Application Driven Graph Partitioning

Wenfei Fan[1,2,3], Ruochun Jin[1], Muyang Liu[1], Ping Lu[2], Xiaojian Luo[4], Ruiqi Xu[1],
Qiang Yin[4], Wenyuan Yu[4], Jingren Zhou[4]
[1]University of Edinburgh [2]BDBC, Beihang University [3]SICS, Shenzhen University [4]Alibaba Group
{wenfei@inf., ruochun.jin@, muyang.liu@, ruiqi.xu@}ed.ac.uk, luping@buaa.edu.cn,
{lxj193371, qiang.yq, wenyuan.ywy, jingren.zhou}@alibaba-inc.com

ABSTRACT

Graph partitioning is crucial to parallel computations on large graphs. The choice of partitioning strategies has strong impact on not only the performance of graph algorithms, but also the design of the algorithms. For an algorithm of our interest, what partitioning strategy fits it the best and improves its parallel execution? Is it possible to develop graph algorithms with *partition transparency*, such that the algorithms work under different partitions without changes?

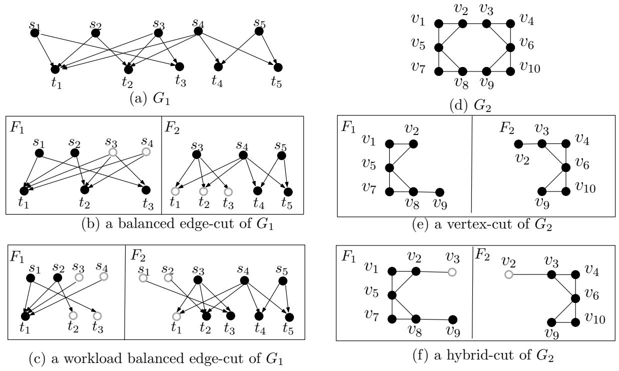
This paper aims to answer these questions. We propose an application-driven hybrid partitioning strategy that, given a graph algorithm $\mathcal{A}$, learns a cost model for $\mathcal{A}$ as polynomial regression. We develop partitioners that given the learned cost model, refine an edge-cut or vertex-cut partition to a hybrid partition and reduce the parallel cost of $\mathcal{A}$. Moreover, we identify a general condition under which graph-centric algorithms are partition transparent. We show that a number of graph algorithms can be made partition transparent. Using real-life and synthetic graphs, we experimentally verify that our partitioning strategy improves the performance of a variety of graph computations, up to 22.5 times.

KEYWORDS

graph partition; partition transparency; machine learning

ACM Reference Format:
Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Xiaojian Luo, Ruiqi Xu, Qiang Yin, Wenyuan Yu and Jingren Zhou. 2020. Application Driven Graph Partitioning . In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389745

SIGMOD'20, June 14–19, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389745


s_1 s_2 s_3 s_4 s_5
t_1 t_2 t_3 t_4 t_5
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10}
F_1
F_2

(a) G_1

(d) G_2

(b) a balanced edge-cut of G_1

(e) a vertex-cut of G_2

(c) a workload balanced edge-cut of G_1

(f) a hybrid-cut of G_2

Figure 1: CN **and** TC

1 INTRODUCTION

To handle real-life graphs, graph partitioning is often a must. It is to cut a large graph G into smaller fragments and distribute the fragments to a cluster of processors (*a.k.a.* workers) so that the workers have even workload for parallel computations and their communication is minimized.

A number of partitioning algorithms (*a.k.a.* partitioners) are already developed. These algorithms are often either *edge-cut* [5, 27], which evenly partitions vertices and cuts edges, or *vertex-cut* [9, 19, 28], which evenly partitions edges by replicating vertices. There have also been *hybrid* partitioners, which cut both edges and vertices [13, 15, 30, 51].

These partitioners typically follow two quality criteria, *balance and replication factors.* To balance workload and reduce synchronization overhead, a partitioner often seeks to cut a graph into fragments of "even" sizes, and reduce replicated edges and vertices. In the real world, however, such criteria do not always capture the bottleneck factors that affect the performance of parallel graph algorithms, since the computation and communication patterns of algorithms vary.

Example 1: Consider the following real-life examples.

(1) Common neighbor. Consider running Common Neighbor (CN) [31] on a directed graph G_1 shown in Fig. 1(a). CN computes the number of common neighbors for each pair of vertices. It is widely used in link prediction, product recommendation and fraud detection [14, 31]. To simplify the discussion, we consider outgoing common neighbors, where a vertex u is an outgoing common neighbor of v_1 and v_2 if there exist edges from v_1 and v_2 to u. A common program

of CN works as follows: for each triple (u, v_1, v_2) such that both v_1 and v_2 link to u, it increases the common neighbor count for the pair (v_1, v_2). Suppose that G_1 is partitioned into fragments F_1 and F_2 as shown in Fig. 1(b). The partition is well balanced *w.r.t.* both vertices and edges, since each fragment has 5 inner vertices (solid discs) and 9 edges.

(a) However, the workload of CN on the partition of Fig. 1(b) is skewed. For the program of CN above, each vertex u contributes at most $\frac{1}{2}d^+(u)(d^+(u) - 1)$ triples of the form (u, v_1, v_2), where $d^+(u)$ denotes the in-degree of node u, and v_1 and v_2 are incoming neighbors of u. Thus the computational cost on a fragment F_i is determined by the aggregation $\sum_{u \in F_i} \frac{1}{2}d^+(u)(d^+(u) - 1)$. As a result, the workloads on F_1 and F_2 are 10 and 2, respectively. That is, the maximum load of CN is 5X of the minimum one, even when the partition of Fig. 1(b) is balanced *w.r.t.* both vertices and edges.

(b) Figure 1(c) depicts another partition of G_1. The vertices and edges are not as balanced as that of Fig. 1(b) since F_1 has 3 vertices and 6 edges, while F_2 has 7 vertices and 11 edges. This said, the workloads of CN on F_1 and F_2 are both 6, which are well balanced. Taken together with Fig. 1 (b), this shows that static metrics such as vertex and edge balance do not ensure workload balance for applications such as CN.

(2) Triangle counting. Consider counting all triangles (TC) in the undirected graph G_2 of Fig. 1(d). TC has been used in clustering [46], cycle detection [22] and transitivity [34]. A program of TC works as follows: for each node v and each edge (v, u) of v, it counts the number of triangles that involve the edge (v, u) (see Section 7.2 for more details). Graph G_2 is vertex cut evenly into F_1 and F_2 by splitting v_2 and v_9, as depicted in Fig. 1(e). Observe the following.

(a) Communication is required when not all neighbors of a vertex are stored locally, *i.e.,* split vertices v_2 and v_9. Let $N = \{v_1, v_3, v_5\}$ be the set of neighbors of v_2; to count triangles involving v_2, all pairs of vertices in $N \times N$ must be checked. It has to inspect the remote edge (v_2, v_3); similarly for v_9.

(b) Replication helps reduce communication cost. Consider the partition of G_2 of Fig. 1(f). As opposed to Fig. 1(e), it replicates a vertex v_3 and an edge (v_2, v_3) at F_1. When running program TC on it, to count triangles of v_2, no communication is needed since all verification can be done locally. To reduce communication when counting triangles of v_9, one can further replicate edge (v_8, v_9) and vertex v_8 at F_2. □

Worse still, an algorithm developed under an edge-cut partition may not work under vertex-cut, and vice versa. Hence, when developing a parallel algorithm for a graph computation problem, one has to pick which partitioning strategy to use in advance. We may have to rewrite our algorithms and switch to the other if the strategy picked does not work well.

Example 2: Consider PageRank [10]. In each iteration, each vertex collects the page-rank score of its incoming neighbors, and updates its own score accordingly. It is easy to write such an algorithm under edge-cut partition in which each vertex keeps its incoming edges [16]. However, the algorithm does not work on a vertex-cut partition since vertices may not have all its incoming edges locally. The algorithm has to be rewritten to gather incoming scores in advance or introduce a nontrivial aggregation mechanism (see Section 7.3). □

These examples give rise to several questions. For an algorithm $\mathcal{A}$ of our interest, what parameters should we consider to partition graphs for $\mathcal{A}$? After all, the primary goal is to improve parallel execution of $\mathcal{A}$ no matter whether the partition is edge-cut, vertex-cut or hybrid, regardless of its balance ratio and replication factor. Can we learn partition parameters for $\mathcal{A}$? If so, how can we partition graphs based on the learned parameters? Moreover, can we make $\mathcal{A}$ work under different partitions without requiring any change?

Contributions & organization. This paper aims to answer these questions. We propose an application-driven partitioning strategy, and a notion of partition transparency.

(1) Application-driven partitioning (Section 3). We propose a *hybrid partitioning strategy* to find partitions tailored for a graph algorithm $\mathcal{A}$. We introduce a cost model to characterize the computational and communication patterns of $\mathcal{A}$. We formalize the *application-driven partition problem* (ADP), which aims to find a partition that reduces the cost of $\mathcal{A}$ based on its cost model. We show that ADP is NP-complete.

(2) Cost model learning (Section 4). We show how to learn the cost model for a given algorithm $\mathcal{A}$. We approximate the cost model as polynomial regression following [47], which has proven effective in practice [21]. We train the model with the stochastic gradient descent algorithm. The learned cost model can be applied to different graphs on which $\mathcal{A}$ runs.

(3) Hybrid partitioners (Section 5). We develop parallel partitioners ParE2H and ParV2H that given algorithm $\mathcal{A}$ and a graph G, develop a hybrid partition of G for $\mathcal{A}$ guided by the learned cost model of $\mathcal{A}$. We show that ParE2H (resp. ParV2H) refines an edge-cut (resp. vertex-cut) to a hybrid partition that accommodates the cost patterns of $\mathcal{A}$.

(4) Partition transparency (Section 6). We identify a condition under which an algorithm $\mathcal{A}$ is guaranteed *partition transparent*, *i.e.,* $\mathcal{A}$ works under edge-cut, vertex-cut and hybrid partitions without requiring any change to $\mathcal{A}$. We prove the condition for graph-centric programs of GRAPE [17] and vertex-centric programs of PowerGraph [19]. Note that this condition is *not mandatory* for our approach. Our approach can handle non-transparent $\mathcal{A}$ (see Section 5). Moreover, one can rewrite $\mathcal{A}$ and make it transparent (Section 7).

(5) Transparent algorithms (Section 7). As case studies, we develop partition transparent algorithms for common neighbor (CN), triangle counting (TC) and PageRank (PR). They work correctly no matter what partitions are used.

(6) Experimental study (Section 8). Using real-life and synthetic graphs, we empirically verify the effectiveness, scalability and efficiency of ParE2H and ParV2H. We find the following. (a) The application-driven partitioners are effective. Over real-life graphs, they improve the performance of CN, TC and PR by 7.0, 4.6 and 2.8 times on average, respectively, up to 22.5 times. (b) They are efficient, taking 11.9% and 11.3% of the total partitioning time on average to refine edge-cut and vertex-cut partitions, respectively. (c) They scale well with graphs, taking 59.7s and 32.5s, respectively, on graphs of $500M$ vertices and $6B$ edges, with 96 workers. (d) With small training cost, the learned cost models are accurate, with MSRE (mean squared relative error) ≤ 0.11.

Related work. Various algorithms have been developed for edge-cut and vertex-cut partitions (see [7, 11] for surveys). Edge-cut (resp. vertex-cut) aims to partition vertices (resp. edges) into disjoint subsets of even sizes and reduce replication. Exact edge-cut algorithms of [5, 29] compute balanced partitions and cut minimum edges. METIS [25, 26] and its parallel version ParMETIS [24] adopt a multi-level heuristic scheme and are edge-cut partitioners widely used in practice. Other popular heuristics include parallel partitioners such as XtraPuLP [43] and stream partitioners FENNEL [44]. Vertex-cut partitioners include spectral algorithm of [39] and heuristics Grid [23], SHEEP [32], NE [49] and HDRF [38].

