Noname manuscript No.
(will be inserted by the editor)

Application Driven Graph Partitioning

Wenfei Fan · Ruiqi Xu · Qiang Yin · Wenyuan Yu · Jingren Zhou

Abstract Graph partitioning is crucial to parallel computations on large graphs. The choice of partitioning strategies has strong impact on the performance of graph algorithms. For an algorithm of our interest, what partitioning strategy fits it the best and improves its parallel execution? Is it possible to provide a uniform partition to a batch of algorithms that run on the same graph simultaneously, and speed up each and every of them?

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 extend the cost-driven strategy to support multiple algorithms at the same time and reduce the parallel cost of each of them. Using real-life and synthetic graphs, we experimentally verify that our partitioning strategy improves the performance of a variety of graph algorithms, up to 22.5×.

Keywords graph partition; machine learning

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 computation workload and their communication is minimized.

W. Fan
University of Edinburgh, UK & Shenzhen Institute of Computing Sciences, China & BDBC, Beihang University, China
E-mail: wenfei@inf.ed.ac.uk

R. Xu
University of Edinburgh, UK
E-mail: ruiqi.xu@ed.ac.uk

Q. Yin (corresponding author)
Shanghai Jiao Tong University, China
E-mail: q.yin@sjtu.edu.cn

W. Yu · J. Zhou
Alibaba Group, China
E-mail: wenyuan.ywy@alibaba-inc.com
E-mail: jingren.zhou@alibaba-inc.com

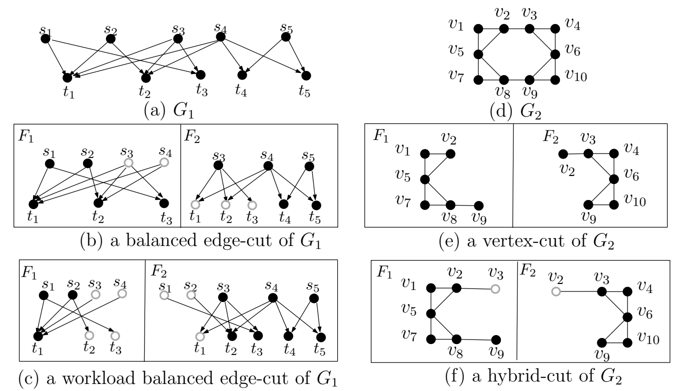

(a) G_1 (d) G_2 (b) a balanced edge-cut of G_1 (e) a vertex-cut of G_2 (c) a workload balanced edge-cut of G_1 (f) a hybrid-cut of G_2

Fig. 1: CN and TC

A number of partitioning algorithms (*a.k.a.* partitioners) are already developed. These algorithms are often either *edge-cut* [12,32,29], which evenly partitions vertices and cuts edges, or *vertex-cut* [24,33,53], which evenly partitions edges by replicating vertices. There has also been recent work on *hybrid* partitioners, which cut both edges and vertices [16,19,54,35].

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 algorithm CN (Common Neighbor) [36] 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 [36,18]. To simplify the discussion, we consider outgoing common neighbors, where a vertex u is a outgoing common neighbor of v_1 and v_2 if there exist edges from v_1 and v_2 to u. 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. Indeed, CN counts common neighbors by aggregating the contribution of its incoming neighbors to the common neighbor count. More specifically, for each vertex v, it collects its incoming neighbors and increases the count of each pair of such neighbors. Thus the computation cost on a fragment F_i is determined by $\sum_{v \in F_i} \frac{1}{2} d^+(v)(d^+(v)-1)$, where $d^+(v)$ is node v's in-degree. As a result, the workload on F_1 and F_2 is 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 graph 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 [50], cycle detection [27] and transitivity [40]. 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).

(a) Observe that 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 vertex v_9.

(b) Replication helps us reduce the communication cost. Consider the partition of G_2 in Fig. 1(f). As opposed to vertex-cut in Fig. 1(e), it replicates a vertex v_3 and an edge (v_2, v_3) at F_1. When running 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 involving v_9, one can further replicate edge (v_8, v_9) and vertex v_8 at F_2. □

It is more intriguing when mixed workloads are considered. In practice, multiple applications (graph algorithms) often need to run on the same graph at the same time. For example, one needs to run algorithms PageRank (PR) [13], CN and TC together to find the most influential vertices in G, discover community structures and predict missing links in G. However, a cost-balanced partition for one algorithm might become skewed for another, as illustrated by the example below.

Example 2 Consider running PageRank (PR) and CN on the same graph partition at the same time.

(a) As remarked in Example 1, the computation cost on F_i for CN is determined by $\sum_{v \in F_i} \frac{1}{2} d^+(v)(d^+(v) - 1)$, while the cost for PR is $\sum_{v \in F_i} d^+(v)$. Thus the workload of the partition in Fig. 1(a) is skewed for CN but is balanced for PR, while the one in Fig. 1(b) is exactly the contrary, *i.e.,* balanced for CN but skewed for PR.

(b) A brute-force solution is to store both partitions, *i.e.,* the partition in Fig. 1(a) to run PR and the one in Fig. 1(b) to run CN. However, to support k different algorithms on a graph G, we need to compute and store k partitions of the same graph G. This not only increases the time and space cost, but also introduces the coherence problem when G is updated. □

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 reduce parallel cost 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, is it possible to provide a uniform partition that supports a batch of algorithms on the same graph and reduces the cost of each algorithm in the batch?

Contributions & organization. This paper aims to answer these questions. We propose an application-driven graph partitioning strategy both for a single algorithm and for a mixed workload of algorithms.

(1) Application-driven partitioning (Section 3). We propose a *hybrid partitioning strategy* to find partitions tailored for a given graph algorithm $\mathcal{A}$. We introduce a cost model, consisting of two functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ to characterize the computational and communication patterns of algorithm $\mathcal{A}$, respectively. 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 ap-

proximate the cost model as polynomial regression following [51], which has proven effective in the real life [26]. We train the model with the stochastic gradient descent algorithm. The learned cost model can be applied to different graphs on which algorithm $\mathcal{A}$ runs.

(3) Application-driven partitioners (Section 5). We develop parallel partitioners ParE2H and ParV2H which, 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) Composite cost-driven partitioners (Section 6). To handle mixed workloads, we extend our parallel hybrid partitioners to composite ones ParME2H and ParMV2H. Given a batch of k algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_k$ and a graph G, they generate a *composite partition* of G that uniformly represents k hybrid partitions, such that each partition is driven by the cost model of one of the k algorithms and speeds up its parallel execution. It should be remarked that ParE2H (resp. ParV2H) is a special case of ParME2H (resp. ParMV2H) when $k = 1$.

(5) Experimental study (Section 7). Using real-life and synthetic graphs, we empirically verify the effectiveness, scalability and efficiency of our application-driven partitioners. We find the following. (a) The partitioners are effective. Over real-life graphs, ParE2H and ParV2H improve the performance of CN, TC, PR, WCC (connected components [9]) and SSSP (single source shortest path [21]) by 7.5, 4.7, 2.6, 2.2, and 1.3 times on average, respectively, up to 22.5 times. The improvement by the composite ParME2H and ParMV2H is comparable, incurring at most 8.2% more time of these algorithms despite their compact representations. (b) They are efficient, taking 11.5% and 11.1% 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) Composite ParME2H (resp. ParMV2H) saves space up to 55% (resp. 67%) for CN, TC, WCC, PR and SSSP and speeds up partitioning by at least 2.0 (resp. 1.2) times. (e) At a small training cost, the learned cost models are accurate, with MSRE (mean squared relative error) ≤ 0.11.

Related work. This work extends its conference version [20] as follows. (1) We give a proof for the NP-completeness of ADP problem (Theorem 1). (2) We present algorithm V2H and its implementation details (Section 5.2), which are not included in [20]. (3) We introduce a systematic solution to partitioning a graph for a batch of algorithms, by extending ParE2H and ParV2H into composite ParME2H and ParMV2H, respectively (Section 6). (4) We implement our parallel composite partitioners ParME2H and ParMV2H, and conduct new experiments to verify their effectiveness, efficiency and scalability (Exp-2, Exp-4 and Exp-5, Section 7).

The other related work is categorized as follows.

Various algorithms have been developed for edge-cut and vertex-cut partitions (see [14,10] 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 [7,34] compute balanced partitions and cut minimum edges. METIS [30,31] and its parallel version ParMETIS [29] adopt a multi-level heuristic scheme and are widely used in practice. Other popular heuristics include parallel partitioners such as XtraPuLP [46] and stream partitioners FENNEL [47]. Vertex-cut partitioners include spectral algorithm of [44] and heuristics Grid [28], SHEEP [38], NE [53] and HDRF [43].

Edge-cut promotes locality: for each vertex v in a 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 limitations, there has been work on hybrid partitioners. PowerLyra [16] and IOGP [19] combine edge-cut and vertex-cut by cutting only high-degree vertices, controlled by a user-defined threshold. TopoX [35] not only splits high-degree vertices, but also merges neighboring low-degree vertices into super nodes to prevent splitting such vertices. Gemini [54] and MD-BGP [8] balance hybrid workload by combining vertex and edge loads based on a balancing metric.

There have also been efforts to speed up distributed graph computations by replicating various parts of a graph across partitions [25,52,39], such that read operations can be done locally without communication.

This work differs from the prior methods as follows.

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

(2) 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 exist no need to develop another partitioner starting from scratch. In addition, the partitioners strike a balance between replication and computation speedup.

(3) We develop the first ML models that train cost models for given graph algorithms (Section 4), as opposed to prior partitioners that adopt one-size-fits-all static metrics [7,34,46], follow intuitions [35,54] or manually pick partitioning parameters [16,19,8]. In contrast to prior partitioners, the learned cost model is an required input for our application-driven partitioners. With the cost model of a given algorithm $\mathcal{A}$, our partitioners aim to produce partitions that fit the cost patterns of $\mathcal{A}$.

(4) We propose a systematic solution to partitioning a graph for mixed workloads on the same graph at the same time (Section 6). We introduce a compact representation of multiple partitions, and extend our cost-driven partitioners to support a batch of algorithms. The composition partitioners reduce not only the time and space costs of the partitioning process, but also the execution cost of each graph algorithm in the batch.

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, ..., 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 fragment F_i, *i.e.,* $E^v \neq E_i^v$ for all $i \in [1, n]$. We refer to each copy of such v in the partition $\mathsf{HP}(n)$ as 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).

(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. The copies of v_9 in F_1 and F_2 are v-cut nodes.

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

(3) Observe that $F_1.O = F_2.O = \{v_2, v_3, v_9\}$. Thus $\mathcal{F}.O = \{v_2, v_3, v_9\}$. If a master node mapping designates v_2 and v_3 of F_1 and v_9 of F_2 as masters, then the copies v_2 and v_3 of F_2 and v_9 of F_1 are mirrors. □

Special cases. Edge-cut partitions [7,32] and vertex-cut partitions [24] are special cases of hybrid partitions.

