University of Stuttgart, in 1985. Between 1986-1987, he was a postdoctoral fellow with the IBM Almaden Research Center in San Jose and then joined IBM European Networking Center in Heidelberg. He left IBM in 1990 to become a professor of computer science with the University of Stuttgart, where he now leads the distributed systems research group. His research interests include distributed and mobile systems, computer networks, and sensor networks.