Edge-cut promotes locality: for each vertex v in graph G, it keeps all edges emanating from v in the same fragment; however, it often leads to imbalanced partitions, especially when G is skewed, *i.e.,* when a small portion of G connects to a large fraction of G. In contrast, vertex-cut makes it easier to balance partitions, but may have a lower level of locality and increase communication cost for low-degree vertices.

To rectify these, there has been work on hybrid partitioners. PowerLyra [13] and IOGP [15] combine edge-cut and vertex-cut by cutting only high-degree vertices, controlled by a user-defined threshold. TopoX [30] not only splits high-degree vertices, but also merges neighboring low-degree vertices into super nodes to prevent splitting such vertices. Gemini [51] and MDBGP [6] balance hybrid workload by combining vertex and edge loads based on a balancing metric.

There have also been attempts [20, 33, 48] to speed up distributed graph computations by replicating various parts of graphs across partitions, such that read operations can be performed locally without communication.

This work differs from the prior methods in the following.

(1) This work is the first study of partition transparency and provides the first condition under which graph algorithms are transparent to underlying partitions (Section 6).

(2) We propose an application-driven partitioning strategy that given an algorithm $\mathcal{A}$, learns a cost model beyond balance and replication, such that we can tailor graph partitions to speed up parallel execution of $\mathcal{A}$ (Sections 3, 4 and 5).

(3) As opposed to prior hybrid partitioners, our partitioners are guided by a cost model of a given algorithm $\mathcal{A}$, and generate partitions tailored for the best performance of $\mathcal{A}$ (Section 5). We show that such partitions can be readily computed by extending existing edge-cut or vertex-cut partitioners, *i.e.,* there is no need to develop another partitioner starting from scratch. In addition, the partitioners strike a balance between replication and computation speedup.

(4) We develop the first ML models that train cost models for given algorithms (Section 4), as opposed prior partitioners that adopt one-size-fits-all static metrics, follow intuitions [30, 51] or manually pick parameters [6, 13, 15].

2 PRELIMINARIES

We start with a review of basic notations. We consider (un)directed graphs $G = (V, E)$, where V is a finite set of vertices, and $E \subseteq V \times V$ is its set of edges.

Partitions. Given a natural number n, a *n-cut hybrid partition* $\mathsf{HP}(n) = (F_1, \ldots, F_n)$ of a graph G, or simply a partition of G, divides G into n small fragments $F_1, \ldots, F_n$ such that (a) $F_i = (V_i, E_i)$, (b) $V = \bigcup_{i=1}^{n} V_i$, and (c) $E = \bigcup_{i=1}^{n} E_i$.

Denote by E^v (resp. E_i^v) the set of edges incident to vertex v in G (resp. F_i). We also use the following notations.

(1) A vertex v is *v-cut* in $\mathsf{HP}(n)$ if the set of edges incident to v is not "complete" at any F_i, *i.e.,* $E^v \neq E_i^v$ ($\forall i \in [1, n]$). Each copy of such v in $\mathsf{HP}(n)$ is called a *v-cut node* of v.

(2) A vertex v is *e-cut* if there exists a fragment F_i such that all edges incident to v are included in F_i, *i.e.,* $E_i^v = E^v$. When there exist multiple copies of v in $\mathsf{HP}(n)$, we refer to the copy in F_i as an *e-cut node* and the others as *dummy nodes* of v.

(3) Denote by $F_i.O = \{v \in V_i \mid v \in V_j \wedge i \neq j\}$ the set of border nodes of F_i. Intuitively a border node is replicated among fragments. Let $\mathcal{F}.O = \bigcup_{i=1}^{n} F_i.O$. We associate a *master node mapping* with $\mathsf{HP}(n)$ for vertices in $\mathcal{F}.O$. More specifically, for each vertex $v \in \mathcal{F}.O$, the mapping treats one copy of v as its *master* and the other copies as its *mirrors.*

Example 3: Consider the hybrid partition (F_1, F_2) depicted in Fig. 1(f) of graph G_2 of Fig. 1(d). Observe the following.

(1) Vertex v_9 is v-cut since edge (v_9, v_6) and (v_9, v_{10}) are missing from fragment F_1, and edge (v_9, v_8) is missing from F_2.

(2) Vertex v_2 is e-cut, since all edges incident to v_2 are included in F_1. The copy of v_2 at F_1 is an e-cut node, while

G, F_i	graph and a fragment of G
V_i (resp. E_i)	the vertex set (resp. edge set) of fragment F_i
E^v (resp. E_i^v)	the set of edges incident to v in G (resp. F_i)
$F_i.O$ (resp. $\mathcal{F}.O$)	the set of border nodes in F_i (resp. $\mathsf{HP}(n)$)
$d_L^+, d_L^-, d_G^+, d_G^-, D$	various vertex degree metrics (Section 3.1)
$h_\mathcal{A}, g_\mathcal{A}$	cost functions of $\mathcal{A}$ (Section 3.1)
$C^h_\mathcal{A}(F_i), C^g_\mathcal{A}(F_i)$	computational and communication cost of F_i
$C_\mathcal{A}(F_i)$	the cost of $\mathcal{A}$ on F_i

Table 1: Notations

the copy at F_2 is a dummy node. Similarly v_3 is also e-cut. Vertices $v_1, v_4, v_5, v_6, v_7, v_8, v_{10}$ are also e-cut since they are not replicated and edges incident to them are all kept locally.

(3) $F_1.O = F_2.O = \{v_2, v_3, v_9\}$; thus $\mathcal{F}.O = \{v_2, v_3, v_9\}$. If a master node mapping maps v_2 and v_3 of F_1 and v_9 of F_2 as masters, then v_2 and v_3 of F_2 and v_9 of F_1 are mirrors. □

Special cases. Two special cases of hybrid partitions are edge-cut partitions [5, 27] and vertex-cut partitions [19, 28].

(1) Partition $\mathsf{HP}(n)$ is *edge-cut* if (i) all vertices are e-cut; and (ii) the e-cut node sets of the fragments are pairwise disjoint.

(2) Partition $\mathsf{HP}(n)$ is *vertex-cut* if the edge sets are disjoint, *i.e.,* $E_i \cap E_j = \emptyset$ for $i \neq j$, while v-cut nodes are replicated.

Example 4: Consider the hybrid partitions depicted in Fig. 1.

(1) The partition (F_1, F_2) depicted in Fig. 1(b) is an edge-cut partition of graph G_1 given in Fig. 1(a), since (i) all vertices of G_1 are e-cut; and (ii) the e-cut node sets of F_1 and F_2 are disjoint, *i.e.,* $\{s_1, s_2, t_1, t_2, t_3\} \cap \{s_3, s_4, s_5, t_4, t_5\} = \emptyset$. Note that the e-cut (resp. dummy) nodes are depicted as black (resp. white). Another edge-cut partition of G_1 is shown in Fig. 1(c).

(2) Partition (F_1, F_2) of Fig. 1(e) is a vertex-cut partition of G_2 given in Fig. 1(d), since the edge sets of F_1 and F_2 are disjoint.

(3) Partition (F_1, F_2) of Fig. 1(f) is neither edge-cut (since v_9 is v-cut) nor vertex-cut (since edge (v_2, v_3) is replicated). □

Quality. The *partition quality* of hybrid partition $\mathsf{HP}(n)$ is traditionally characterized by two factors defined as follows.

Replication ratio. Denote by $f_v = \sum_{i=1}^{n} |V_i|/|V|$ the vertex replication ratio, and by $f_e = \sum_{i=1}^{n} |E_i|/|E|$ the edge replication ratio. Usually vertex-cut partitioning aims to minimize f_v and edge-cut partitioning aims to minimize f_e, since the number of v-cut nodes and cut-edges can be expressed as $(f_v - 1)|V|$ (vertex-cut) and $(f_e - 1)|E|$ (edge-cut).

Balance factor. A hybrid partition $\mathsf{HP}(n)$ is said λ_v-balanced *w.r.t.* vertices if $|V_i| \leq (1+\lambda_v) \sum_{j=1}^{n} |V_j|/n$ for all $i \in [1, n]$, *i.e.,* the number of vertices of each fragment is not too deviated from the average. Similarly, $\mathsf{HP}(n)$ is λ_e-balanced *w.r.t.* edges if $|E_i| \leq (1 + \lambda_e) \sum_{j=1}^{n} |E_j|/n$ for all $i \in [1, n]$.

The notations of this paper are summarized in Table 1.

3 APPLICATION DRIVEN PARTITIONING

The primary goal of partitioners is to speed up parallel computations of our interest. As shown in Example 1, traditional metrics such as replication ratio and balance factor do not suffice to capture the variety of computational and communication patterns of graph algorithms. Hence, a partition that fits one algorithm may not work well for another.

This motivates us to propose an application-driven partitioning strategy. We first introduce a model that captures the computational and communication patterns of a given algorithm $\mathcal{A}$ (Section 3.1). We then present a partitioning strategy that partitions graphs to minimize the parallel computation cost of $\mathcal{A}$ based on the cost model of $\mathcal{A}$ (Section 3.2).

3.1 A Cost Model

Given a graph algorithm $\mathcal{A}$, we estimate the cost of $\mathcal{A}$ under a partition $\mathsf{HP}(n)$ of a graph G in terms of a *computation cost function* $h_\mathcal{A}$ and a *communication cost function* $g_\mathcal{A}$.

Cost model. Let $\mathcal{X} = \{x_1, \ldots, x_k\}$ be a set of *metric variables*, where each $x_i \in \mathcal{X}$ is associated with a vertex metric in $\mathsf{HP}(n)$ to be given shortly. Functions $h_\mathcal{A}$ and $g_\mathcal{A}$ are two multivariate functions over $\mathcal{X}$. Given a vertex v, $h_\mathcal{A}$ and $g_\mathcal{A}$ estimate the computational cost $h_\mathcal{A}(\mathcal{X}(v))$ and communication cost $g_\mathcal{A}(\mathcal{X}(v))$ incurred by v, respectively. Denote by $C^h_\mathcal{A}(F_i)$ and $C^g_\mathcal{A}(F_i)$ the computational cost and communication cost of algorithm $\mathcal{A}$ on fragment F_i, respectively. Then the cost of $\mathcal{A}$ on fragment F_i is estimated as

$$C_\mathcal{A}(F_i) = C^h_\mathcal{A}(F_i) + C^g_\mathcal{A}(F_i). \tag{1}$$

We next show how to estimate costs $C^h_\mathcal{A}(F_i)$ and $C^g_\mathcal{A}(F_i)$ of $\mathcal{A}$ on fragment F_i using cost functions $h_\mathcal{A}$ and $g_\mathcal{A}$.

Computational cost $C^h_\mathcal{A}(F_i)$. On fragment F_i, we define

$$C^h_\mathcal{A}(F_i) = \sum_{v \in F_i \wedge v \text{ is a non-dummy node}} h_\mathcal{A}(\mathcal{X}(v)). \tag{2}$$

Intuitively, the computational cost of $\mathcal{A}$ on a fragment F_i is amortized among its e-cut and v-cut nodes (*i.e.,* non-dummy nodes, see Section 2), and $C^h_\mathcal{A}(F_i)$ is the aggregation of the costs incurred by its vertices estimated by $h_\mathcal{A}$.

Communication cost $C^g_\mathcal{A}(F_i)$. Unlike $C^h_\mathcal{A}(F_i)$, $C^g_\mathcal{A}(F_i)$ is usually incurred by vertices replicated in $\mathsf{HP}(n)$. Here we measure the communication cost incurred by master nodes:

$$C^g_\mathcal{A}(F_i) = \sum_{v \in F_i.O \wedge v \text{ is a master node}} g_\mathcal{A}(\mathcal{X}(v)) \tag{3}$$

Intuitively, for a vertex $v \in F_i.O$, its communication and synchronization often take place at its master copy, which is responsible for receiving possible updates from its mirrors and sending the aggregation back to its mirrors [17, 19].