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

(2) A hybrid partition $\mathsf{HP}(n)$ is also *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 in multiple fragments.

Example 4 Consider the 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) The 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. Vertices v_2 and v_9 are replicated.

(3) The 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). It is a hybrid partition. □

Quality. The *partition quality* of hybrid partitions is usually 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. Conventional vertex-cut partitioning aims to minimize f_v, and edge-cut partitioning

Table 1: Notations

G, V, E	a graph, its vertex set and edge set
F_i (resp. F_i^j)	the i-the fragment of G (for $\mathcal{A}_j$)
C_i, $\hat{F}_i^j$	the core and residual part of F_i^j for $\mathcal{A}_j$
V_i (resp. E_i)	the vertex set (resp. edge set) of F_i
E^v (resp. E_i^v)	edges incident to v in G (resp. F_i)
$F_i.O$ (resp. $\mathcal{F}.O$)	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)
$C_\mathcal{A}^h(F_i)$	computational cost of F_i
$C_\mathcal{A}^g(F_i)$	communication cost of F_i
$C_\mathcal{A}(F_i)$	the cost of $\mathcal{A}$ on F_i
f_v (resp. f_e)	vertex (resp. edge) replication ratio
λ_v (resp. λ_e)	balance factor *w.r.t.* vertices (resp. edges)
$\lambda_\mathcal{A}$	balance factor *w.r.t.* the cost functions of $\mathcal{A}$

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), respectively.

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. The vertex balance factor λ_v of hybrid partition $\mathsf{HP}(n)$ is defined as $\lambda_v = \min\{\lambda \mid |V_i| \leq (1+\lambda)\sum_{j=1}^n |V_j|/n \ (\forall j \in [1, n])\}$.

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 edge balance factor λ_e of hybrid partition $\mathsf{HP}(n)$ is defined as $\lambda_e = \min\{\lambda \mid |E_i| \leq (1+\lambda)\sum_{j=1}^n |E_j|/n \ (\forall j \in [1, n])\}$.

Example 5 For the partition shown in Fig. 1(b), we have that $f_v = 1$, $f_e = 17/13$, $\lambda_v = 1$ and $\lambda_e = 18/17$. For the one in Fig. 1(c), $f_v = 1$, $f_e = 17/13$, $\lambda_v = 7/5$ and $\lambda_e = 22/17$. The vertex replication factors are both 1, since both partitions are edge-cut. Similarly, for the partition in Fig. 1(d), $f_v = 6/5$, $f_e = 1$, $\lambda_v = 1$ and $\lambda_e = 1$. Here $f_e = 1$, as the partition of Fig. 1(d) is vertex-cut. For the hybrid partition in Fig. 1(e), $f_v = 6/5$, $f_e = 15/14$, $\lambda_v = 1$ and $\lambda_e = 8/7$. □

We summarize the notations in Table 1.

3 Application Driven Graph Partitioning

The primary goal of graph partitioners is to speed up parallel computations of applications 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 target a single algorithm given by users in this section, and will handle multiple algorithms in Section 6. Below 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 and communication costs 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* $g_\mathcal{A}$, elaborated as follows.

Cost model. Let $\mathcal{X} = \{x_1, \ldots, x_k\}$ be a set of *metric variables*, where each variable $x_i \in \mathcal{X}$ 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 $C_\mathcal{A}^h(F_i)$ and $C_\mathcal{A}^g(F_i)$ the computational cost and communication cost of algorithm $\mathcal{A}$ on fragment F_i, respectively. Then the cost of algorithm $\mathcal{A}$ on fragment F_i is estimated as

$$C_\mathcal{A}(F_i) = C_\mathcal{A}^h(F_i) + C_\mathcal{A}^g(F_i). \quad (1)$$

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

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

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

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

Communication cost $C_\mathcal{A}^g(F_i)$. Unlike $C_\mathcal{A}^h(F_i)$, $C_\mathcal{A}^g(F_i)$ is incurred by vertices replicated in $\mathsf{HP}(n)$. We measure the communication cost incurred by master nodes:

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

Intuitively, for a vertex $v \in F_i.O$, its communication and synchronization often take place at its master, which is responsible for receiving updates from its mirrors and sending the aggregation back to its mirrors [22, 24].

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

- $d_L^+(v) = |\{u \mid (u, v) \in E_i\}|$, *i.e.,* v's in-degree in F_i;
- $d_L^-(v) = |\{u \mid (v, u) \in E_i\}|$, *i.e.,* v's out-degree in F_i;

- $d^+_G(v) = |\{u \mid (u,v)\in E\}|$, *i.e.,* v's in-degree in G;
- $d^-_G(v) = |\{u \mid (v,u)\in E\}|$, *i.e.,* v's out-degree 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 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; and $d^+_G(v)$, $d^-_G(v)$ and D may affect the size of messages synchronized between the master of v and its mirrors. In particular, degree-based variables such as d^+_L and d^-_L can be used to assess the impact of super nodes on the computational cost of the applications on the fragments.

We will use the following metric variable set:

$$\mathcal{X} = \{d^+_L, d^-_L, d^+_G, d^-_G, r, D\}. \tag{4}$$

Remark. We highlight the ideas of (1) how we select the variables in the metric variable set $\mathcal{X}$ as defined in Eq. (4) and (2) how to extend the set $\mathcal{X}$ when necessary.

(1) In general, a metric variable is included in $\mathcal{X}$ if it can be used to determine the input size for the computation or communication incurred by a vertex. Take an algorithm $\mathcal{A}$ under a vertex-centric graph computation model, *e.g.,* Pregel [37], as an example. To estimate the computational cost, we include those metric variables in $\mathcal{X}$ that define the input size of the Compute function of $\mathcal{A}$. Typical examples are d^+_L and d^-_L since each vertex in $\mathcal{A}$ receives messages from its neighbors. For the communication cost, we include r in $\mathcal{X}$ to determine the communication size of a master vertex, *i.e.,* the number of mirrors that require to synchronize with the master. The variables d^+_G, d^-_G and D are included in $\mathcal{X}$ mostly for graph-centric algorithms. This is because, unlike vertex-centric algorithms, there are no explicit vertex functions like Compute in graph-centric algorithms. Combining with d^+_L, d^-_L and r, the variables d^+_G, d^-_G and D are are needed to estimate the amortized input size of vertex computation and communication.

(2) For a specific algorithm $\mathcal{A}$, the set $\mathcal{X}$ can serve as a start point, since it includes the metric variables that affect the cost of most graph algorithms. One can extend $\mathcal{X}$ or pick a subset of $\mathcal{X}$, depending on the computation of $\mathcal{A}$. Suppose that the vertex data size is not uniform in algorithm $\mathcal{A}$, *e.g.,* each vertex carries a mutable array Ary as its intermediate data, and the computation of $\mathcal{A}$ is required to scan and update Ary in the process. In this case, the vertex data size |Ary| plays a role in determining the input size in the vertex computation and communication, and hence should also be included in $\mathcal{X}$. We do not include such variable in Eq. (4) because $\mathcal{X}$ suffices for the algorithms considered in this paper.

Example 6 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 (*i.e.,* when $d^+_L(v) = d^+_G(v)$), 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 $D d^-_G(v)$, 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 $d_L(v)$ and $d_G(v)$ are the degrees of v in F_i and G, respectively; and $I(v)$ 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. 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 the neighbors of v is proportional to $d_G(v)\cdot r(v)$. Based on cost function g_{TC}, to reduce the communication cost of TC, we may make more vertices e-cut as shown in Fig. 1(f). □

Balance factor revised. Incorporating parallel computation cost, we revise the conventional balance factor as follows. 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 using cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ (see Equation (1)).

Example 7 Continuing with Example 6, 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 factor for balancing the workload of CN, rather than merely the number of edges and vertices. □

3.2 Application-driven Graph Partitioning Strategy

We now present an *application-driven partitioning strategy*. Given a graph algorithm $\mathcal{A}$ of our interest, the strategy works in two steps as follows:

(1) it first learns the cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of $\mathcal{A}$;
(2) when $\mathcal{A}$ is applied to any graph G, given a natural number n, it computes a partition $\mathsf{HP}(n)$ of G to minimize the parallel cost $\mathsf{max}_{i\in[1,n]}C_{\mathcal{A}}(F_i)$ of $\mathcal{A}$.

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

The strategy is carried out by the following:

(1) *Polynomial regression models*: given an algorithm $\mathcal{A}$, learn the cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of $\mathcal{A}$ (Section 4).

(2) *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. To settle the complexity of application-driven partitioning, we study its decision problem, denoted by ADP and 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 algorithm $\mathcal{A}$.
- Question: Does there exist a $\mathsf{HP}(n)$ of G such that 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 An algorithm for ADP works as follows: (1) guess a $\mathsf{HP}(n)$ of G; and (2) check whether $\max_{i\in[1,n]}C_{\mathcal{A}}(F_i) \leq B$; if so, return true. The algorithm is in NP since step(2) is can be done in polynomial time (PTIME); so is ADP.

We verify the lower bound by reduction from the set partition problem, which is NP-complete (cf. [23]); that problem is to decide, given a set S of positive integers, whether S can be partitioned into two disjoint sets A_1 and A_2 of equal sum, *i.e.,* $\sum_{a_i\in A_1} a_i = \sum_{b_j\in A_2} b_j$.

Given a set of positive integers $S = \{s_1, \ldots, s_m\}$, we construct a graph G, a number n, a cost budget B and two cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ to ensure the following property: S can be partitioned into two disjoint sets of equal sum if and only if $\max_{i\in[1,n]}C_{\mathcal{A}}(F_i) \leq B$. More specifically, the construction is given as follows.

(1) Denote by K_ℓ the clique that consists of ℓ vertices. The graph G is defined as the collection of $K_{s_1}, \ldots, K_{s_m}$.

(2) $n = 2$ and $B = (\sum_{i=1}^{m} s_i)/2$. Note that we can assume *w.l.o.g.* that $\sum_{i=1}^{m} s_i$ is even; B is also an integer.

(3) The cost functions is defined as follows: (a) $h_{\mathcal{A}}(v) = 1$ and (b) $g_{\mathcal{A}}(v) = r(v)-1$, where $r(v)$ is the replication number of vertex v. Intuitively, these cost functions can be used to characterize the cost of a simple parallel algorithm $\mathcal{A}$ that counts the number of vertices of a given graph. That is, $\mathcal{A}$ first counts the vertices in each fragment, where each vertex incurs a unit computational cost; for each vertex v that is replicated, it then computes $r(v)-1$, which incurs communication cost $r(v)-1$ for synchronization and aggregation.

We next verify the correctness of our reduction.

($\Rightarrow$) Suppose that S can be partitioned into two disjoint sets A_1 and A_2 with equal sum. Then we can partition G by putting the cliques corresponding to A_1 into one fragment and other cliques into the other. Observe that the communication cost in terms of $g_{\mathcal{A}}$ is 0, since there is no replication in the partition. Thus $\max_{i\in[1,2]}C_{\mathcal{A}}(F_i) = \max\{\sum_{a_i\in A_1} a_i, \sum_{b_j\in A_2} b_j\} = B$.

($\Leftarrow$) Suppose that the graph G can be partitioned into two fragments so that $\max_{i\in[1,2]}C_{\mathcal{A}}(F_i) \leq B$. Since a vertex incurs a unit cost in terms of $h_{\mathcal{A}}$, we have that $\max_{i\in[1,2]}C_{\mathcal{A}}(F_i) = B$ and there is no vertex replication in the partition. By construction, no clique is divided. From the partition of G we derive a disjoint partition $\{A_1, A_2\}$ of S so that A_1 and A_2 have an equal sum. □

4 Learning Cost Functions

In this section, we show how to learn computational and communication cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ for a given graph 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. Below we focus on learning $h_{\mathcal{A}}$, which differs from $g_{\mathcal{A}}$ only in training data.

Cost function as polynomial regression. Given the set of 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. For instance,

if $\mathcal{X}(v) = \{d_L^+(v), d_L^-(v)\}$ and $p = 2$, then $h_\mathcal{A}(\mathcal{X}(v)) = \omega_1 d_L^+(v)^2 + \omega_2 d_L^+(v) d_L^-(v) + \omega_3 d_L^-(v)^2 + \omega_4$, where $\omega_1, \ldots, \omega_4$ are determined by the learning algorithm.

The learning algorithm employs training samples, each denoted as $[\mathcal{X}(v_k), t_k]$, which is extracted from the running log of algorithm $\mathcal{A}$ 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) [41] 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 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 a penalty function to avoid over-fitting [11].

The reason for implementing $h_\mathcal{A}$ as a polynomial function is twofold. First, in theory polynomials can closely approximate a continuous function defined on a closed interval [51]. Indeed, polynomial regression has proven effective in predicting computational cost in the real world [26]. Second, polynomial is explainable compared with other black-box ML models.

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 [11]. When collecting $\mathcal{D}_{h_\mathcal{A}}$, we only pick vertices that are used in computation. For example, we only record the metric variables $\mathcal{X}$ and the computation time of vertices t_1, t_2, t_3, t_4 and t_5 in Fig. 1(b) to collect training data for h_{CN}, since only vertices with incoming edges are involved in the computation. To get $\mathcal{D}_{g_\mathcal{A}}$ for $g_\mathcal{A}$, we only collect the communication cost of master nodes on fragment borders.

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

Example 8 We have seen h_{CN} and g_{TC} in Example 6. We next illustrate how these cost functions are learned for CN and TC; 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. With 80% (resp. 20%) of a total 100,000 samples for training (resp. testing) and the highest order p set as 2, the SGD algorithm learned $h_{\mathsf{CN}} = 9.23 \times 10^{-5} d_L^+(v) d_G^+(v) + 1.04 \times 10^{-6} d_L^+(v) + 1.02 \times 10^{-6}$, 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. Measured by this h_{CN}, the computational cost of F_1 and F_2 in Fig. 1(c) is 1.48×10^{-3}ms and 1.95×10^{-3}ms, respectively, as opposed to 2.69×10^{-3}ms and 7.45×10^{-4}ms in Fig. 1(b).

(2) TC. We learned g_{TC}=$8.42 \times 10^{-5} d_G(v) r(v) I(v)$ with 80,000 training 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. Based on g_{TC}, the communication cost of F_1 and F_2 in Fig. 1(f) is 0 and 2.5×10^{-4}ms, respectively, as opposed to 2.5×10^{-4}ms and 2.5×10^{-4}ms in Fig. 1(e). Here the master copies of v_2, v_3 and v_9 are in F_1, F_2 and F_2, respectively. □

Training cost reduction. Both the accuracy of the cost functions learned and the training cost depend on the choice of metric variables $\mathcal{X}$. We select metric variables that are influential on cost estimation, by employing feature selection methods [15] and domain knowledge. For example, since $d_L^+(v)$ and $d_G^+(v)$ of a vertex v are adequate to estimate the computational cost of CN, other metric variables in $\mathcal{X}$ are not included in h_{CN}. Moreover, we may specify that $h_{\mathsf{CN}}(v) = \omega_1 d_L^+(v) d_G^+(v) + \omega_2 d_L^+(v) + \omega_3$. This yields merely three weight parameters ω to learn, and reduces the learning cost.

5 Application Driven 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 the cost functions $h_\mathcal{A}$ and $g_\mathcal{A}$.

In this section we 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 HP(n) to fit the cost patterns of $\mathcal{A}$.

In a nutshell, both partitioners have two stages. Guided by cost function $h_\mathcal{A}$, the first stage balances computational cost and reduces parallel cost by redistributing vertices and edges among fragments. The second stage is guided by function $g_\mathcal{A}$. It reduces communication cost by adjusting the master node mapping; it does not increase the computational cost of stage 1.

In the reset of this section, 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 partitioners E2H and V2H in Section 5.3.

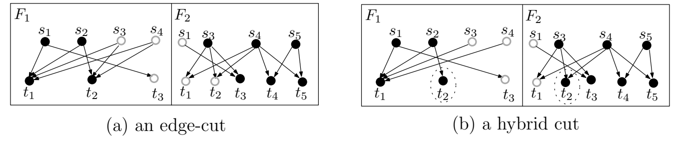
5.1 From Edge Cut to Hybrid Cut

Edge-cut promotes locality, *i.e.,* each vertex in an edge-cut partition 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 *super nodes* are adjacent to a large fraction of edges. Observe that several degree-based variables, *e.g.,* d_L^+ and d_L^-, are introduced in the set $\mathcal{X}$ for cost function learning and cost estimation (see Eq.(4)). In practice, the workload of a fragment can be dominated by the costs of its super nodes due to their large degrees. 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 hybrid partitioner E2H. Given an edge-cut $\mathsf{HP}_E(n)$ and two cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of algorithm $\mathcal{A}$, E2H extends $\mathsf{HP}_E(n)$ to a hybrid partition $\mathsf{HP}(n)$ to reduce the parallel cost $\max_{i\in[1,n]}C_{\mathcal{A}}(F_i)$ of $\mathcal{A}$ in two stages. Note that $C_{\mathcal{A}}(F_i)= C^h_{\mathcal{A}}(F_i)+C^g_{\mathcal{A}}(F_i)$. Guided by function $h_{\mathcal{A}}$, the first stage of partitioner E2H balances computational workload to reduce the cost $\max_{i\in[1,n]}C^h_{\mathcal{A}}(F_i)$ of $\mathcal{A}$ (and thus $\max_{i\in[1,n]}C_{\mathcal{A}}(F_i)$). Guided by $g_{\mathcal{A}}$, its second stage reduces $\max_{i\in[1,n]}C^g_{\mathcal{A}}(F_i)$ by redistributing communication cost.

Balancing computational cost. This stage consists of two phases, namely, EMigrate and ESplit. To balance the computational cost, both phases migrate vertices and edges from overloaded fragment to underloaded fragments. To this end, we estimate a budget B, *e.g.,* average computational cost of the 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 candidates will be processed in the next phase.



(a) an edge-cut (b) a hybrid cut

Fig. 2: EMigrate and ESplit

Example 9 Consider the edge-cut partition depicted in 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 as shown in Fig. 2(a). Note that the migration operation 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, in order to further balance workload. For graphs with skewed degree distribution, Emigrate may not suffice since the computational cost incurred by an e-cut node v (*e.g.,* super nodes that are incident to a large number of edges) may already exceed the budget. In that case, ESplit cuts vertex 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 10 When 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 vertex 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 vertex t_2 does not keep 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. □

Redistributing 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 algorithm $\mathcal{A}$, E2H utilizes an MAssign phase to update master node assignments and redistribute communication cost. Note that MAssign does not increase the computational cost 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 $F_{i_1}, \ldots, F_{i_k}$

```
Input: Edge-cut HP_E(n) = (F_1, ..., F_n),
       cost functions h_A and g_A of A.
Output: Revised hybrid partition HP(n) = (F_1, ..., F_n).
1.  B ← Σ_{i=1}^{n} C^h_A(F_i)/n;   O ← ∅;   U ← ∅;
2.  for each i ∈ [1,n] do
3.    if C^h_A(F_i) > B then
4.      O ← O ∪ {F_i};   S_i ← GetCandidates(F_i, B);
5.    else U ← U ∪ {F_i};   S_i ← ∅;
6.  for each F_i ∈ O and (v, E^v_i) ∈ S_i do      /* EMigrate */
7.    for each F_j ∈ U do
8.      if C^h_A(F_j ∪ {(v, E^v_i)}) ≤ B then
9.        migrate (v, E^v_i) to F_j;   S_i ← S_i \ {(v, E^v_i)};
10.       break ;
11. for each F_i ∈ O and each (v, E^v_i) ∈ S_i do /* ESplit */
12.   for each e ∈ E^v_i do
13.     t ← argmin_{j∈[1,n]} C^h_A(F_j);   migrate (v, {e}) to F_t;
14.   S_i ← S_i \ {(v, E^v_i)}
15. adjust master node mapping;                  /* MAssign */
16. return HP(n) = (F_1, ..., F_n);
Procedure GetCandidates(F_i, B)
17. F'_i ← ∅;   k ← |V_i|;
18. let v_1, ..., v_k be a BFS traversal of e-cut nodes in F_i;
19. for each j ∈ [1,k] do
20.   if C^h_A(F'_i ∪ {(v_j, E^{v_j}_i)}) ≤ B then
21.     F'_i ← F'_i ∪ {(v_j, E^{v_j}_i)};
22. return F_i \ F'_i;
```

Fig. 3: Algorithm E2H

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 the minimum cost. More specifically, the master of v is assigned to F_{i^*}, where

$$i^* = \text{argmin}_{j\in\{i_1,\ldots,i_k\}} C^h_{\mathcal{A}}(F_j) + C^g_{\mathcal{A}}(F_j) + g^j_{\mathcal{A}}(v). \quad (5)$$

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

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 moves it to the fragment with the minimum computational cost.

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

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 (line 18). Following the BFS order, it includes vertices and its incident edges to F'_i within budget B in a greedy manner (lines 19-21). The vertices and edges excluded from F'_i are returned as migration candidates (line 22).