Remark. (a) While computational cost is amortized among vertices, the cost model is applicable to not only vertex-centric models, but also edge-centric [42] and graph-centric

[17] models; *e.g.,* edge-centric algorithms access the data of the endpoints of edges, although the computation is distributed on edges. Thus we can use the sum of amortized cost on each vertex to approximate the total computational cost.

(b) In Eq. (1), the communication cost does not include the part overlapped with computation, *i.e.,* it only measures the *non-overlapping* time. Under synchronous parallel model (BSP), each round starts with a computation phase followed by a communication phase, and the overlap is small. Under asynchronous model, $C^{g}_{\mathcal{A}}(F_i)$ excludes overlap.

Metric variables. We next identify a set $\mathcal{X}$ of vertex metric variables that affect the computational and communication costs of most graph algorithms. For a vertex v in fragment $F_i = (V_i, E_i)$, $\mathcal{X}$ includes the following metric variables:

- $d^+_L(v) = |\{u \mid (u, v) \in E_i\}|$, *i.e.,* the in-degree of v in F_i;
- $d^-_L(v) = |\{u \mid (v, u) \in E_i\}|$, *i.e.,* the out-degree of v in F_i;
- $d^+_G(v) = |\{u \mid (u, v) \in E\}|$, *i.e.,* the in-degree of v in G;
- $d^-_G(v) = |\{u \mid (v, u) \in E\}|$, *i.e.,* the out-degree of v in G;
- $r(v) = |\{j \mid v \in V_j \wedge j \neq i\}|$, *i.e.,* the number of mirrors of v among all fragments;
- $D = \sum_{v\in V} d^+_G(v)/|V| = \sum_{v\in V} d^-_G(v)/|V|$, *i.e.,* the average in/out degree of G, which is a constant metric for v.

For undirected graphs, $d^+_L(v) = d^-_L(v)$ and $d^+_G(v) = d^-_G(v)$.

Intuitively, the metric variables above have impact on the computational and communication cost incurred by a vertex. For instance, $d^+_L(v)$ (resp. $d^-_L(v)$) determines the number of incoming (resp. outgoing) neighbors that v may access during computation; $r(v)$ decides whether synchronization is necessary; $d^+_G(v)$, $d^-_G(v)$ and D may affect the size of messages synchronized between the master of v and its mirrors.

In the sequel we use the following metric variable set:

$$\mathcal{X} = \{d^+_L, d^-_L, d^+_G, d^-_G, r, D\}.$$

Note that $\mathcal{X}$ above only includes variables that affect the cost of most graph algorithms. For a specific algorithm $\mathcal{A}$, one can either extend $\mathcal{X}$ or pick a subset of $\mathcal{X}$, depending on $\mathcal{A}$.

Example 5: The cost functions for CN and TC on a partition $\mathsf{HP}(n) = (F_1, \ldots, F_n)$ can be defined as follows.

(1) $h_{\mathsf{CN}} = \alpha d^+_L(v) d^+_G(v) + \beta d^+_L(v) + \gamma$ and $g_{\mathsf{CN}} = \delta D d^-_G(v)$ for some positive α, β, γ and δ. Function h_{CN} indicates that in an edge-cut partition (*i.e.,* $d^+_L(v) = d^+_G(v)$), the computation cost of a vertex v is dominated by the "square" of its incoming degree (recall Example 1). Function g_{CN} estimates the communication cost incurred by a master. That is, given a vertex v, the number of triples (v, w, u) to be aggregated can be estimated by $Dd^-_G(v)$, where w is a common neighbor of v and u.

(2) $h_{\mathsf{TC}}=\alpha d^+_L(v)+\beta d^+_L(v) d^+_G(v)$ and $g_{\mathsf{TC}}=\gamma I(v) d^+_G(v) r(v)$ for some positive α, β and γ. Here $I(v)$ is an e-cut indicator such that $I(v)$=1 if v is *not* an e-cut node in F_i, and $I(v)$=0 otherwise. To avoid counting the same triangles repeatedly, we only check the neighbors of v with smaller degrees (see Section 7.2). Then, (a) h_{TC} estimates the cost for checking the neighbors of v, $\alpha d^+_L(v)$ is for searching the small-degree neighbors of v, and $\beta d^+_L(v) d^-_G(v)$ is for counting triangles with these neighbors; (b) g_{TC} estimates communication incurred by v. If v is an e-cut node, then its computation can be done locally; if v is a v-cut or dummy node, then extra communication for verifying neighbors of v is proportional to $d^+_G(v) r(v)$. Based on g_{TC}, to reduce the communication of TC, we make more vertices e-cut as indicated in Fig. 1(f). □

Balance factor revised. Incorporating parallel computation cost, we revise balance factor. For an algorithm $\mathcal{A}$, we say that a partition $\mathsf{HP}(n)$ of graph G is *λ-balanced for $\mathcal{A}$* if

$$C_{\mathcal{A}}(F_i) \leq (1 + \lambda) \sum_{j=1}^{n} C_{\mathcal{A}}(F_j)/n \quad (\forall i \in [1, n]).$$

That is, the cost of $\mathcal{A}$ on each fragment is not far from the average. Here $C_{\mathcal{A}}(F_i)$ is the cost of $\mathcal{A}$ on F_i (see Equation (1)).

Example 6: Continuing with Example 5, under an edge-cut partition, the computation cost function h_{CN} tells us that the computation workload of CN is proportional to the sum of squares of the incoming degrees of the vertices in a fragment. As shown in Fig. 1(c), this is the key to balance the workload of CN, rather than the number of edges and vertices. □

3.2 A Graph Partitioning Strategy

We now present an *application-driven partitioning strategy*. Given an algorithm $\mathcal{A}$ of our interest, it works as follows:

(1) it first learns the cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of $\mathcal{A}$; then

(2) when $\mathcal{A}$ is applied to any graph G, given a natural number n, it computes a partition $\mathsf{HP}(n)$ of G such that it minimizes the parallel cost $\max_{i\in[1,n]} C_{\mathcal{A}}(F_i)$ of $\mathcal{A}$.

As opposed to prior partitioning strategies, this strategy targets a given algorithm $\mathcal{A}$ and guides partitions by the cost model of $\mathcal{A}$, not by the traditional one-size-fits-all metrics.

The strategy is carried out by the following:

- *Polynomial regression models*: given an algorithm $\mathcal{A}$, learn the cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of $\mathcal{A}$ (Section 4).
- *Hybrid partitioners*: given a graph G and a number n, compute a partition $\mathsf{HP}(n)$ of G to minimize the parallel cost of $\mathcal{A}$ estimated by $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ (Section 5).

Complexity. The decision problem for application-driven partitioning, denoted by ADP, can be stated as follows.

- Input: A graph G, a number $n > 0$, a cost budget B, and two cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of $\mathcal{A}$.
- Question: Is there a $\mathsf{HP}(n)$ of G s.t. the parallel cost of $\mathcal{A}$ under $\mathsf{HP}(n)$ is bounded by B, *i.e.,* $\max_{i\in[1,n]} C_{\mathcal{A}}(F_i) \leq B$.

It is not surprising that problem ADP is intractable.

Theorem 1: *ADP is NP-complete.* □

Proof: Clearly ADP is in NP. Its NP-hardness can be verified by reduction from the set partition problem [18]. □

Remark. To simplify the discussion, our partitioning strategy targets important applications that run *repeatedly*, *e.g.,* graph pattern matching with various patterns. It can be extended to cope with multiple applications running on the same graph: (a) assign weights to multiple tasks based on their importance, frequency and cost; (b) extend Eq (1) to an overall cost model for these tasks, by incorporating weight coefficients; and (c) find a hybrid partition to reduce the overall cost.

4 LEARNING COST FUNCTIONS

We next show how to learn $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ for a given algorithm $\mathcal{A}$. We first give a multivariate polynomial regression model for $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ with metric variables $\mathcal{X}$. We then show how we collect training data and train the model. Since the objective function and learning algorithm for $g_{\mathcal{A}}$ are similar to their $h_{\mathcal{A}}$ counterparts, below we focus on learning $h_{\mathcal{A}}$. Learning $g_{\mathcal{A}}$ differs from $h_{\mathcal{A}}$ in only training data.

Cost function as polynomial regression. Given metric variables $\mathcal{X}(v) = \{x_1(v), \ldots, x_k(v)\}$ of vertex v (Section 3.1), we model $h_{\mathcal{A}}$ as a polynomial function $h_{\mathcal{A}}(\mathcal{X}(v)) = \sum_{\gamma_j \in \Gamma} \omega_j \gamma_j(v)$, where Γ is the set of all terms in the expansion of $(1+\sum_{x_i(v)\in\mathcal{X}(v)} x_i(v))^p$, ω_j is the weight of $\gamma_j(v)$ and $p \in \mathbb{N}$ controls the highest order of the polynomial expression.

The learning algorithm employs training samples, each denoted as $[\mathcal{X}(v_k), t_k]$, which is extracted from the running log of $\mathcal{A}$ and includes computational cost t_k of each node v_k, to adjust each weight parameter ω_j so that every $h_{\mathcal{A}}(\mathcal{X}(v_k))$ approximates t_k. Using *mean squared relative error* (MSRE) [35] as loss function, the learning objective for $h_{\mathcal{A}}$ is written as

$$\min_{\Omega} \quad \frac{1}{|\mathcal{D}_{h_{\mathcal{A}}}|} \sum_{[\mathcal{X}(v_k), t_k] \in \mathcal{D}_{h_{\mathcal{A}}}} (h_{\mathcal{A}}(\frac{\mathcal{X}(v_k) - t_k}{t_k})^2) + \sum_{\omega_i \in \Omega} |\omega_i|,$$

where $\mathcal{D}_{h_{\mathcal{A}}}$ is the set of all training samples for $h_{\mathcal{A}}$, $|\mathcal{D}_{h_{\mathcal{A}}}|$ is the number of training samples, $\Omega = \{\omega_1, \ldots, \omega_{|\Gamma|}\}$, and $\sum_{\omega_i \in \Omega} |\omega_i|$ is the penalty function to prevent over-fitting [8].

The reason for implementing $h_{\mathcal{A}}$ as a polynomial function is twofold. (1) In theory polynomials can closely approximate a continuous function defined on a closed interval [47]. Polynomial regression has proven effective in predicting computational cost in practice [21]. (2) The polynomial model is explainable compared with other black-box ML models [4, 21, 41], *e.g.,* it gives the cost calculation expression and shows which variable in $\mathcal{X}$ contributes most to the cost.

Model training. Given an algorithm $\mathcal{A}$, we first run $\mathcal{A}$ on real-life and synthetic graphs to collect $\mathcal{D}_{h_{\mathcal{A}}}$, and then train the regression model with $\mathcal{D}_{h_{\mathcal{A}}}$ by the stochastic gradient descent (SGD) algorithm [8]. When collecting $\mathcal{D}_{h_{\mathcal{A}}}$, we only pick nodes that are used in computation. For example, we only record the metric variables $\mathcal{X}$ and the computation time of nodes t_1, t_2, t_3, t_4 and t_5 in Fig. 1(b) to collect training data for h_{CN}, since only nodes with incoming edges are involved in the computation. To get $\mathcal{D}_{g_{\mathcal{A}}}$ for $g_{\mathcal{A}}$, (1) we only collect the communication costs of master nodes on fragment borders; and (2) we exclude its overlapping portion with computation for more accurate learning (see Section 3.1).

We find that cost $C_{\mathcal{A}}$ heavily depends on algorithm $\mathcal{A}$. To make $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ generic, we put no restrictions on either graphs used in the training or how the graphs are partitioned.