Example 11 We show how the algorithm E2H works on the edge-cut partition of Fig. 1(b) based on computation cost function h_{CN} and communication cost function g_{CN}=$5.57 \times 10^{-5} Dd^-_G$ learned in Example 8.

(1) Algorithm 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, fragment 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. As a result, the algorithm marks the e-cut nodes t_3 and t_2 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. Algorithm E2H then tries to migrate t_2 from F_1 to F_2. However, this operation is aborted as it would increase $C^h_{\mathsf{CN}}(F_2)$ from 1.12×10^{-3}ms to 1.95×10^{-3}ms, which exceeds 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 (see Example 10). It ends up with the partition depicted in Fig. 2(b).

(5) To further reduce the parallel execution cost of CN, E2H updates the master node mapping by MAssign. 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. Their communication costs are $g_{\mathsf{CN}}(s_1) = 1.45\times10^{-4}$ms, $g_{\mathsf{CN}}(s_3) = 2.17\times10^{-4}$ms and $g_{\mathsf{CN}}(s_4) = 2.90 \times 10^{-4}$ms. Since $C^h_{\mathcal{A}}(F_1) = 1.76 \times 10^{-3}$ms and

$C^h_{\mathcal{A}}(F_2)$=1.67 × 10^{-3}ms, the masters of s_1 and s_4 are assigned to F_2, while that 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 $\max_{i\in[1,2]} C_{\mathsf{CN}}(F_i) = \max\{1.98, 2.11\} \times 10^{-3}$ms= 2.11×10^{-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. This does not suffice for workload balance, as shown in Example 1. Also note that vertex-cut often incurs larger communication cost due to bad locality [24].

Overview of partitioner V2H. Given a vertex-cut $\mathsf{HP}_V(n)$ and cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of algorithm $\mathcal{A}$, hybrid partitioner 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 function $h_{\mathcal{A}}$, it first not only balances computational workload but also reduces communication cost. Guided by function $g_{\mathcal{A}}$, its second stage redistributes communication cost. Below we focus on the first stage; the second stage is similar to MAssign in E2H.

Balancing computational cost. This stage consists of two phases itself, namely, VMigrate and VMerge. As in E2H, it first estimates a cost budget B, *e.g.,* the average computational cost of all fragments. It classifies the fragments as overloaded and underloaded. Then VMigrate migrates v-cut nodes from overloaded fragments to underloaded ones in order 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. Like EMigrate, VMigrate employs the same procedure GetCandidate to identify candidate vertices and edges to migrate; it moves a vertex v together with all its *local* incident edges E^v_i in the migration. To retain the locality, it also requires that the destination fragment contains a copy of v. More specifically, a v-cut node v and its associated edges E^v_i 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
- $C^h_{\mathcal{A}}(F_j \cup \{(v, E^v_i\})\} \leq B$.

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

```
Input: Vertex-cut HP_V(n) = (F_1, ..., F_n),
       cost functions h_A and g_A of A.
Output: Revised hybrid partition HP(n) = (F_1, ..., F_n).
1.  B ← Σ_{i=1}^{n} C^h_A(F_i)/n;   O ← ∅;   U ← ∅;
2.  for each i ∈ [1, n] do
3.     if C^h_A(F_i) > B then
4.        O ← O ∪ {F_i};   S_i ← GetCandidates(F_i, B);
5.     else U ← U ∪ {F_i};   S_i ← ∅;
6.  for each F_i ∈ O and each (v, E^v_i) ∈ S_i do /*VMigrate*/
7.     for each F_j ∈ U such that v ∈ F_j do
8.        if v resides in F_j and C^h_A(F_j ∪ {(v, E^v_i)} ≤ B then
9.           migrate (v, E^v_i) to F_j;   S_i ← S_i \ {(v, E^v_i)};
10.          break ;
11. for each F_i ∈ U and each (v, E^v_i) ∈ F_i do /* VMerge */
12.    let Ē^v_i = E^v \ E^v_i;
13.    if C^h_A(F_i ∪ {(v, Ē^v_i)} ≤ B then
14.       migrate or replicate (v, Ē^v_i) to F_i;
15. adjust master node mapping;                  /* MAssign */
16. return HP(n) = (F_1, ..., F_n);
```

Fig. 4: Algorithm V2H

VMerge. This phase iteratively merges v-cut nodes into e-cut nodes. In each iteration, VMerge (a) first selects an underloaded fragment F_i and a v-cut node v in F_i; and (b) changes v to an e-cut node by 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 the minimum cost $C^h_{\mathcal{A}}(F_i)$. Phase VMerge continuously merges v-cut nodes until no valid changes can be made.

In contrast to VMigrate, phase VMerge may also move a v-cut node from an underloaded fragment. Nevertheless, it reduces the parallel cost of algorithm $\mathcal{A}$ by reducing its communication cost because the new e-cut node no longer requires communication to aggregate information from different fragments.

Algorithm V2H. We are now ready to present algorithm V2H, as shown in Fig. 4. Given a vertex-cut partition $\mathsf{HP}_V(n)$ and cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ of graph algorithm $\mathcal{A}$, V2H revises $\mathsf{HP}_V(n)$ to a hybrid partition $\mathsf{HP}(n)$ to reduce the parallel cost of $\mathcal{A}$. Using function $h_{\mathcal{A}}$, V2H first estimates a cost budget B as the average computational cost $\sum_{i=1}^{n} C^h_{\mathcal{A}}(F_i)/n$. It classifies the fragments as overloaded and underloaded; and for each overloaded fragment F_i, it identifies a set of v-cuts nodes S_i as migration candidates via procedure Get-

Candidates (lines 2-5). Then VMigrate of V2H migrates v-cut nodes and their incident edges from overload fragments to underloaded ones (lines 6-10). After that, V2H carries out phase VMerge to merge v-cut nodes into e-cut nodes, to further balance computational workload and reduce communication cost (lines 11-14). In the end, V2H adjusts master node mapping to redistribute communication cost as in E2H (line 15).

Example 12 Applying VMerge to v-cut node v_2 in fragment F_1 of Fig. 1(e), it replicates edge (v_2, v_3) at F_1. Now vertex 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.

Parallel setting. We adopt a shared-nothing distributed setting as commonly used nowadays.

(a) The fragments $F_1, \ldots, F_n$ of the input edge-cut (resp. vertex-cut) partition are initially distributed to n shared-nothing workers $P_1, \ldots, P_n$, respectively. The workers run under synchronous BSP model [48], which separates the computation into supersteps.

(b) Based on the cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$, each worker maintains a shared state, including costs $C^h_{\mathcal{A}}(F_i)$ and $C^g_{\mathcal{A}}(F_i)$ of each fragment F_i and other cost-related metrics for the vertices and edges processed.

(c) In each superstep, each worker conducts a small batch of computation to refine the partition and exchange updates to synchronize the shared state.

More specifically, below we show how to parallelize phases EMigrate, ESplit and MAssign of E2H; the parallelization of V2H can be done in a similar way.

Parallel EMigrate. In each 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. Let $P_{i_1}, \ldots, P_{i_k}$ be the 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 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 candidate is either accepted by some P_{i_j} or rejected by all $P_{i_1}, \ldots P_{i_k}$.

Parallel ESplit. In a superstep, each overloaded worker processes a small batch of edges incident to the candidates rejected in the previous phase, in parallel. Based on the shared state with $C^h_{\mathcal{A}}(F_i)$ of each F_i, it greedily assigns and migrates edges as shown in Fig. 3 (lines 12-13). At the end, the workers synchronize the edge assignments and update the shared state. The process terminates after processing all such edges.

Parallel MAssign. In a superstep, each worker selects a small batch of unassigned vertices in $F_i.O$, and adjusts master nodes by Equation (5) given earlier, 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 vertices in $\mathcal{F}.O$ are processed.

Analysis. We now analyze the computational costs and communication costs of the parallel phases. Let c_1 and c_2 be the cost to evaluate the functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ on a vertex, respectively; and let c_3 be the communication cost to synchronize the shared state in a superstep. Assume *w.l.o.g.* that each phase takes b as the batch size.

(1) The computational cost of parallel EMigrate can be bounded by $n \cdot c_1|V|$. This is because each migration candidate can be tried for at most n times and there exist at most $|V|$ migration candidates. The communication cost is bounded by $\frac{c_3|V|}{b}$ since parallel EMigrate is executed for at most $\frac{|V|}{b}$ supersteps.

(2) The computational cost of parallel ESplit is bounded by $c_2|E|$. This is because in the worst case, a super node can be adjacent to every other vertex. The communication cost can be bounded by $\frac{c_3|E|}{b}$.

(3) The computation cost of parallel MAssign is bounded by $\frac{(c_1+c_2)|V|}{n}$ since at most $|V|$ vertices are cut by parallel ESplit, and the master assignment operations can be evenly distributed among n fragments. The communication cost of parallel MAssign is bounded by $\frac{c_3|V|}{n \cdot b}$.

6 Composite Application Driven Partitioners

We have seen how to partition a graph G for a target application $\mathcal{A}$ to improve the parallel execution of $\mathcal{A}$. In the real world, however, users are often required to run multiple different analytical tasks on the same graph G at the same time. As remarked in Section 1, the cost patterns differ for different applications; a partition that fits one application may become skewed for

another, *e.g.,* the partition depicted in Fig. 1(b) is balanced for PR but skewed for CN, while the one shown in Fig. 1(c) is balanced for CN but skewed for PR.

A brute-force approach is to generate multiple partitions of graph G separately, one for each of the applications in use, to reduce the cost of each application. However, this incurs unnecessary partitioning time and space costs, and worse yet, introduces the coherence problem when graph G is updated as commonly found in practice. One naturally wants a better solution to support a batch of algorithms on G, such that (1) we want to reduce the time and space costs of generating multiple partitions separately, and (2) the parallel cost of each algorithm $\mathcal{A}$ in the batch is comparable to its counterpart when G is partitioned by E2H or V2H and is tailored for reducing the cost of specific $\mathcal{A}$.

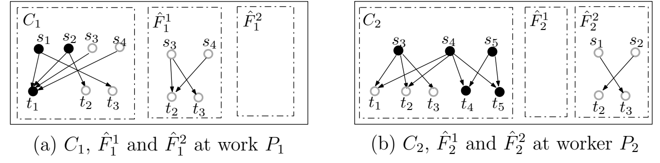
Motivated by the practical need, in this section we extend our application-driven partitioners to support mixed workloads, *i.e.,* to partition an input graph G and reduce the overall parallel cost for a batch of graph algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_k$. We first introduce *composite partitions* as a compact representation for the combinations of different partitions of the same graph (Section 6.1); we then extend our hybrid partitioner E2H (resp. V2H) to *composite application-driven partitioner* ME2H (resp. MV2H) to deal with mixed workloads (Sections 6.2 and 6.3). Finally we briefly show how to parallelize ME2H and MV2H (Section 6.4).