Training cost reduction. To reduce the training cost while preserving high prediction accuracy, we shorten $\mathcal{X}$ by selecting influential variables for the prediction. This variable selection can be automatic via feature selection algorithms [12]. We employ "L1-based feature selection" algorithm [37] in our approach. In addition, domain knowledge can also be incorporated into feature selection. Note that domain knowledge is *not mandatory* but it helps improve model prediction and makes training easier. For example, since $d^+_L(v)$ and $d^+_G(v)$ of a node v are adequate to estimate the computational cost of CN, other metric variables in $\mathcal{X}$ are not included in h_{CN}. Moreover, we can even specify that $h_{\text{CN}}(v) = \omega_1 d^+_L(v) d^+_G(v) + \omega_2 d^+_L(v) + \omega_3$. This yields merely three weight parameters ω to learn, and reduces the learning cost.

Example 7: We have seen h_{CN} and g_{TC} in Example 5. We next illustrate how these cost functions are learned for CN and TC, respectively; the learning of h_{TC} and g_{CN} is similar.

(1) CN. First, we run CN on 10 graphs randomly partitioned by either edge-cut or vertex-cut, and record $[\mathcal{X}(v_i), t_i]$ of each vertex v_i as training samples. While these samples may not cover extreme scenarios, such as super nodes with high degrees, the training data suffices for our tasks as the impact of the extreme cases is diluted by cost aggregation in the fragment. With 80% (resp. 20%) of a total 100,000 samples for training (resp. testing) and the highest order p set as 2, we learned $h_{\text{CN}} = 9.23 \times 10^{-5} d^+_L(v) d^+_G(v) + 1.04 \times 10^{-6} d^+_L(v) + 1.02 \times 10^{-6}$, where the testing MSRE is 0.023. The learned h_{CN} shows that the computational cost of a vertex is dominated by the "square" of its in-degree, consistent with its complexity.

(2) TC. We learned $g_{\text{TC}} = 8.42 \times 10^{-5} d_G(v) r(v) I(v)$ with 80,000 samples. As samples for g_{TC}, we only pick master nodes since other vertices incur little communication. The learned g_{TC} shows that the communication cost is determined by the degrees of vertices and the number of mirrors. □

5 HYBRID PARTITIONERS

We have seen how to learn cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ for a graph algorithm $\mathcal{A}$ (Section 4). The second component of our application-driven partitioning strategy (Section 3.2) is a partitioner that, given a graph G, finds a partition of G to reduce the parallel cost of $\mathcal{A}$, guided by $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$.

We next develop such partitioners. Instead of developing yet another partitioner, we show that edge-cut and vertex-cut partitions can be revised to consent to the cost functions. We present two such algorithms, E2H and V2H. Given an edge-cut (resp. vertex-cut) partition of G produced by any widely-used partitioner, E2H (resp. V2H) improves it and produces a hybrid partition $\mathsf{HP}(n)$ to fit the cost pattern of $\mathcal{A}$.

Below we first present the sequential version of algorithms E2H and V2H in Sections 5.1 and 5.2, respectively. We then show how to parallelize E2H and V2H in Section 5.3.

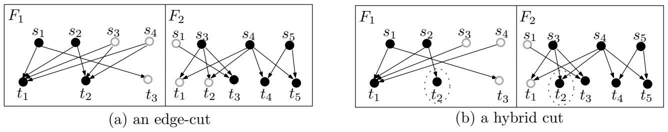
5.1 From Edge Cut to Hybrid Cut

Edge-cut promotes locality, *i.e.,* each node in an edge-cut tends to keep all its incident edges locally. However, this may lead to imbalanced workload. The reasons are twofold. First, real-life graphs often follow power-law, *i.e.,* a small number of vertices are adjacent to a large fraction of edges. It is hard to balance workload while retaining the locality. Second, as shown in Example 1, computational cost patterns vary for algorithms. A partition with balanced workload for one algorithm may still exhibit skew workload for another.

Overview of E2H. Given an edge-cut $\mathsf{HP}_E(n)$ and two cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of $\mathcal{A}$, E2H extends $\mathsf{HP}_E(n)$ to a hybrid partition $\mathsf{HP}(n)$ to reduce the parallel cost $\max_{i\in[1,n]} C_{\mathcal{A}}(F_i)$ of $\mathcal{A}$ in two stages. Recall that $C_{\mathcal{A}}(F_i) = C^h_{\mathcal{A}}(F_i) + C^g_{\mathcal{A}}(F_i)$. Guided by $h_{\mathcal{A}}$, the first stage of E2H balances computational workload to reduce $\max_{i\in[1,n]} C^h_{\mathcal{A}}(F_i)$ of $\mathcal{A}$ (and hence $\max_{i\in[1,n]} C_{\mathcal{A}}(F_i)$). Guided by $g_{\mathcal{A}}$, its second stage reduces $\max_{i\in[1,n]} C^g_{\mathcal{A}}(F_i)$ by redistributing communication cost.

Balance computational cost. This stage consists of two phases, namely, EMigrate and ESplit. To balance computational cost, both phases migrate nodes and edges from overloaded fragment to underloaded fragments. To this end, we estimate a budget B, *e.g.,* average computational cost of fragments. A fragment F_i is *overloaded* if its computational cost exceeds B, *i.e.,* $C^h_{\mathcal{A}}(F_i) > B$; and F_i is *underloaded* if $C^h_{\mathcal{A}}(F_i) \leq B$.

EMigrate. This phase reduces $\max_{i\in[1,n]} C^h_{\mathcal{A}}(F_i)$ by migrating e-cut nodes and their incident edges from overloaded fragments to underloaded ones. To retain the locality of edge-cut partitions, for each overloaded fragment, E2H identifies a coherent sub-fragment within budget B and marks the rest of e-cut nodes and their incident edges for migration. Denote by (v, E') a migration candidate, where v is marked for migration and E' is the set of edges incident to v. In each iteration, E2H invokes an EMigrate operation to move a migration candidate (v, E') to an underloaded fragment F_j if $C^h_{\mathcal{A}}(F_j \cup \{(v, E')\} \leq B$. This phase terminates when no more EMigrate operations can be performed. The remaining migration candidates will be processed in the next phase.



(a) an edge-cut

(b) a hybrid cut

Figure 2: EMigrate and ESplit

Example 8: Consider the edge-cut partition of Fig. 1(b). We move an e-cut node t_3 from fragment F_1 to F_2 via EMigrate. This yields another edge-cut shown in Fig. 2(a). Note that the migration also leaves a dummy copy (a white node) in F_1 since t_3 is linked to another e-cut node s_1 of F_1. □

ESplit. This phase cuts e-cut nodes into v-cut nodes and migrates them to underloaded fragments to further balance workload. For heavily skewed graphs, Emigrate may not suffice since the computational cost incurred by an e-cut node v (*e.g.,* nodes that are incident to a large number of edges) may already exceed the budget. In that case, ESplit cuts v into multiple v-cut nodes and splits the computation among fragments. Unlike EMigrate, ESplit splits v and only migrates a subset of v's incident edges. The ESplit phase only processes the migration candidates left from the EMigrate phase. It terminates when all migration candidates have been processed.

Example 9: Applying ESplit to e-cut node t_2 in the partition of Fig. 2(a), a possible outcome is depicted in Fig. 2(b). Now node t_2 of F_1 is cut and two edges (s_3, t_2) and (s_4, t_2) are migrated from fragment F_1 to F_2. The resulting partition is hybrid. It is not an edge-cut since t_2 does not keeps all incident edges locally; it is not a vertex-cut since there exists edge duplication, *e.g.,* (s_1, t_3) is in both F_1 and F_2. □

Redistribute communication cost. Communication cost is often incurred by master nodes (see Equation (3)). Recall that $C_{\mathcal{A}}(F_i) = C^h_{\mathcal{A}}(F_i) + C^g_{\mathcal{A}}(F_i)$. To further reduce the parallel cost $\max_{i\in[1,n]} C_{\mathcal{A}}(F_i)$ of $\mathcal{A}$, E2H utilizes an MAssign phase to update master node assignments and redistribute communication cost. Note that MAssign does not increase the computational costs of partitions obtained by EMigrate and ESplit, since it only adjusts the master node mapping.

MAssign. Initially, MAssign marks all border nodes in $\mathcal{F}.O$ as unassigned and set $C^g_{\mathcal{A}}(F_i) = 0$ for $i \in [1, n]$. It processes the master node assignment in an one-pass fashion. Consider $v \in \mathcal{F}.O$ and let $F_{i_1}, \ldots, F_{i_k}$ be the fragments in which v resides. Denote by $g^{i_1}_{\mathcal{A}}(v), \ldots g^{i_k}_{\mathcal{A}}(v)$ the communication cost incurred by v if the master of v is assigned to one of $F_{i_1}, \ldots F_{i_k}$, respectively. To minimize $\max_{i\in[1,n]} C_{\mathcal{A}}(F_i)$, the master of v is assigned to the fragment with minimal cost. More specifically, the master of v is assigned to F_{i^*}, where

$$i^* = \mathrm{argmin}_{j\in\{i_1,\ldots,i_k\}} C^h_{\mathcal{A}}(F_j) + C^g_{\mathcal{A}}(F_j) + g^j_{\mathcal{A}}(v). \quad (4)$$

Once the master of v is assigned to fragment F_{i^*}, MAssign includes the communication cost $g^{i^*}_{\mathcal{A}}(v)$ of v in $C^g_{\mathcal{A}}(F_{i^*})$.

```
Input:  Edge-cut $\mathsf{HP}_E(n) = (F_1, \dots, F_n)$, cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$.
Output: Revised hybrid partition $\mathsf{HP}(n) = (F_1, \dots, F_n)$.
1.  $B \leftarrow \sum_{i=1}^{n} C^h_{\mathcal{A}}(F_i)/n$;   $O \leftarrow \emptyset$;   $U \leftarrow \emptyset$;
2.  for each $i \in [1, n]$ do
3.      if $C^h_{\mathcal{A}}(F_i) > B$ then
4.          $O \leftarrow O \cup \{F_i\}$;   $S_i \leftarrow$ GetCandidates$(F_i, B)$;
5.      else $U \leftarrow U \cup \{F_i\}$;   $S_i \leftarrow \emptyset$;
6.  for each $F_i \in O$ and each $(v, E^v_i) \in S_i$ do    /* phase EMigrate */
7.      for each $F_j \in U$ do
8.          if $C^h_{\mathcal{A}}(F_j \cup \{(v, E^v_i)\} \leq B$ then
9.              migrates $(v, E^v_i)$ to $F_j$;   $S_i \leftarrow S_i \setminus \{(v, E^v_i)\}$;
10.             break;
11. for each $F_i \in O$ and each $(v, E^v_i) \in S_i$ do    /* phase ESplit */
12.     for each $e \in E^v_i$ do
13.         $t \leftarrow \mathrm{argmin}_{j \in [1,n]} C^h_{\mathcal{A}}(F_j)$;   migrates $(v, \{e\})$ to $F_t$;
14.     $S_i \leftarrow S_i \setminus \{(v, E^v_i)\}$
15. adjust master node mapping;                             /* phase MAssign */
16. return $\mathsf{HP}(n) = (F_1, \dots, F_n)$;
```

Figure 3: Algorithm E2H

Algorithm E2H. Putting these together, we present algorithm E2H in Fig. 3. Given an edge-cut partition $\mathsf{HP}_E(n)$ and cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of algorithm $\mathcal{A}$, E2H extends $\mathsf{HP}_E(n)$ to a hybrid partition $\mathsf{HP}(n)$ by reducing the parallel cost of $\mathcal{A}$. Using function $h_{\mathcal{A}}$, E2H first sets a computational cost budget B for each fragment (line 1). Based on B, it divides the fragments into two sets: overload fragments O and underloaded ones U (lines 2-5). For each overloaded $F_i \in O$, E2H identifies a set of e-cut nodes S_i as candidates for migration by procedure GetCandidates (line 4; see below).