6.1 Composite Partition

Given a graph G, the number n of fragments in a partition, and a batch consisting of k algorithms $\{A_1, \ldots, \mathcal{A}_k\}$, we want to compute a composite partition, which is essentially equivalent to a collection of k separate n-way partitions, *i.e.,* $\mathsf{HP}(n,k) = \{\mathsf{HP}^1(n), ..., \mathsf{HP}^k(n)\}$, where $\mathsf{HP}^j(n)$=$\{F_1^j, ..., F_n^j\}$ and it is the partition for reducing the parallel cost of algorithm $\mathcal{A}_j$.

More specifically, a *composite partition* $\mathsf{HP}(n,k)$ of G is $\{\mathsf{HP}^1(n), ..., \mathsf{HP}^k(n)\}$ such that each fragment $F_i^j \in \mathsf{HP}^j(n)$ is stored in two disjoint parts, *i.e.,* $F_i^j = C_i \cup \hat{F}_i^j$. Here $C_i = \bigcap_{j\in[1,k]} F_i^j$ and $\hat{F}_i^j = F_i^j \setminus C_i$. That is, the *core* C_i is the overlapped area in all partitions $\mathsf{HP}^j(n)$ for fragment F_i, and $\hat{F}_i^j$ is the residual area in F_i^j.

Intuitively, the composite partition provides a compact storage format for k individual partitions tailored for specific algorithms. Since core C_i is the area shared by all partitions in F_i, we only need to store it once. Moreover, when accessing a vertex and its incident edges, we only need to access two parts, namely, the core C_i and the residual $\hat{F}_i^j$ in F_i^j, *i.e.,* it only incurs a small overhead. As will be verified in our experimental study (Section 7), composite partitions effectively reduce the time cost of partitioning and the space cost of storing the partition, while retaining the property of cost-driven partitions for each algorithm in the batch.



(a) C_1, $\hat{F}_1^1$ and $\hat{F}_1^2$ at work P_1 (b) C_2, $\hat{F}_2^1$ and $\hat{F}_2^2$ at worker P_2

Fig. 5: A composite partition

<u>*Composite replication ratio.*</u> To measure the compactness of a composite hybrid partition, we further extend the replication ratio to composite ones, denoted as f_c. Here $f_c = \frac{1}{|G|}\sum_{i\in[1,n]}(|C_i| + \sum_{j\in[1,n]}|\hat{F}_i^j|)$, *i.e.,* it measures the ratio of space usage of a composite partition to the size of the original graph.

Example 13 Now consider combining the partitions Figures 1 (b) and (c) of graph G_1 shown in Fig 1 (a). Their composite is shown in Fig 5. At each worker P_i, each of the two partitions consists of a core C_i, and two residual parts $\hat{F}_i^1$ and $\hat{F}_i^2$. Here partition in Fig 1 (b) can be obtained by combining C_i and $\hat{F}_i^1$ for $i \in [1,2]$, where the partition in Fig 1 (c) is the combination of C_i and $\hat{F}_i^2$. Note that we use two separate vertices for s_3, s_4, t_2, t_3 in F_1^1 (resp. t_2, t_3 in F_2^2) only for the ease of representation. There is no need to replicate them in practice, since they are co-located with C_1 (resp. C_2).

The composite replication ratio $f_c = \frac{1}{|G_1|}(|C_1| + |C_2| + |\hat{F}_1^1| + |\hat{F}_2^1| + |\hat{F}_1^2| + |\hat{F}_2^2|) = 1.52$. That is, storing the composite partition of these two partitions only incurs $1.52|G_1|$ space. In contrast, storing these two partitions separately incurs a space cost of $2.9|G_1|$. That is, composite partitions reduces space usage by storing the overlapped areas C_i only once. The larger $|C_i|$ is, the lower f_c is and the more space is saved. □

<u>*Coherence.*</u> Composite partition also makes it easier to update the graph coherently across different partitions. Here we only consider the updates of edge deletion and edge insertion. The updates of vertex insertion and vertex deletion can be handled in a similar way.

Edge deletion. To delete an edge e in G coherently, we first need to locate all copies of e in the composite partition $\mathsf{HP}(n,k)$. To this end, we build an index for each composite fragment that consists of a core C_i and k residual fragments $F_i^1, \ldots, F_i^k$. The index maps an edge e to a pair (c_i, r_i), where c_i is a Boolean indicating whether e is in C_i and $r_i = \{j_1, \ldots, j_\ell\}$ corresponds the set of residual fragments $F_i^{j_1}, \ldots, F_i^{j_\ell}$ that contain

e. Observe that when $c_i = \mathsf{true}$, the set $r_i = \emptyset$ and it incurs only a constant space cost. Thus the index is relatively small, when partitions heavily overlap with each other, leaving very few edges outside of C_i. Instead, a poorly aligned partition, *i.e.,* a composition partition with a small overlapping area, needs a large index.

In contrast, if k partitions of $F_i^1, ..., F_i^k$ are stored separately, such index maps each edge in $\bigcup_{j\in[1,k]} F_i^j$ to a nonempty set of partitions F_i^j that contains it. The index size could be as large as $|\bigcup_{j\in[1,k]} F_i^j|$, which is inefficient in both time and space.

Edge insertion. Different from edge deletion, when inserting an edge to G, it has to be mapped to fragments. Hence the edge carries with it the target fragments for insertion. This said, composite partitions still speed up the insertion, when the inserted edge falls into some core C_i. In this case, the insertion is performed only once and the index is updated by mapping the inserted edge to $(\mathsf{true}, \emptyset)$ at C_i. In contrast, for k individual partitions that are separately stored, we need to insert the edge to each of them and update the corresponding index each time, without little chance to reduce its cost.

6.2 From Edge Cut to Composite Hybrid Cut

We first extend E2H to a composite hybrid partitioner ME2H to handle mixed workloads. Algorithm ME2H aims to produce a composite hybrid partition to reduce the parallel cost of each algorithm in a workload.

Overview of algorithm ME2H. Given an edge-cut partition $\mathsf{HP}_E(n)$ and cost functions $h_{\mathcal{A}_1}, ..., h_{\mathcal{A}_k}$ and $g_{\mathcal{A}_1}, ..., g_{\mathcal{A}_k}$ of k algorithms $\mathcal{A}_1, ..., \mathcal{A}_k$, ME2H generates a composite hybrid partition $\mathsf{HP}(n, k)$. For each $j \in [1, k]$, it refines $\mathsf{HP}_E(n)$ into a hybrid partition $\mathsf{HP}^j(n)$ in $\mathsf{HP}(n, k)$, guided by the cost functions of $\mathcal{A}_j$. In addition to balancing computational cost and retaining the locality, ME2H also reduces replication f_c of the output composite partition. The reasons are twofold: when f_c is low, (1) less space is required to store the composite partition, and (2) the composite partitioner is more efficient, since a large part of the partitioning results are shared among k individual partitions.

Like E2H, algorithm ME2H consists of two stages, namely, workload balancing guided by $h_{\mathcal{A}_1}, \ldots, h_{\mathcal{A}_k}$, and communication cost re-distribution guided by $g_{\mathcal{A}_1}, \ldots, g_{\mathcal{A}_k}$. Below we focus on the first stage, since the second stage is similar to the phase MAssign in E2H.

Balancing computational cost. ME2H outputs a load balanced partition $\mathsf{HP}^j(n)$ for each application $\mathcal{A}_j$ ($j \in [1, k]$), by extending the phases EMigrate and ESplit of E2H to phases VAssign and EAssign, respectively. Different from E2H that mitigates unbalanced workload by exchanging vertices and edges among fragments, VAssign and EAssign of ME2H iteratively assign elements from the input partition $\mathsf{HP}_E(n)$ to n initially empty fragments $F_1^j, ..., F_n^j$ of $\mathsf{HP}^j(n)$, for each $\mathcal{A}_j$ ($j \in [1, k]$). Both phases ensure that no fragment becomes overloaded during the process. To this end, we estimate a budget B^j for each target hybrid partition $\mathsf{HP}^j(n)$, which is the average computational cost tailored by $h_{\mathcal{A}_j}$. The budget B^j serves to retain the balance of $\mathsf{HP}^j(n)$: a vertex or edge in G is first assigned to a fragment F_i^j of partition $\mathsf{HP}^j(n)$ only if $C^h_{\mathcal{A}_j}(F_i^j) \leq B^j$.

VAssign. This phase iteratively *assigns e-cut nodes* and their incident edges to $\mathsf{HP}^j(n)$ for each $\mathcal{A}_j$, while ensuring the balancing constraints. Like E2H, it assigns each e-cut node together with its incident edges as a whole. Different from E2H, ME2H also tries to lower the composite replication factor f_c for the output partitions $\mathsf{HP}(n, k) = \{\mathsf{HP}^j(n) \mid 1 \leq j \leq k\}$, *i.e.,* to make them overlap with each other as much as possible.

EAssign. This phase further assigns edges one by one and turns e-cut nodes into v-cut ones by assigning its incident edges to different fragments. Note that after phase VAssign, there might be vertices and edges in G that are not yet assigned to $\mathsf{HP}^j(n)$ for some $\mathcal{A}_j$. This is because in VAssign a fragment F_i^j ensures that its cost estimated by $h_{\mathcal{A}_j}$ does not exceed its budget B^j throughout the process. This may leave some parts of graph unassigned, *e.g.,* vertices with high degree that may incur large computational costs. The residual graph is handled by EAssign. The process terminates when all vertices and edges in $\mathsf{HP}_E(n)$ have been assigned to each partition $\mathsf{HP}^j(n)$, for $j \in [1, k]$.

We next present the algorithm ME2H in Fig. 6.