To balance workload, E2H first carries out the EMigrate phase (lines 6-10). Each time, it selects an e-cut node and its incident edges from migration candidates and reallocates them to an underloaded fragment F_u such that the move does not make the cost of F_u exceed budget estimated in terms of $h_{\mathcal{A}}$. When no more Emigrate operation can be applied, E2H conducts the ESplit phase (lines 11-14). Each time, it selects an edge associated to the rest of candidate e-cut nodes and migrates it to the fragment with minimal computational cost.

At last, E2H revises the master node mapping to further reduce parallel cost of $\mathcal{A}$ on $\mathsf{HP}(n)$ via MAssign (line 15).

Remark. Phase Esplit is applicable only if the application $\mathcal{A}$ is partition transparent (see Section 6). If otherwise, *i.e.,* if $\mathcal{A}$ only works under edge-cut, then we *can drop* Esplit, and E2H yields edge-cut partitions instead. As shown in Exp-1 (see Section 8), such partitions still substantially improve the original partitions although not as effective as hybrid-cut.

Procedure GetCandidates. Given a fragment F_i and a budget B, GetCandidates identifies a set of vertices and edges as migration candidates. As an effort to retain the locality of F_i, GetCandidates identifies a coherent sub-fragment F'_i of F_i within the budget B. To do that, GetCandidates first performs a BFS traversal on F_i. Following the BFS order, it includes nodes and their edges to F'_i within budget B in a greedy manner. More specifically, a vertex v and its incident edges E' are added to F'_i if $C^h_{\mathcal{A}}(F'_i \cup \{(v, E')\}) \leq B$. The vertices and edges excluded from F'_i are returned as migration candidates.

Example 10: We show how E2H works on the edge-cut of Fig. 1(b) based on h_{CN} and g_{CN}=$5.57\times10^{-5}Dd^-_G$ (Example 7).

(1) E2H estimates $C^h_{\mathsf{CN}}(F_1) = 2.69 \times 10^{-3}$ms and $C^h_{\mathsf{CN}}(F_2) = 7.45 \times 10^{-4}$ ms for F_1 and F_2. With the cost budget $B = 1.72 \times 10^{-3}$ms, F_1 is overloaded while F_2 is underloaded.

(2) To balance the workload, E2H uses GetCandidates to identify migration candidates in fragment F_1. Let t_1, s_1, s_2, t_3, t_2 be a BFS order. Observe that the sub-fragment induced by $\{t_1, s_1, s_2\}$ is a maximal one within the budget. Then e-cut nodes t_3 and t_2 are marked as migration candidates.

(3) Suppose that E2H first migrates t_3 by EMigrate from F_1 to F_2. This yields the partition shown in Fig. 2(a). This increases the cost $C^h_{\mathsf{CN}}(F_2)$ of F_2 from 7.45×10^{-4}ms to 1.12×10^{-3}ms. E2H then tries to migrate t_2 from F_1 to F_2. However, this operation is aborted as it would exceed F_2's budget.

(4) E2H then applies ESplit to cut t_2 and migrates edges incident to u_2 in a greedy manner. It migrates two edges (s_3, t_2) and (s_4, t_2) from F_1 to F_2 as demonstrated in Example 9. It ends up with the hybrid partition shown in Fig. 2(b).

(5) To further reduce the execution cost of CN, E2H updates the master node mapping. By g_{CN}, only vertices with $d^+_G > 0$ incur communication. Thus we only consider master mapping for s_1, s_3 and s_4. By MAssign, the masters of s_1 and s_4 are assigned to F_2, while the master of s_3 is assigned to F_1.

(6) The original edge-cut has parallel cost $\max_{i\in[1,2]} C_{\mathsf{CN}}(F_i) = 2.98 \times 10^{-3}$ms if the masters of s_3 and s_4 are mapped to F_2. In contrast, the hybrid-cut obtained via E2H has parallel cost $\max_{i\in[1,2]} C_{\mathsf{CN}}(F_i) = \max\{1.98, 2.11\} \times 10^{-3}\text{ms} = 2.11 \times 10^{-3}$ms. Thus E2H indeed reduces the parallel cost of CN. □

5.2 From Vertex Cut to Hybrid Cut

Compared with edge-cut, vertex-cut has better balance. However, most vertex-cut partitioners aim to balance edge size. As we have seen in Example 1, this does not suffice for workload balance. Also observe that vertex-cut often incurs larger communication cost due to bad locality [19].

Overview of V2H. Given a vertex-cut $\mathsf{HP}_V(n)$ and cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of algorithm $\mathcal{A}$, V2H produces a hybrid partition $\mathsf{HP}(n)$ by adjusting $\mathsf{HP}_V(n)$, to reduce the parallel cost $\max_{i\in[1,n]} C_{\mathcal{A}}(F_i)$ of $\mathcal{A}$. It also has two stages. Guided by $h_{\mathcal{A}}$, it first not only balances computational workload but also reduces communication cost. Guided by $g_{\mathcal{A}}$, its second stage redistributes communication cost. Below we focus on the first stage; the second stage is similar to MAssign in E2H.

Balance computational cost. This stage consists of two phases, namely, VMigrate and VMerge. As in E2H, it first estimates a cost budget B, and classifies fragments as overloaded and underloaded. Then VMigrate migrates v-cut nodes from overloaded fragments to underloaded ones to balance workload. VMerge makes v-cut nodes to e-cut nodes to further balance workload and reduce communication cost.

VMigrate. This phase migrates v-cut nodes and their incident edges from overloaded fragments to underloaded ones. To both improve locality and reduce communication cost, VMigrate adopts a different approach to identifying migration candidates. A v-cut node v and its associated edges E' are migrated from an overloaded F_i to an underloaded F_j if

- ○ there exists another v-cut node (v, E'') in F_j; and
- ○ $C^h_{\mathcal{A}}(F_j \cup \{(v, E')\})\} \leq B$.

This reduces the replication of v by one. Thus VMigrate reduces both $\max_{i\in[1,n]} C^h_{\mathcal{A}}(F_i)$ and communication cost. The process stops when no VMigrate operation can be applied.

VMerge. This phase iteratively merges v-cut nodes into e-cut nodes. In each iteration, VMerge (a) first selects a fragment F_i with the smallest $C^h_{\mathcal{A}}(F_i)$ and a v-cut node v in F_i; and (b) changes v to an e-cut node by either moving or replicating all v's missing edges in F_i based on the respective costs. This change is valid if the new $C^h_{\mathcal{A}}(F_i)$ does not exceed the budget. To reduce the computational cost incurred by v, it marks the copies of v in fragments other than F_i as dummy nodes. That is, by converting a v-cut node to an e-cut one, it balances workload by reallocating computation to the fragment with minimal cost $C^h_{\mathcal{A}}(F_i)$. VMerge continuously merges v-cut nodes until no valid changes can be made.

Example 11: Applying VMerge to v-cut node v_2 in F_1 of Fig. 1(e), it replicates edge (v_2, v_3) at F_1. Now v_2 becomes an e-cut node, and v_2 in F_2 is marked as a dummy copy. This yields a hybrid partition depicted in Fig. 1(f). As remarked in Example 1, this reduces the communication of TC. □

5.3 Parallelization

We next show how to parallelize E2H and V2H.

Setting. We adopt a shared-nothing distributed setting. Initially, fragments $F_1, \ldots, F_n$ of the input edge-cut (resp. vertex-cut) partition are distributed to n workers $P_1, \ldots, P_n$, respectively. The workers run under BSP model [45], which separates the computation into supersteps. Based on $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$, each worker maintains a shared state, including costs $C_{\mathcal{A}}(F_i)$ and $C^g_{\mathcal{A}}(F_i)$ of each F_i and other cost-related metrics for the vertices and edges processed. In each superstep, each worker conducts a small batch of computation to refine the partition and synchronize the shared state via messages.

Below we only show how to parallelize phase EMigrate and MAssgin. The parallelization of other phases is similar.

Parallel EMigrate. In a superstep, each overloaded worker sends a small batch of migration candidates to underloaded workers in a round-robin manner. A worker is *overloaded* (resp. *underloaded*) if it hosts an overloaded (resp. underloaded) fragment. Denote by k the number of underloaded fragments and let $P_{i_1}, \ldots, P_{i_k}$ be the associated underloaded workers. An overloaded worker selects k migration candidates and sends them to $P_{i_1}, \ldots, P_{i_k}$ in parallel, respectively. Upon receiving migration candidates, underloaded worker P_{i_j} ($j \in [1, k]$) processes them one by one. Worker P_{i_j} accepts a candidate if it does not exceed the budget; otherwise P_{i_j} rejects it. The original sender includes the rejected candidate in the next batch and sends it to worker P_{i_ℓ}, where $\ell \equiv (j+1)$ mod k. The process proceeds until each migration candidate is either accepted by some P_{i_j} or rejected by all $P_{i_1}, \ldots P_{i_k}$.

Parallel MAssign. In a superstep, each worker selects a small batch of unassigned vertices in $F_i.O$, and adjusts master nodes by Eq. (4) above in parallel, based on the shared $C^h_{\mathcal{A}}(F_i)$ and $C^g_{\mathcal{A}}(F_i)$ of each F_i. The adjustments are synchronized among workers to resolve conflicts and update the shared state. The process stops after all nodes in $\mathcal{F}.O$ are processed.

6 PARTITION TRANSPARENCY

We next study partition transparency. We consider *w.l.o.g.* algorithms in the graph-centric model of GRAPE [17]. We first review the model of GRAPE (Section 6.1), and then define partition transparency and provide a condition under which graph algorithms are partition transparent (Section 6.1).

6.1 Graph Centric Programming

Consider a class $\boldsymbol{Q}$ of queries. Given a query $Q \in \boldsymbol{Q}$ and a partition HP(n) of a graph G, a parallel program for $\boldsymbol{Q}$ computes the set $Q(G)$ of answers to Q in G.

PIE algorithms. To develop a parallel algorithm for $\boldsymbol{Q}$ with GRAPE, one only needs to specify three functions.

(1) PEval: a *sequential* algorithm that given a query $Q \in \boldsymbol{Q}$ and a graph G, computes the answer $Q(G)$ to Q in G.

(2) IncEval: a *sequential incremental* algorithm that given Q, G, $Q(G)$ and updates ΔG to G, computes updates ΔO to the old output $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$, where $G \oplus \Delta G$ denotes G updated by ΔG [40].

(3) Assemble: a function that collects partial answers computed locally at each worker by PEval and IncEval, and assembles the partial results into complete answer $Q(G)$.

The three functions are referred to as *a* PIE *program for* $\boldsymbol{Q}$ (PEval, IncEval and Assemble). PEval and IncEval can be any *existing sequential* (incremental) algorithms for $\boldsymbol{Q}$.

The only additions are the following declarations in PEval.

(a) Update parameters. PEval declares (a) a set C_i of vertices in fragment F_i as the *update region* of F_i, *e.g.*, vertices in $F_i.O$,

and (b) *status variables* $\bar{x}$ for C_i. We denote by $C_i.\bar{x}$ the set of *update parameters* of F_i, *i.e.,* the status variables associated with the vertices in C_i. As will be seen shortly, $C_i.\bar{x}$ marks candidates to be updated by incremental steps IncEval.

(b) Aggregate functions. PEval also specifies an aggregate function f_{aggr}, *e.g.,* min and max, for conflict resolution, *i.e.,* to resolve conflicts when multiple workers attempt to assign different values to the same update parameter.

Parallel computation. Given a graph G, GRAPE partitions G into $\mathsf{HP}(n) = (F_1, \ldots, F_n)$ and distributes the fragments across n workers $(P_1, \ldots, P_n)$, respectively. Upon receiving a query Q at master P_0 (a designated worker), GRAPE posts Q to all workers and computes $Q(G)$ under BSP as follows.