Algorithm ME2H. Given an edge-cut partition $\mathsf{HP}_E(n)$ and cost functions, $h_{\mathcal{A}_1}, ..., h_{\mathcal{A}_k}$ and $g_{\mathcal{A}_1}, ..., g_{\mathcal{A}_k}$, of applications $\mathcal{A}_1, \ldots \mathcal{A}_k$, ME2H revises $\mathsf{HP}_E(n)$ to hybrid partitions $\mathsf{HP}^1(n), ..., \mathsf{HP}^k(n)$ for $\mathcal{A}_1, ..., \mathcal{A}_k$, respectively, in a batch. For each algorithm $\mathcal{A}_j (j \in [1, k])$, ME2H first sets a computational cost budget B^j by using cost function $h_{\mathcal{A}_j}$(line 1). It then initializes fragments $F_i^1, \ldots, F_i^k$ in each fragment F_i by calling procedure Init (line 3). Procedure Init (described blow) ensures that the initialized fragment F_i^j ($i \in [1, n], j \in [1, k]$) does not exceed budget B^j of $\mathcal{A}_j$. ME2H next identifies the overlapped subgraph of $F_i^1, \ldots F_i^k$ as core C_i of F_i ($i \in [1, n]$), where the allocations of edges and vertices are shared by all k partitions $\mathsf{HP}^1(n), \ldots \mathsf{HP}^k(n)$, and do not require any further changes (line 4). It marks the subgraph in F_i

Input: Edge-cut $\mathsf{HP}_E(n) = (F_1, \ldots, F_n)$, $2k$ cost functions
$h_{\mathcal{A}_1}, ..., h_{\mathcal{A}_k}, g_{\mathcal{A}_1}, ..., g_{\mathcal{A}_k}$ of applications $\mathcal{A}_1, \ldots, \mathcal{A}_k$.
Output: A composite partition $\mathsf{HP}(n, k)$ for $\mathcal{A}_1, \ldots, \mathcal{A}_k$.
1. **for each** $j \in [1, k]$ **do** $B_j \leftarrow \sum_{i=1}^{n} C^h_{\mathcal{A}_j}(F_i)/n$;
2. **for each** $i \in [1, n]$ **do**
3. $(F_i^1, ..., F_i^k) \leftarrow \mathsf{Init}(F_i, B_1, B_2, ..., B_k)$;
4. $C_i \leftarrow \bigcap_{j\in[1,k]} F_i^j$;
5. **for each** $j \in [1, k]$ **do** $S_i^j \leftarrow F_i \setminus F_i^j$;
6. **for each** $j \in [1, k]$ **do**
7. $\mathcal{U}^j \leftarrow \{F_i^j \mid C^h_{\mathcal{A}_j}(F_i^j) < B_j \bigwedge S_i^j = \emptyset\}$;
8. **for each** $i \in [1, n]$ and **each** $(v, E_i^v) \in F_i \setminus C_i$ **do**
9. $\mathcal{O}_v \leftarrow \{\mathcal{A}_j \mid (v, E_i^v) \in S_i^j\}$; /*VAssign*/
10. $d_v() \leftarrow \mathsf{GetDest}(\mathcal{O}_v, \mathcal{U}^1, \mathcal{U}^2, ..., \mathcal{U}^k)$;
/* $d_v()$ is a mapping from $\mathcal{O}_v$ to $[1, n]$ */
11. **for each** $\mathcal{A}_j \in \mathcal{O}_v$ **let** $x = d_v(\mathcal{A}_j)$ **do**
12. assign (v, E_i^v) to F_x^j; update $\mathcal{U}^j$;
13. $S_i^j \leftarrow S_i^j \setminus \{(v, E_i^v)\}$;
14. **for each** $i \in [1, n], j \in [1, k]$ and **each** $(v, E_i^v) \in S_i^j$ **do**
15. **for each** $e \in E_i^v$ **do** /*EAssign*/
16. $x \leftarrow \text{argmin}_{y\in[1,n]} C^h_{\mathcal{A}}(F_y^j)$;
17. assigns $(v, \{e\})$ to F_x^j;
18. $S_i^j \leftarrow S_i^j \setminus \{(v, E_i^v)\}$;
19. adjust master node mapping; /*MAssign*/
20. **return** $(\mathsf{HP}^1(n), \ldots, \mathsf{HP}^k(n))$ as $\mathsf{HP}(n, k)$.

Fig. 6: Algorithm ME2H

Procedure $\mathsf{Init}(F_i, B_1, B_2, ..., B_k)$
1. **for each** $j \in [1, N]$ **do** $F_i^j \leftarrow \emptyset$;
2. $N \leftarrow |V_i|$, $S \leftarrow \{F_i^j \mid j \in [1, k]\}$;
3. let $v_1, \ldots, v_N$ be a BFS traversal of e-cut nodes in F_i;
4. **for each** $x \in [1, N]$ **do**
5. **for each** $F_i^j \in S$ **do**
6. **if** $C^h_{\mathcal{A}_j}(F_i^j \cup \{(v_x, E_i^{v_x})\}) \leq B_j$ **then**
7. $F_i^j \leftarrow F_i^j \cup \{(v_x, E_i^{v_x})\}$;
8. **return** $(F_i^1, F_i^2, ..., F_i^k)$;

Procedure $\mathsf{GetDest}(\mathcal{O}_v, \mathcal{U}^1, ..., \mathcal{U}^k)$
1. **while** $\mathcal{O}_v$ is not empty **do**
2. $i \leftarrow \arg_x \max |\{j \mid \mathcal{A}_j \in \mathcal{O}_v \wedge F_x^j \in \mathcal{U}^j\}|$;
3. $\mathcal{O}_\Delta \leftarrow \{\mathcal{A}_j \mid F_i^j \in \mathcal{U}^j\} \cap \mathcal{O}_v$;
4. **for each** $\mathcal{A}_j \in \mathcal{O}_\Delta$ **do** $d_v(\mathcal{A}_j) \leftarrow i$;
5. $\mathcal{O}_v \leftarrow \mathcal{O}_v \setminus \mathcal{O}_\Delta$;
6. **return** $d_v()$;

Fig. 7: Procedure Init and GetDest

that is not yet assigned to $\mathsf{HP}^j(n)$ as candidates S_i^j for later processing (line 5). It also marks fragment F_i^j in $\mathcal{U}^j$, if it is underloaded *w.r.t.* the budget B^j (lines 6-7).

ME2H next processes the candidates in S_i^j by VAssign and EAssign, where $i \in [1, n]$ and $j \in [1, k]$.

(1) ME2H first executes phase VAssign (lines 8-13). Each time it processes a candidate (v, E_i^v) from $F_i \setminus C_i$. Observe that $\mathcal{O}_v = \{\mathcal{A}_j \mid (v, E_i^v) \in S_i^j\}$ is the set of algorithms, where $\mathcal{A}_j$ needs to migrate (v, E_i^v) out of fragment F_i, as the computational cost exceeds the bound. VAssign computes the destination $d_v(\mathcal{A}_j)$ of (v, E_i^v) for each such algorithm $\mathcal{A}_j \in \mathcal{O}_v$ by procedure GetDest. Here d_v is a mapping that maps $\mathcal{A}_j$ to some $x \in [1, n]$, where $F_x^j \in \mathcal{U}^j$ is a destination fragment for candidate (v, E_i^v) (lines 10-11). GetDest also makes an effort to minimize the total number of destination fragments for $\mathcal{O}_v$, to reduce replication factor f_c (see below). The phase ends when all candidates have been processed.

(2) After VAssign, ME2H allocates all edges that remain in each residual S_i^j by EAssign (lines 14-18). It iterates through edges of $E_i^v \in S_i^j$ for a vertex v, and allocates them individually to the fragment with the minimum workload (computational cost) (lines 16-17).

ME2H next revises the master node mapping to further reduce the parallel cost of each $\mathcal{A}_j$ on $\mathsf{HP}^j(n)$ just like E2H (line 19). In the end, it returns $(\mathsf{HP}^1(n), \ldots, \mathsf{HP}^k(n))$ as the composite partition (line 20).

We next elaborate procedures Init and GetDest of ME2H. Intuitively, both procedures assign e-cut nodes and their incident edges to fragments of individual partitions $\mathsf{HP}^1(n), \ldots, \mathsf{HP}^k(n)$. As discussed earlier, apart from balancing workload and retaining locality, ME2H also aims to reduce the composite replication factor f_c to save space. To this end, procedure Init enlarges the overlapped areas $C_1, \ldots, C_k$ as much as possible; and procedure GetDest reduces the overall replication for candidates (v, E_i^v) that are not in C_i.

Procedure Init. As shown in Fig. 7, given a fragment F_i and budgets $B_1, \ldots, B_k$ for algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_k$, respectively, Init initializes fragments $F_i^1, \ldots, F_i^k$ in F_i. Similar to E2H, in order to retain the locality, it first computes a BFS order of the vertices in F_i (lines 1-3). Following the order, each time procedure Init assigns a candidate (v_x, E_x^v) to fragments $F_i^1, \ldots, F_i^k$ within their budgets, whenever possible (lines 4-7). This helps us to enlarge the overlapped area C_i in F_i.

Procedure GetDest. Given a set $\mathcal{O}_v$ of algorithms and collections of underloaded fragments $\mathcal{U}^1, ..., \mathcal{U}^k$ for $\mathcal{A}_1$, $\ldots$, $\mathcal{A}_k$, respectively, GetDest computes a mapping d_v from $\mathcal{O}_v$ to $[1, n]$ such that the following conditions hold: (i) $F^j_{d_v(\mathcal{A}_j)} \in \mathcal{U}^j$, *i.e.,* d_v identifies an underloaded fragment in $\mathcal{U}^j$ as the destination for (v, E^v); and (ii) the cardinality of a set D_v defined by $D_v = \{d(\mathcal{A}_j) \mid \mathcal{A}_j \in \mathcal{O}_v\}$ is minimized, *i.e.,* the number of destination fragments is the minimum. When the D_v is minimized, the number of replications of (v, E_v) is reduced and so is the composition replication factor f_c.

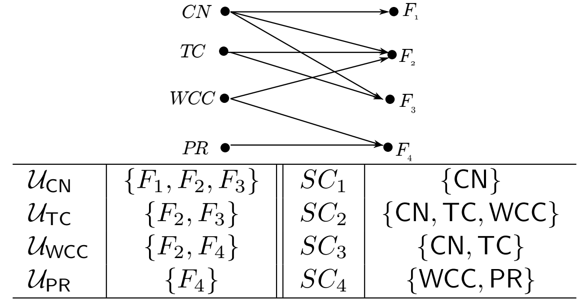
Intuitively, GetDest is to find the minimal number of underloaded fragments to cover the assignments of (v, E^v) for each algorithm $\mathcal{A}_j \in \mathcal{O}_v$. This optimization problem corresponds to the Minimum Set Cover (MSC) problem, which is NP-complete [45]. Given a collection $\mathcal{C}$ of subsets of a finite set S, a *set cover* is a subset $\mathcal{C}' \subseteq \mathcal{C}$ such that every element in S belongs to at least one element in $\mathcal{C}'$. The MSC over input $\langle S, \mathcal{C}\rangle$ is to compute a set cover $\mathcal{C}'$ with the minimum cardinality $|\mathcal{C}'|$. Suppose that $S = \{s_1, \ldots, s_m\}$ and $\mathcal{C} = \{SC_1, \ldots, SC_k\}$. To see the correspondence between GetDest and MSC, we construct an instance of GetDest as follows: (1) $\mathcal{O}_v = \{\mathcal{A}_{s_1}, \ldots, \mathcal{A}_{s_m}\}$; and (2) for each subset SC_i, we define $\mathcal{U}^i = \{F_i^{s_j} \mid s_j \in SC_i\}$ as the set of underloaded fragments in F_i that can server as assignment destinations. One can verify that an optimal solution to $\mathsf{GetDest}(\mathcal{O}_v, \mathcal{U}^1, \ldots, \mathcal{U}^k)$ corresponds to an optimal solution to MSC over the input $\langle S, \mathcal{C}\rangle$.

Due to the inherent hardness, here we adopt the following greedy heuristic [17] as GetDest, shown in Fig. 7. Each time it selects $i \in [1, n]$ such that the set $SC_i = \{\mathcal{A}_j \mid F_i^j \in \mathcal{U}_j, \mathcal{A}_j \in \mathcal{O}_v\}$ covers the most of elements in $\mathcal{O}_v$ (line 11). Elements covered are then removed from $\mathcal{O}_v$ (lines 12-13). The procedure terminates when all elements in $\mathcal{O}_v$ have been covered.

Example 14 Let (v, E_i^v) be the candidates to be considered and assume that $\mathcal{O}_v = \{\mathsf{CN}, \mathsf{TC}, \mathsf{WCC}, \mathsf{PR}\}$, where WCC is an algorithm for computing weak connected components. Figure 8 shows the underloaded fragments $\mathcal{U}_{\mathcal{A}}$ *w.r.t.* algorithms $\mathcal{A} \in \mathcal{O}_v$ during the process of EMigrate. For instance, $\mathcal{U}_{\mathsf{CN}} = \{F_1, F_2, F_3\}$ indicates that fragment F_1, F_2 and F_3 are currently underloaded and can serve as assignment destinations for application CN. The sets SC_i is the reverse index of $\mathcal{U}_{\mathcal{A}}$, which maps the fragment F_i to the algorithms that include F_i in $\mathcal{U}_{\mathcal{A}}$.

(a) Procedure GetDest maps each algorithm to one fragment. The set of destination fragments is minimal, since it corresponds to the number of replications for candidate (v, E_i^v). One can see that this corresponds to computing a minimal set cover in the MSC problem. This is because (1) when all $\mathcal{U}_{\mathcal{A}}$ has been mapped to a destination fragment, its inverse index SC_i covers all four algorithms in $\mathcal{O}_v$; and (2) the number of destination fragments, *i.e.,* the number of sets, is the minimum.

(b) We next show how procedure GetDest computes destination mapping d_v. It first picks fragment F_2, since SC_2 covers most applications in the batch. That is, we make F_2 the destination of the current vertex v for partitions of CN, TC and WCC. Then GetDest chooses fragment F_4 for PR, which is its only possible destination fragment. As a result, the current vertex, along with its



$\mathcal{U}_{\mathsf{CN}}$	$\{F_1, F_2, F_3\}$	SC_1	$\{\mathsf{CN}\}$
$\mathcal{U}_{\mathsf{TC}}$	$\{F_2, F_3\}$	SC_2	$\{\mathsf{CN}, \mathsf{TC}, \mathsf{WCC}\}$
$\mathcal{U}_{\mathsf{WCC}}$	$\{F_2, F_4\}$	SC_3	$\{\mathsf{CN}, \mathsf{TC}\}$
$\mathcal{U}_{\mathsf{PR}}$	$\{F_4\}$	SC_4	$\{\mathsf{WCC}, \mathsf{PR}\}$

Fig. 8: An example of GetDist and MSC

incident edges, are replicated in two copies, one in F_2 in the partition shared by algorithms CN, TC and WCC, and the other in the partition of PR. □

6.3 From Vertex Cut to Composite Hybrid Cut

We next extend V2H to a composite hybrid partitioner MV2H for mixed workloads. MV2H takes a vertex-cut partition as input and outputs a composite partition to reduce the parallel cost for a batch of algorithms.

Overview of algorithm MV2H. Given a vertex-cut partition $\mathsf{HP}_V(n)$ and cost functions of $h_{\mathcal{A}_1}, ..., h_{\mathcal{A}_k}$ and $g_{\mathcal{A}_1}, ..., g_{\mathcal{A}_k}$ of applications $\mathcal{A}_1, \ldots, \mathcal{A}_k$, MV2H computes a composite partition $\mathsf{HP}(n, k) = \{\mathsf{HP}^1(n), ..., \mathsf{HP}^k(n)\}$. For each $j \in [1, k]$, $\mathsf{HP}_V(n)$ is refined into a hybrid partition $\mathsf{HP}^j(n)$ in $\mathsf{HP}(n, k)$, tailored for $\mathcal{A}_j$ to reduce its parallel cost $\max_{i \in [1,n]} C_{\mathcal{A}_j}(F_i)$. Different from V2H, in addition to balancing computational cost and communication cost, MV2H also aims to reduce the replication ratio f_c of the output composite partition.

Like V2H, MV2H also has three stages, namely, VAssign, VMerge and MAssign. Below we focus on VAssign; the other two are similar to their counterparts in V2H.

Balancing computational cost. Like V2H, the first stage VAssign of MV2H aims to balance the computational cost of $C^h_{\mathcal{A}_j}(F_i)$ for all algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_k$.

VAssign. This phase iteratively assigns *v-cut nodes* to each hybrid partitions in $\mathsf{HP}(n, k)$; it ensures that no fragment F_i^j exceeds the balancing constraints imposed by $C^h_{\mathcal{A}_j}(F_i)$. Like the VAssign phase of ME2H, it first identifies a subgraph in F_i as C_i, such that $C^h_{\mathcal{A}_j}(C_i)$ is underloaded for all $j \in [1, k]$. Recall the additional objective of reducing the replication ratio f_c. The overlapped area C_i ($i \in [1, n]$) is initialized to be as large as possible. This is because the larger $|C_i|$ is, the smaller f_c is, as long as C_i does not violate the balancing constraints and is shared by all k partitions. Then it iteratively processes candidates (v, E_i^v) left in $F_i \setminus C_i$. Observe that (v, E_i^v) is in $F_i \setminus C_i$ if it exceeds the budget

of some algorithm $\mathcal{A}_j$. VAssign decides which fragment in $\mathsf{HP}^j(n)$ to allocate (v, E_i^v) with the same procedure GetDest of ME2H. That is, it assigns these candidates in such a way that the replication of (v, E_i^v) is minimized. This reduces the composite replication factor f_c.

6.4 Parallelization

Algorithms ME2H and MV2H are parallelized along the same lines as E2H and V2H (Section 5.3). Indeed, the only additional procedures are Init and GetDest, which are all local procedures; that is, they can be processed within a fragment F_i located on a single machine without communication with fragments on other machines.

We denote by ParME2H and ParMV2H the parallelized algorithms ME2H and MV2H, respectively.

7 Experimental Study

Using real-life and synthetic graphs, we conducted six sets of experiments to evaluate our application-driven partitioners and their composite extensions for their (1) effectiveness (Exp-1 & Exp-2), (2) efficiency (Exp-3 & Exp-4), (3) parallel scalability (Exp-5) and (4) the accuracy and efficiency of cost function learning (Exp-6).

Experimental setting. We start with the settings.

Datasets. We used three real-life graphs: (a) liveJournal [2], a social network with 4.8 million entities and 68 million relationships (edges); (b) Twitter [4], a social network with 42 million users and 1.5 billion links; and (c) UKWeb [5], a large crawled Web graph with 106 million pages and 3.7 billion hyperlinks.

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

Partitioners. We implemented our parallel hybrid partitioners ParE2H and ParV2H, as well as their composite extensions ParME2H and ParMV2H for mixed workloads, all in C++ and compared them with the following: (1) xtraPuLP [46], a state-of-the-art edge-cut partitioner; (2) Fennel [47], a streaming partitioner for edge-cut; (3) Grid [28], a hash partitioner for vertex-cut with provable bound on vertex replication; (4) NE [53], a state-of-the-art vertex-cut heuristic; (5) Ginger [16], a hybrid partitioner that revises Fennel; we evaluated Ginger to compare improvements over Fennel; and (6) TopoX [35], a hybrid partitioner that not only splits high-degree vertices, but also merges neighboring low-degree vertices into super nodes to avoid splitting.

To get a fair comparison when evaluating the effectiveness and efficiency of the baselines, 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 and TopoX as they already produce hybrid partitions.

Table 2: Hybrid partitioners notations

ParE2H, ParV2H	parallel hybrid partitioners
ParME2H, ParMV2H	composite parallel hybrid par.
ParHP	ParE2H and ParV2H
ParMHP	ParME2H and ParMV2H
HxtraPuLP, MxtrPuLP	hybrid par. based on xtraPuLP
HFennel, MFennl	hybrid par. based on Fennel
HGrid, MGrid	hybrid par. based on Grid
HNE, MNE	hybrid par. based on NE

Similarly, for parallel composite hybrid partitioners ParME2H (resp. ParMV2H), we denote by MxtraPuLP and MFennel (resp. MGrid a MNE) the variants derived from xtraPuLP and Fennel (resp. Grid and NE).

Notation. We summarize our hybrid partitioners in Table 2. For simplicity, we denote by ParMHP the composite hybrid partitioners ParME2H and ParMV2H; and by ParHP the partitioners ParE2H and ParV2H. Note that in Table 2, a partitioner that starts with letter "H" is a parallel hybrid partitioner derived from an existing baseline, while a partitioner that starts with "M" is a composite variant. For example, HxtraPuLP and MxtraPuLP are the parallel hybrid partitioner and its composite variant derived from xtraPuLP, respectively.

Graph algorithms. We used five graph algorithms in our experiments, including CN (Common Neighbor), TC (Triangle Counting), WCC (Weak Connected Components), PR (PageRank) and SSSP (Single Source Shortest Path). For hybrid partitions, we used their "partition-transparent" algorithms from [20,21], *i.e.,* algorithms that work correctly under different partitions, edge-cut, vertex-cut and hybrid partitions. For edge-cut and vertex-cut partitions, we implemented (non-transparent) versions for these algorithms with our best effort. The batch of algorithms is fixed for mixed workloads as $\{\mathsf{CN}, \mathsf{TC}, \mathsf{WCC}, \mathsf{PR}, \mathsf{SSSP}\}$, unless otherwise stated.

ML learning setting. To train the cost models, we ran each algorithm on 10 graphs as described in Section 4. The number of training (resp. testing) samples for CN, TC, WCC, SSSP and PR is 80,000 (resp. 20,000), which are sampled from the algorithms' running log. Regression models are constructed by Pytorch [42] and trained on a server with one NVIDIA Tesla V100 GPU (16GB). The learned cost functions are reported in Table 5, and are used as the inputs for our (composite) partitioners.