(1) Partial evaluation (**PEval**). In the first superstep, GRAPE computes partial results $R_i^0 = \mathsf{PEval}(Q, F_i)$ over fragment F_i at each worker P_i, in parallel ($i \in [1, n]$). Here R_i^r denotes partial results in superstep r at worker P_i. At the end, P_i sends the set $C_i.\bar{x}$ of update parameters to master P_0 as a message.

For each status variable $x \in C_i.\bar{x}$, master P_0 collects the multi-set S_x of values from messages of all workers, and computes $x_{\text{aggr}} = f_{\text{aggr}}(S_x)$ by applying the aggregate function f_{aggr} declared in PEval. It generates message M_i to worker P_i, which includes only those $f_{\text{aggr}}(S_x)$'s such that $f_{\text{aggr}}(S_x) \neq x$, *i.e.,* only the *changed* values of the update parameters of F_i.

(2) Incremental computation (**IncEval**). In superstep $r + 1$, upon receiving message M_i, worker P_i invokes IncEval to *incrementally* compute $R_i^{r+1} = \mathsf{IncEval}(Q, R_i^r, F_i, M_i)$ by *treating message* M_i *as updates*, in parallel for $i \in [1, n]$. At the end of the process, P_i sends a message to P_0 that consists of *updated values* of $C_i.\bar{x}$, if any. After receiving messages from all workers, master P_0 deduces a message M_i just like in PEval. It sends message M_i to worker P_i in the next superstep.

(3) Termination (**Assemble**). The computation terminates when it reaches a fixpoint, *i.e.,* $R_i^{r+1} = R_i^r$ for all $i \in [1, n]$. At this time, GRAPE invokes Assemble at P_0, which pulls partial results from all workers, takes a union of the partial results, and gets the final result at P_0, denoted by $\mathcal{A}(Q, G)$.

6.2 Condition for Partition Transparency

PIE programs have been developed for vertex-cut and edge-cut. As shown in Example 2, PIE programs developed for edge-cut may not work under vertex-cut and vice versa.

Partition transparency. We say that a PIE program $\mathcal{A}$ is *partition transparent* if $\mathcal{A}$ works under edge-cut, vertex-cut and hybrid partitions without requiring any change to $\mathcal{A}$.

That is, we can uniformly use the same algorithm without worrying about the choice of graph partitioning strategies.

Conditions. We next present conditions for a PIE program to be partition transparent. We use the following notations. (a) Assume the existence of a partial order $\preceq$ on partial results R_i^l. (b) Denote by $G_1 \sqsubseteq G_2$ if graph G_1 is a subgraph of G_2.

We say that PEval (resp. IncEval) is *monotonic* if for all queries $Q \in \mathcal{Q}$ and graphs $G_1, G_2, G_1 \sqsubseteq G_2$ (resp. $R_i^s \preceq R_i^t$) implies that $\mathsf{PEval}(Q, G_1) \preceq \mathsf{PEval}(Q, G_2)$ (resp. $R_i^{s+1} \preceq R_i^{t+1}$). Here R_i^s and R_i^t are partial results in (possibly different) runs of $\mathcal{A}$.

Transparency condition. We give two conditions for $\mathcal{A}$.

P1 PEval and IncEval are monotonic.

P2 $\mathcal{A}$ is *correct under vertex cut, i.e.,* for any graph G, query Q and under any vertex-cut partition of G, $\mathcal{A}(Q, G) = Q(G)$.

Theorem 2: *A* PIE *algorithm* $\mathcal{A}$ *is partition transparent if* $\mathcal{A}$ *satisfies conditions* P1 *and* P2. □

Proof: We show that for any graph G, query $Q \in \mathcal{Q}$ and hybrid partition $\mathsf{HP}(n) = (F_1, \ldots, F_n)$ of G, $\mathcal{A}(Q, G) = Q(G)$. Denote by $\mathcal{A}(Q, \mathsf{HP}(n))$ the result of running $\mathcal{A}$ on $\mathsf{HP}(n)$.

(1) First consider a hybrid partition $\mathsf{HP}^G(n) = (G, \ldots, G)$ that duplicates G. One can verify that $\mathcal{A}(Q, \mathsf{HP}^G(n)) = Q(G)$ under the computation model of GRAPE (Section 6.1). Moreover, since $F_1, \ldots, F_n$ are subgraphs of G, $\mathcal{A}(Q, \mathsf{HP}(n)) \preceq \mathcal{A}(Q, \mathsf{HP}^G(n))$ by P1. Thus $\mathcal{A}(Q, \mathsf{HP}(n)) \preceq Q(G)$.

(2) Now consider vertex-cut partition $\mathsf{HP}^r(n) = (F_1', \ldots, F_n')$ by removing duplicated edges from $\mathsf{HP}(n)$. Since F_i' is a subgraph of F_i ($i \in [1, n]$), $\mathcal{A}(Q, \mathsf{HP}^r(n)) \preceq \mathcal{A}(Q, \mathsf{HP}(n))$ by P1. Because $\mathcal{A}$ is correct under vertex cut (P2), we have that $\mathcal{A}(Q, \mathsf{HP}^r(n)) = Q(G)$. Hence $Q(G) \preceq \mathcal{A}(Q, \mathsf{HP}(n))$. □

Vertex-centric model. We next present a condition for vertex-centric algorithms to be partition transparent.

As shown in [16, 17], a vertex-centric algorithm is a specific PIE program. To see this, consider the GAS model [19]. Such a vertex-centric algorithm $\mathcal{B}$ can be modeled as a PIE program when each fragment consists of a single vertex; it is *not* necessarily partition transparent, *e.g.,* the PR algorithm.

Corollary 3: *A vertex-centric algorithm* $\mathcal{B}$ *of the GAS model is partition transparent if* $\mathcal{B}$ *satisfies conditions* P1 *and* P2. □

7 TRANSPARENT ALGORITHMS

As case studies, we next provide PIE algorithms for common neighbor, triangle counting and PageRank. We show that these programs are partition transparent, *i.e.,* the same algorithms work under vertex-cut, edge-cut and hybrid-cut.

7.1 Common Neighbor

We start with common neighbor (CN; see Example 1). Denote by $\Gamma^+(v)$ (resp. $\Gamma^-(v)$) the set of outgoing (resp. incoming) neighbors of vertex v. The problem is stated as follows.

- Input: A directed graph $G = (V, E)$.
- Output: The count $\mathsf{CN}(u, v) = |\Gamma^+(u) \cap \Gamma^+(v)|$ of outgoing common neighbors for all pairs $(u, v) \in V \times V$.

(1) Algorithm. We outline a PIE program for CN. For each vertex v, PEval declares a status variable $v.x$ to maintain its incoming neighbors $\Gamma^{-}(v)$, and defines the set $F_i.O$ of border nodes as the update region C_i of fragment F_i.

At each F_i, PEval does the following: (a) for each master vertex v, it increases $\mathsf{CN}(u,w)$ for its incoming neighbors u and w in F_i, and (b) for each non-master vertex u in $F_i.O$, $u.x$ collects its local incoming neighbors. At the end of PEval, aggregate function f_{aggr} takes the union of $u.x$ for border nodes across different fragments. This collects remote changes $u.\Delta x$ to the neighbor set of border master node u of F_i.

Upon receiving such $u.\Delta x$ via messages, IncEval incrementally identifies pairs that share u as their common outgoing neighbor, and increases their count accordingly.

The process iterates until no changes can be incurred. At this point Assemble collects counts on master vertices and computes the sum of $\mathsf{CN}(u,v)$ for each pair $(u,v) \in V \times V$.

(2) Transparency. The PIE program is partition-transparent by Theorem 2. It is easy to verify that the program satisfies condition P2. For P1, PEval and IncEval are monotonic because when $G_1 \sqsubseteq G_2$, $\mathsf{CN}(u,v)$ in G_2 is larger than in G_1 since each vertex in G_2 has no less incoming neighbors.

7.2 Triangle Counting

We next study the problem of *triangle counting* (TC):

- Input: An undirected graph $G = (V,E)$.
- Output: The number of all triples $(u,v,w) \in V \times V \times V$ such that $(u,v) \in E$, $(u,w) \in E$ and $(v,w) \in E$.

(1) Algorithm. Our PIE program declares a status variable $v.x$ for each vertex v to store its neighbors. For each fragment F_i, its update region C_i is the set $F_i.O$ of border nodes.

For each vertex v in fragment F_i, PEval adds v to $u.x$ for each neighbor u of v if the degree of u is larger than that of v (*i.e.,* $d_G(v) < d_G(u)$). This is to avoid counting the same triangles repeatedly (*e.g.,* (u,v,w) and (v,w,u) are the same triangle). At the end of PEval, f_{aggr} takes the union of $v.x$ for border nodes v of F_i, and collects in $v.x$ all remote neighbors.

Upon receiving messages, IncEval updates the status variables of those border nodes that are not e-cut, and counts the triangles. More specifically, for each master vertex v in fragment F_i, IncEval scans $v.x$ and $u.x$ for those neighbors u of v such that $d_G(v){>}d_G(u)$, and increases the count of triangles when there exists a common vertex in $u.x$ and $v.x$.

The computation terminates when no more changes can be made to any $C_i.\bar{x}$ by IncEval. At this point, Assemble computes the sum of triangle counts across all fragments.

(2) Transparency. By Theorem 2, the PIE program is partition transparent. It satisfies condition P2. For P1, PEval and IncEval are monotonic since for $G_1{\sqsubseteq}G_2$, $v.x$ in G_2 is no less than $v.x$ in G_1, as more triangles are in G_2 than in G_1.

7.3 PageRank

Next we consider PageRank (PR) for ranking Web pages and links. It takes as input a directed graph $G = (V,E)$ and computes a ranking score $p(v)$ for each $v \in V$, defined as follows:

$$p(v) = \rho \sum_{(u,v)\in E} \frac{p(u)}{d^{+}_{G}(v)} + (1-\rho), \tag{5}$$

where ρ is a damping factor between 0 and 1. PR is:

- Input: A directed graph $G = (V,E)$.
- Output: Ranking score $p(v)$ for each vertex $v \in V$.

(1) Algorithm. Our PIE program $\mathcal{A}$ for PR declares a status variable $v.x$ for each vertex v, which is its PR score $p(v)$, initialized as 1. For each fragment F_i, its update region C_i is $F_i.O$. The aggregate function f_{aggr} is defined as sum. For each edge e associated with vertices in $F_i.O$, we maintain its global replication count, denoted as $||e||$, which is the total number of copies of e in all fragments.

At each fragment F_i, PEval does the following: (a) For each master node v that has gathered all its incoming edges, it updates $p(v)$ by applying Equation (5), and (b) for other vertices u in F_i, it computes a partial ranking score:

$$p(v) = \rho \sum_{(u,v)\in E_i} \frac{p(u)}{d^{+}_{G}(v)||(u,v)||} + (1-\rho). \tag{6}$$

To reduce the communication cost, if not all incoming edges of a vertex v are in place, we compute the partial ranking scores of v before sending messages. At the end of PEval, f_{aggr} aggregates partial scores of each border node using sum.

Upon receiving messages, IncEval first updates the score $p(u)$ of each border node u. It then iteratively updates ranking scores starting from the border nodes, and propagates the updates through outgoing edges like PEval.

The process proceeds until the sum of changes to scores $p(\cdot)$ is below a user-defined threshold ϵ. At this moment Assemble simply returns $p(v)$ for all vertices v.

(2) Transparency. We verify the partition transparency of the program as follows. Given any hybrid partition $\mathsf{HP}(G)$, we construct an edge-cut partition $\mathsf{HP}'(G)$ by making all master nodes carrying all its adjacent edges locally, and removing vertices and edges not linked to any master nodes. One can verify that (a) $\mathcal{A}(Q, \mathsf{HP}(G)){=}\mathcal{A}(Q, \mathsf{HP}'(G))$ and (b) $\mathcal{A}(Q, \mathsf{HP}'(G))$ correctly computes PR scores [50]. That is, the program correctly computes PR scores under hybrid $\mathsf{HP}(G)$.

(3) Cost Model. We have learned the following for PR:

$$h_{\mathsf{PR}} = 4.88 \times 10^{-5} d^{+}_{L}(v) + 4 \times 10^{-4},$$
$$g_{\mathsf{PR}} = 6.60 \times 10^{-4} r(v) + 1.1 \times 10^{-4},$$

where h_{PR} (in ms) is determined by d^{+}_{L} for collecting $p(\cdot)$ via incoming edges, and g_{PR} is dominated by $r(v)$ for synchronizing changes across various fragments.

Remark. Partition transparent algorithms also include the PIE programs of [17] for, *e.g.,* WCC (weakly connected component) and SSSP (single source shortest path). Moreover, many algorithms can be rewritten and made transparent, *e.g.,* MST (minimal spanning tree), Graph Simulation, Personalized PageRank and Subgraph Isomorphism.

8 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we conducted four sets of experiments to evaluate our application-driven partitioners for (1) effectiveness, (2) efficiency, (3) parallel scalability, and (4) accuracy and efficiency of cost function learning.

Experimental setting. We start with the setting.

Datasets. We used three real-life graphs: (a) liveJournal [1], a social network with 4.8 million entities and 68 million relationships; (b) Twitter [2], a social network with 42 million users and 1.5 billion links; and (c) UKWeb [3], a large Web graph with 106 million nodes and 3.7 billion edges.

We also generated synthetic graphs with size up to 500 million vertices and 6 billion edges, to test scalability.

Partitioners. We implemented the parallel version ParE2H and ParV2H of E2H and V2H (Section 5) in C++ and compared them with the following: (1) xtraPuLP [43], a state-of-the-art edge-cut partitioner; (2) Fennel [44], a streaming partitioner for edge-cut; (3) Grid [23], a hash partitioner for vertex-cut with provable bound on vertex replication; (4) NE [49], a state-of-the-art vertex-cut heuristic; and (5) Ginger [13], a hybrid partitioner that revises Fennel; we evaluated Ginger to compare improvements over Fennel.

To get a fair comparison when evaluating the effectiveness and efficiency, we equipped each edge-cut (resp. vertex-cut) partitioner above with ParE2H (resp. ParV2H) as a post-partitioning adjustment process. Denote by HxtraPuLP, HFennel, HGrid and HNE the hybrid partitioners derived in such ways, *e.g.,* HxtraPuLP first applies xtraPuLP to get an initial edge-cut $\mathsf{HP}_E(n)$, and then invokes ParE2H to extend $\mathsf{HP}_E(n)$ to a hybrid partition. We do not extend Ginger since Ginger already produces hybrid partitions.

ML learning setting. For training, we ran each algorithm on 10 graphs as described in Section 4. The number of training (resp. testing) samples for CN, TC and PR is 80,000 (resp. 20,000), which are sampled from the algorithms' running log. Regression models are constructed by Pytorch [36] and trained on a server with one NVIDIA Tesla V100 GPU.

The experiments were conducted on GRAPE [17] (see Section 6.1) deployed on 32 machines in an HPC cluster, each with 12 cores powered by Xeon 2.2GHz, 128GB RAM, and 10Gbps NIC. In the experiments, each fragment was processed by one worker running on an exclusive core. All experiments were repeated 5 times. The average is reported here.

Experimental results. We next report our findings.

Exp-1: Effectiveness. We first tested how our application-driven partitions speed up execution of graph algorithms.

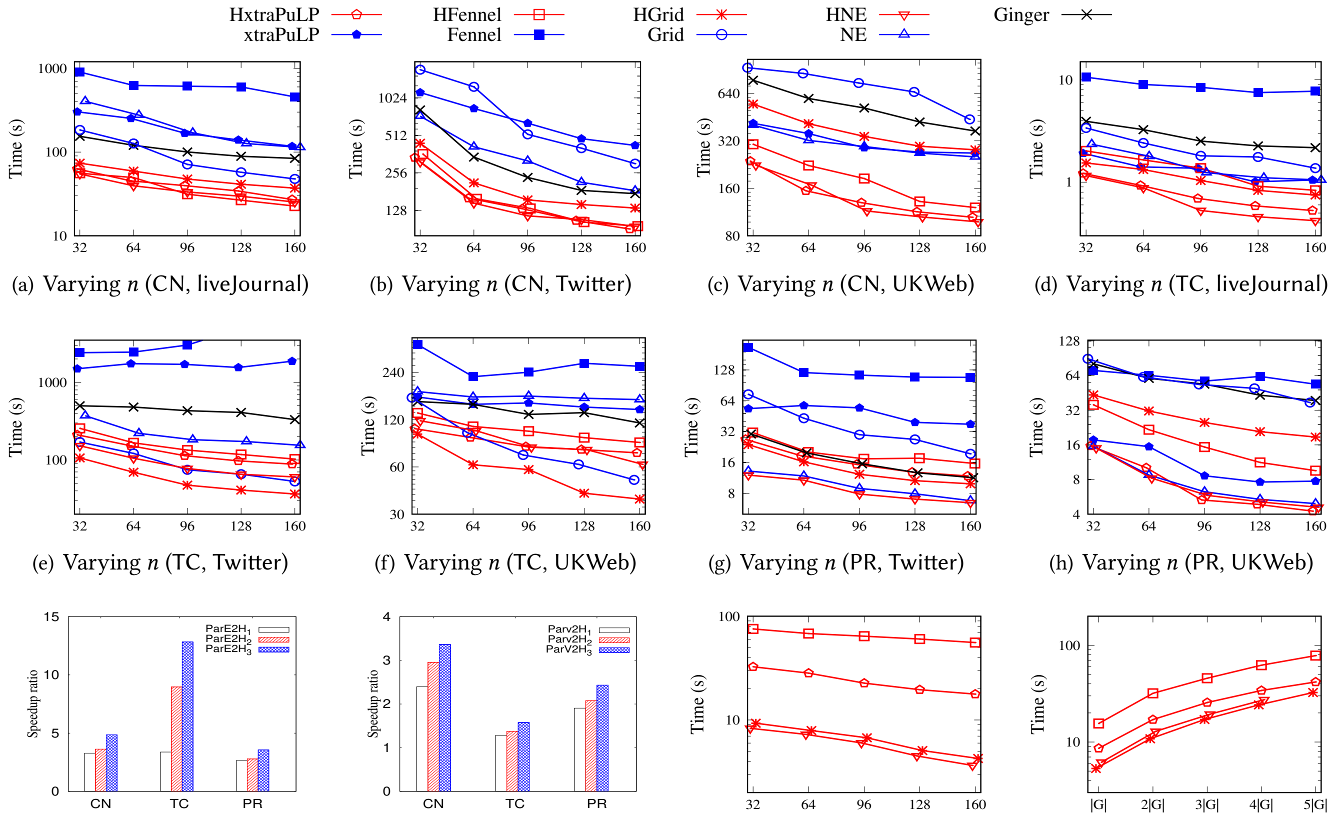
Application speedup. Varying the partition number n from 32 to 160, we tested the performance of the transparent algorithms CN, TC and PR (Section 7) under edge-cut, vertex-cut and their hybrid refinements. Note that we use different y-axes to present the costs of different algorithms.

We also tested performance gap between transparent algorithms $\mathcal{A}$ and their normal (non-transparent) version $\mathcal{B}$. We find the following. (a) When $\mathcal{A}$ and $\mathcal{B}$ run on edge-cut and vertex-cut partitions, the gap is rather small (within 5%). For instance, transparent PR is 3.9% (resp. 4.7%) slower than its normal version on vertex-cut (resp. edge-cut) partitions. (b) When $\mathcal{A}$ runs on hybrid partitions, on average it outperforms $\mathcal{B}$ by 4.7 times, up to 22.1 times, when $\mathcal{B}$ runs on edge-cut and vertex-cut partitions (note that $\mathcal{B}$ may not work on hybrid partitions). For PR, the gap is 2.7 times. In light of these, we only report the performance of the transparent algorithms in the sequel to demonstrate the impact of partitions, excluding the impact of algorithm implementations.

(1) CN. Figures 4(a) to 4(c) report the performance of CN on all three real-life datasets. Due to memory limit, we filtered common neighbors with incoming degree above a threshold θ, *i.e.,* a common neighbor w is excluded from the result if $d^+_G(w) > \theta$. This is a common practice in applications of CN, since common neighbors with lower degrees usually provide more useful information. We set $\theta = 300$ for Twitter, $\theta = 1000$ for UKWeb and $\theta = \infty$ for liveJournal. Note that θ for Twitter is smaller than for UKWeb since Twitter is much more skewed than UKWeb. The results tell us the following.

(a) By extending edge-cut of xtraPuLP and Fennel to hybrid partition, ParE2H improves the performance of CN by 3.8 (resp. 18.2) times on average, up to 22.5 times. This is because the edge-cut partitions do not fit the cost pattern of CN due to workload imbalance, while ParE2H balances workload. For example, as shown in Table 2, on Twitter with $n = 96$, the balance factor λ_{CN} for CN of xtraPuLP (resp. Fennel) is 7.2 (resp. 13.7). With ParE2H, λ_{CN} of HxtraPuLP (resp. HFennel) drops to 1.4 (resp. 1.3). CN of Fennel ran out-of-memory over Twitter and UKWeb due to large λ_{CN}, since CN stores large intermediate results during the computation.

(b) ParV2H improves CN on vertex-cut of Grid (resp. NE) by 3.5 (resp. 2.5) times on average, up to 7.4 (resp. 5.9) times by balancing the computation workload of CN, as ParE2H. Observe that the speedup ratio over Grid is greater than over NE, because CN incurs communication cost and the locality of Grid is not as good as that of NE, *i.e.,* Grid has a larger f_v than NE (see Table 2). By balancing workload and reducing communication, ParV2H improves Grid better than NE.



(a) Varying n (CN, liveJournal) (b) Varying n (CN, Twitter) (c) Varying n (CN, UKWeb) (d) Varying n (TC, liveJournal)

(e) Varying n (TC, Twitter) (f) Varying n (TC, UKWeb) (g) Varying n (PR, Twitter) (h) Varying n (PR, UKWeb)

(i) ParE2H (Twitter, n = 96) (j) ParV2H (Twitter, n = 96) (k) Varying n (efficiency, Twitter) (l) Varying G (partition time)

Figure 4: Performance Evaluation

(c) Note that Ginger is also a hybrid partitioner based on Fennel [13]. Compared with HFennel, it has smaller f_v, f_e, λ_e, but larger λ_{CN} (see Table 2). As a result, HFennel beats Ginger on CN by 2.7 times on average, up to 3.7 times.

These verify the benefit of application-driven partitioning, and the need for revising balance factor and replication ratio.

(2) TC. Figures 4(d) to 4(f) report the performance of TC on the three real-life datasets. We find the following.

(a) ParE2H improves TC by balancing workload as for CN. On Twitter with $n = 96$, it improves TC on edge-cut of xtraPuLP and Fennel by 15 and 21 times, respectively. Moreover, TC over HFennel is 3.2 times faster than over Ginger.

(b) ParV2H improves TC over vertex-cut of Grid and NE by 2.3 and 1.7 times on average, up to 2.6 and 2.2 times, respectively. Here the speedup of ParV2H is less than that of ParE2H, since (i) by cutting vertices, Grid and NE get better workload balance for TC than xtraPuLP and Fennel; and (ii) TC ships more data over vertex-cut than over edge-cut.

(3) PR. As shown in Figures 4(g) and 4(h), on average, ParE2H improves PR by 4.8 and 2.8 on Twitter and UKWeb, respectively. By h_{PR} (see Section 7), the edge size dominates the computational cost of PR. ParE2H improves PR by balancing the edge size based on h_{PR}, while xtraPuLP and Fennel suffer from edge imbalance. In contrast to ParE2H, ParV2H has smaller improvement over PR. This is because vertex-cut of Grid and NE has better edge balance (see λ_e in Table 2).

The results on liveJournal are consistent.

Impact of different phases. We further evaluated the phases of ParE2H and ParV2H for their effectiveness. Denote by ParE2H$_k$ (resp. ParV2H$_k$) ($1 \leq k \leq 3$) the partitioner with the first k phases of ParE2H (resp. ParV2H). We assessed the performance gain of the k-th phase of ParE2H by comparing ParE2H$_{k-1}$ and ParE2H$_k$; similarly for ParV2H. Figures 4(i) and 4(j) report the normalized speedup ratio over Twitter with $n = 96$ for HxtraPuLP and HGrid, respectively. The results over liveJournal and UKWeb and other hybrid partitioners are consistent (not shown). We find the following.

(1) ParE2H. (a) Phase EMigrate accounts for 67.5%, 26.3% and 74.4% of the total speedup of CN, TC and PR, respectively. This shows the effectiveness of our approach on non-transparent algorithms. (b) Phase ESplit alone improves CN and TC by 1.1 and 2.7 times respectively. For PR, its impact is smaller, since CN and TC are more sensitive to workload

Partitioner	f_v	f_e	λ_e	λ_v	λ_{CN}
xtraPuLP	11	1.7	11.1	0.1	7.2
HxtraPuLP	10.6	1.6	8.6	0.5	1.4
Fennel	13	1.8	17.2	0.1	13.7
HFennel	14.3	1.7	5.2	0.7	1.3
Grid	9.8	1	0.9	0.6	3.2
HGrid	11.1	1.3	1.2	0.5	1.3
NE	2.7	1	0.0004	8.0	3.6
HNE	3.6	1.2	0.3	10.9	1.4
Ginger	8.6	1	0.03	7.9	2.9

Table 2: Partition metrics of Twitter ($n = 96$)

imbalance. The impact of ESplit on CN over Twitter is less substantial, since we filtered large-degree vertices for CN. Without filtering, ESplit improves CN over liveJournal by 1.9 times. (c) Phase MAssign accounts for another 22.3%, 30.1% and 21.9% of the speedup of CN, TC and PR, respectively.

(2) ParV2H. (a) Phase VMigrate contributes the most to the speedup of CN, TC and PR, about 71.2%, 81.2% and 78.2% of the total speedup, respectively. (b) By merging v-cut nodes into e-cut nodes, phase VMerge contributes 16.5%, 5.8% and 7.1% of the total speedup for the algorithms, respectively. (c) Phase MAssign contributes 13.2% on average.

Exp-2: Efficiency. Varying n from 32 to 160, we evaluated the time taken by ParE2H (resp. ParV2H) in hybrid partitioners HxtraPuLP and HFennel (resp. HGrid and HNE).

(1) As shown in Fig. 4(k), for TC on Twitter ($n = 160$) ParE2H takes 17.8s and 55.7s, *i.e.,* 1.7% and 9.8% of the total time of HxtraPuLP and HFennel, respectively. On average, ParE2H takes 11.9% of the total partitioning time to extend edge-cut to a hybrid partition that fits the cost pattern of algorithms. The price is small compared to the speedup by ParE2H (by 7.3 times on average, up to 22.5 times; Exp-1). The results are consistent for the other algorithms and graphs.

(2) The results of ParV2H are similar. On Twitter with $n = 96$, ParV2H takes 5.9s (resp. 6.6s) to extend a vertex-cut of NE (resp. Grid) to a hybrid partition. On all 3 datesets, ParV2H takes 0.1% and 22.5% of the total partitioning time in HNE and HGrid, respectively, while improving the performance by 2.2 times on average, up to 7.4 times (see Exp-1).

Exp-3: Scalability. Fixing $n = 96$, we varied the size of synthetic graphs $|G| = (|V|, |E|)$ from $(100M, 1.2B)$ to $(500M, 6B)$ to test the scalability of ParE2H and ParV2H. As shown in Fig. 4(l) for CN, (1) both ParE2H and ParV2H scale well. As G grows, ParE2H (resp. ParV2H) takes from 12.2s to 59.7s (resp. from 5.7s to 32.5s). Observe that in Fig 4(l), the point of ParV2H in HNE is missing for $5|G|$, since NE ran out-of-memory. (2) The balance factor of an input partition has impact on the runtime of ParE2H and ParV2H, *e.g.,* HFennel takes ParE2H the longest in all cases since Fennel has the largest λ_{CN}, and more edges are moved to balance workload. (3) The results are consistent for TC and PR.

	$h_{\mathcal{A}}$		$g_{\mathcal{A}}$	
	MSRE	training time(s)	MSRE	training time(s)
CN	0.023	46.2	0.028	48.6
TC	0.11	48.3	0.034	47.4
PR	0.017	43.6	0.011	46.9

Table 3: Accuracy and training time of cost models

Exp-4: Learning accuracy and efficiency. Table 3 reports the prediction accuracy and training time of each cost model. We adopt MSRE as the accuracy metric. The smaller MSRE of a model is, the more accurate the model is. As shown in Table 3, the MSRE of regression model for CN and PR is small, which shows that the metrics in $\mathcal{X}$ are adequate for their cost estimation. However, the accuracy of h_{TC} is relatively poorer since only neighbors with smaller degrees are checked (Section 7.2). This optimization deteriorates h_{TC} as node degrees are not informative enough for cost prediction.

Summary. We find the following. (1) ParE2H (resp. ParV2H) speeds up CN, TC and PR by 11.0, 7.2 and 3.8 times (resp. 3.0, 1.9 and 1.7 times) on average, up to 22.5, 21.2 and 6.9 times when varying n from 32 to 160. These verify the effectiveness of application-driven partitioners. (2) ParE2H and ParV2H are efficient. In the same setting as (1), on average ParE2H (resp. ParV2H) takes 11.9% (resp. 11.3%) of the total partitioning time. (3) The partitioners scale well. On graphs with $500M$ vertices and $6B$ edges, ParE2H (resp. ParV2H) takes 59.7s (resp. 32.5s) to extend edge-cut (resp. vertex-cut) to a hybrid partition, with 96 workers. (4) The learned functions accurately estimate computational and communication costs. The MSRE is below 0.11 for all the algorithms tested. Moreover, the training time is small, *e.g.,* at most 48.6s for g_{CN}.

9 CONCLUSION

We have proposed an application-driven partitioning strategy. For a given graph algorithm $\mathcal{A}$, we have shown how to learn its cost model, and developed partitioners that refine an edge-cut or vertex-cut partition to fit in with the cost patterns of $\mathcal{A}$ and speed up parallel execution of $\mathcal{A}$. We have provided conditions for graph algorithms to be partition transparent, and developed transparent algorithms for several problems. Our experimental study has verified that application-driven partitioning is effective and efficient.

One topic for future work is to extend our cost model to handle multiple applications run on the same graph. Another topic is to find other cost metrics for various algorithms.

Acknowledgements. Fan, Jin, Liu and Xu are supported in part by ERC 652976, Royal Society Wolfson Research Merit Award WRM/R1/180014, EPSRC EP/M025268/1, Shenzhen Institute of Computing Sciences, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Jin is also supported by the China Scholarship Council. Liu is also supported in part by EPSRC EP/L01503X/1, EPSRC CDT in Pervasive Parallelism at the University of Edinburgh, School of Informatics. Lu is supported in part by NSFC 61602023.

REFERENCES

[1] Livejournal. *http://snap.stanford.edu/data/soc-LiveJournal1.html.*

[2] Twitter. *http://twitter.com/.*

[3] UKWeb.*http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05*, 2006.

[4] A. Adadi and M. Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018.

[5] K. Andreev and H. Racke. Balanced graph partitioning.*TCS*, 39(6), 2006.

[6] D. Avdiukhin, S. Pupyrev, and G. Yaroslavtsev. Multi-dimensional balanced graph partitioning via projected gradient descent. *PVLDB*, 12(8):906–919, 2019.

[7] C.-E. Bichot and P. Siarry. *Graph partitioning*. John Wiley & Sons, 2013.

[8] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[9] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *SIGKDD*, pages 1456–1465, 2014.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *WWW*, pages 107–117, 1998.

[11] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering - Selected Results and Surveys*, pages 117–158. 2016.

[12] G. Chandrashekar and F. Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.

[13] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[14] W. Cukierski, B. Hamner, and B. Yang. Graph-based features for supervised link prediction. In *INCC*, pages 1237–1244. IEEE, 2011.

[15] D. Dai, W. Zhang, and Y. Chen. IOGP: An incremental online graph partitioning algorithm for distributed graph databases. In *HPDC*, pages 219–230, 2017.

[16] W. Fan, P. Lu, X. Luo, J. Xu, Q. Yin, W. Yu, and R. Xu. Adaptive asynchronous parallelization of graph algorithms. In *SIGMOD*, pages 1141–1156, 2018.

[17] W. Fan, W. Yu, J. Xu, J. Zhou, X. Luo, Q. Yin, P. Lu, Y. Cao, and R. Xu. Parallelizing sequential graph computations. *TODS*, 43(4):18:1–18:39, 2018.

[18] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[20] J. Huang and D. Abadi. LEOPARD: Lightweight edge-oriented partitioning and replication for dynamic graphs. *PVLDB*, 9(7), 2016.

[21] L. Huang, J. Jia, B. Yu, B. gon Chun, P. Maniatis, and M. Naik. Predicting execution time of computer programs using sparse polynomial regression. In *NIPS*, 2010.

[22] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

[23] N. Jain, G. Liao, and T. L. Willke. Graphbuilder: Scalable graph etl framework. *Graph Data Management Experiences and Systems*, 2013.

[24] G. Karypis. Metis and parmetis. In *Encyclopedia of Parallel Computing*, pages 1117–1124. 2011.

[25] G. Karypis and V. Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[26] G. Karypis and V. Kumar. Metis: A software package for partitioning unstructured graphs. *Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version*, 4, 1998.

[27] G. Karypis and V. Kumar. Multilevelk-way Partitioning Scheme for Irregular Graphs. *JPDC*, 48(1):96–129, 1998.

[28] M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *DKE*, 72:285–303, 2012.

[29] R. Krauthgamer, J. Naor, and R. Schwartz. Partitioning graphs into balanced components. In *SODA*, 2009.

[30] D. Li, Y. Zhang, J. Wang, and K. Tan. TopoX: Topology refactorization for efficient graph partitioning and processing. *PVLDB*, 12(8):891–905, 2019.

[31] D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. *CIKM*, 2003.

[32] D. W. Margo and M. I. Seltzer. A scalable distributed graph partitioner. *PVLDB*, 8(12):1478–1489, 2015.

[33] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD*, pages 145–156, 2012.

[34] M. E. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99(suppl 1):2566–2572, 2002.

[35] H. Park and L. Stefanski. Relative-error prediction. *Statistics & probability letters*, 40(3):227–236, 1998.

[36] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[38] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. HDRF: Stream-based partitioning for power-law graphs. In *CIKM*, 2015.

[39] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIMAX*, 11(3):430–452, 1990.

[40] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.

[41] M. T. Ribeiro, S. Singh, and C. Guestrin. "Why should I trust you?" Explaining the predictions of any classifier. In *SIGKDD*, pages 1135–1144, 2016.

[42] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[43] G. M. Slota, S. Rajamanickam, and K. Madduri. PuLP/XtraPuLP: Partitioning tools for extreme-scale graphs. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2017.

[44] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL: Streaming graph partitioning for massive scale graphs. In *WSDM*, pages 333–342, 2014.

[45] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[46] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440, 1998.

[47] Wikipedia. Stone–Weierstrass Theorem. *https://en.wikipedia.org/wiki/Stone-Weierstrass_theorem.*

[48] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD*, page 517, 2012.

[49] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li. Graph edge partitioning via neighborhood heuristic. In *KDD*, 2017.

[50] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *TPDS*, 25(8):2091–2100, 2013.

[51] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.