Table 3: Partition metrics of Twitter ($n = 96$)

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
TopoX	6.0	1	1.37	7.0	2.2

The experiments were conducted on GRAPE [22], an open source parallel system [6] based on graph-centric programming, deployed on 32 machines in an HPC cluster. Each machine is equipped with 12 cores powered by Xeon E5-2692V2 2.2GHz, 128GB RAM, and 10Gbps NIC. In the experiments, each fragment was processed by one worker running on an exclusive core. We used up to 128 workers for experiment. The workers are scattered evenly across the 32 machines. All experiments were repeated five times. The average is reported here.

Experimental results. We next report our findings.

Exp-1: Effectiveness of application-driven partitioners. We first tested how application-driven hybrid partitions speed up graph algorithms' execution. Varying the partition number n from 16 to 128, we first evaluated the execution time of graph algorithms CN, TC, WCC, PR and SSSP under edge-cut, vertex-cut and their hybrid refinements. This is to test the effectiveness of hybrid partitioners tailored for a single graph algorithm, including ParE2H and ParV2H. The results are reported in Figures 9(a)-9(j). We find the followings.

(1) CN. Figures 9(a) and 9(b) report the execution time of CN on liveJournal and Twitter, respectively. The results on UKWeb are consistent (hence not shown). 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 and $\theta = \infty$ for liveJournal. On Twitter, with $n = 16$, CN reports OOM (out of memory), since it requires a large amount of memory to store large intermediate results during the process. The results tell us the following.

(a) By extending edge-cut of xtraPuLP and Fennel to hybrid partition, ParE2H improves the execution of CN by 4.5 (resp. 18.3) 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 computation load. For example, as shown in Table 3, on Twitter with $n = 96$, the balance factor λ_{CN} for CN of xtraPuLP and Fennel is 7.2 and 13.7, respectively. Equipped with ParE2H, the balance factor λ_{CN} of HxtraPuLP and HFennel drops to 1.4 and 1.3, respectively. As a consequence of its large λ_{CN}, CN of Fennel ran out-of-memory when running over Twitter.

(b) ParV2H improves CN on vertex-cut of Grid (resp. NE) by 4.3 (resp. 2.9) 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 3). By balancing workload and reducing communication, ParV2H improves Grid better than NE.

(c) Ginger is also a hybrid partitioner based on Fennel [16]. Compared with HFennel, it has smaller f_v, f_e, λ_e, but a larger λ_{CN} (see Table 3). As a result, HFennel beats Ginger on CN by 2.5 times on average, up to 3.3 times.

(d) TopoX is another hybrid partitioner that integrates Ginger with topology refactorization. For CN, HFennel beats TopoX by 1.9 times on average, up to 2.8 times. This is in accord with their λ_{CN}, *e.g.,* on Twitter when $n = 96$, the λ_{CN} of HFennel is only 60% of that of TopoX.

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

(2) TC. Figures 9(c) and 9(d) report the execution time of TC over liveJournal and Twitter, respectively. The results over UKWeb are consistent (not shown).

(a) ParE2H improves TC by balancing workload as for CN. On Twitter with $n = 96$, it improves TC over Fennel and xtraPuLP by 22 and 15 times, respectively. Further, TC over HFennel is 3.2 and 2.5 times faster than over Ginger and TopoX, respectively.

(b) ParV2H improves TC over the vertex-cut partition of Grid and NE by 2.3 and 1.8 times on average, up to 2.6 and 2.2 times, respectively. Here the speedup ratio 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) WCC. As shown in Figures 9(e) and 9(f), on average ParE2H improves WCC by 3.9 and 2.0 times over Twitter and UKWeb, respectively. By h_{WCC}, the edge size dominates the computational cost of WCC. ParE2H improves

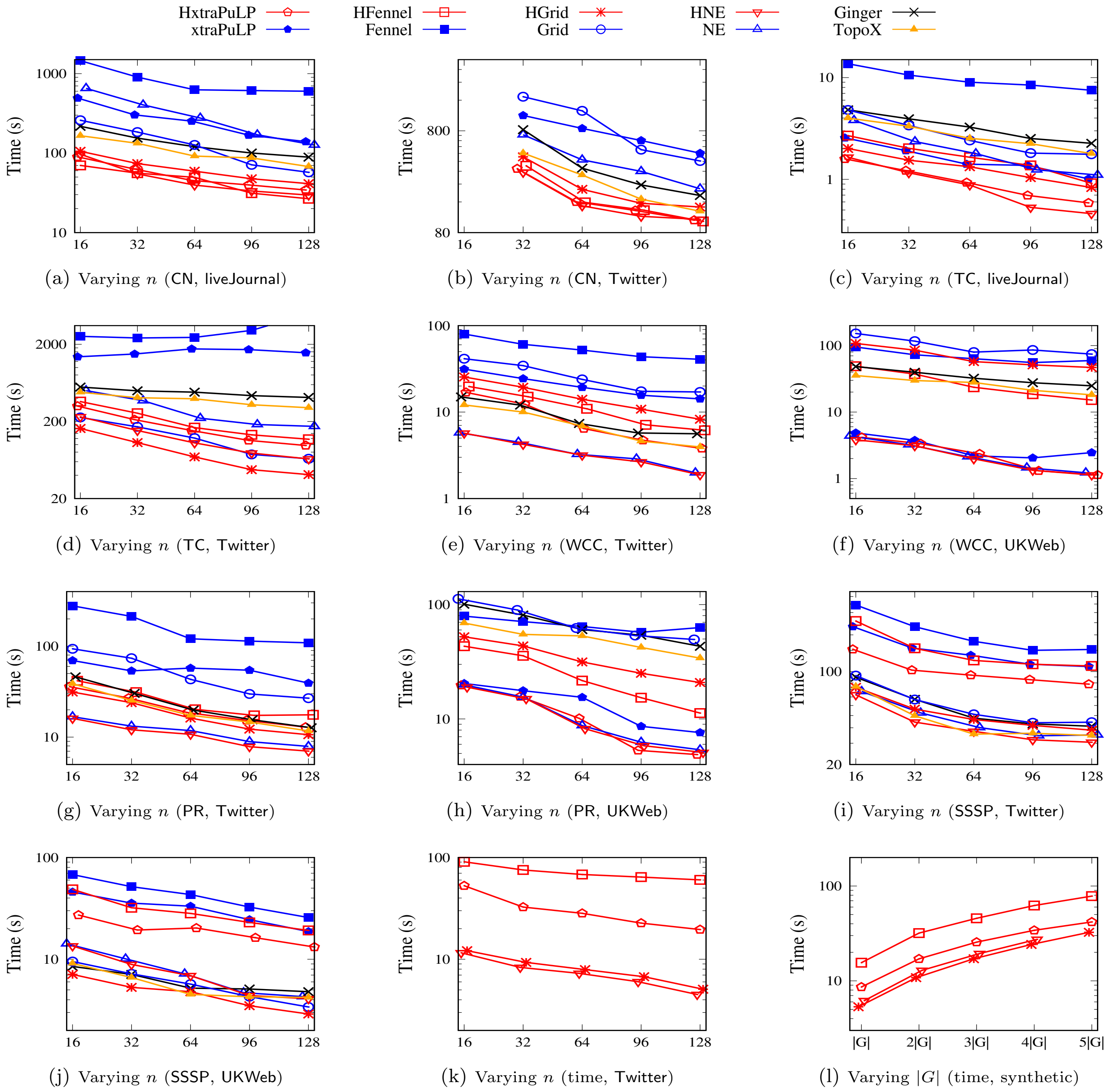


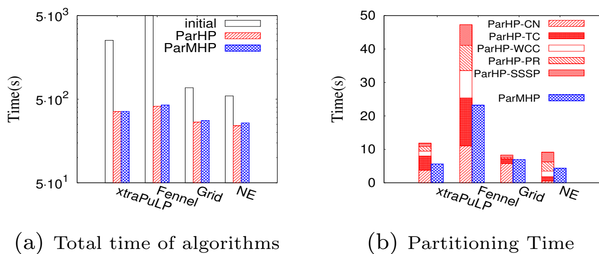
Fig. 9: Performance Evaluation of Application-driven Partitioners

Table 4: Running time of algorithm batch under ParMHP partitions (seconds, using Twitter, $n = 96$)

App.	MxtraPuLP	xtraPuLP	X	MFennel	Fennel	X	MGrid	Grid	X	MNE	NE	X	Ginger	TopoX
CN	130	642	4.9	132	$>$ 4000	$>$ 30	155	522	3.4	120	319	2.7	235	170
TC	118	1706	14	139	3027	22	47.9	74.6	1.6	79.0	182	2.3	431	329
WCC	4.92	15.7	3.2	7.15	43.7	6.1	10.9	17.4	1.6	2.64	2.87	1.1	5.72	4.61
PR	15.7	54.5	3.5	17.9	114	6.4	12.3	29.8	2.4	7.91	8.92	1.1	15.5	14.7
SSSP	86.9	113.6	1.3	113.5	154.4	1.4	39.4	41.2	1.1	30.7	32.9	1.2	40.0	34.4
$\mathbb{B}$	325.5	3022	9.3	410	$>$ 7339	$>$ 18	265.5	685	2.6	240	546	2.3	727	553

$\mathbb{B}$: algorithm batch {CN, TC, WCC, PR, SSSP}, X: speedup ratio

WCC by balancing the edge size based on h_{WCC}, while xtraPuLP and Fennel suffer from edge imbalance. Note that NE and HNE perform comparably on WCC, since NE also balances edge size; similarly for Ginger. Compared with ParE2H, ParV2H has smaller improvement on WCC since (a) vertex-cut partitions of Grid and NE



(a) Total time of algorithms (b) Partitioning Time

Fig. 10: Performance Evaluation of Composite Application-driven Partitioners (Twitter, $n = 96$)

have good edge balance (see Table 3); and (b) the communication overhead of WCC is quite small. The results on liveJournal are consistent (not shown).

(4) PR. As shown in Figures 9(g) and 9(h), on average, ParE2H improves PR by 4.6 and 2.3 times on Twitter and UKWeb, respectively. In contrast to WCC, ParV2H improves PR over vertex-cut of Grid by 2.7 and 2.1 times, respectively. This is because PR incurs larger communication cost than WCC, and ParV2H helps reduce this cost by VMerge and MAssign (see Appendix). The results on liveJournal are consistent (not shown).

(5) SSSP. Figures 9(i) and 9(j) show the performance of SSSP over Twitter and UKWeb, respectively. As we can see, (a) SSSP works the best under application-driven partitions; (b) the performance gap is smaller compared to the previous cases. On average, ParE2H and ParV2H improve SSSP by 45% and 12%, respectively. This is because (i) the workloads of SSSP under edge-cut by xtraPuLP and vertex-cut by NE are already balanced; thus not much can be improved via hybrid partitions; and (ii) like WCC, the communication overhead is small. The results remain consistent on high-diameter graphs. For instance, over traffic [3], a US road network with 23 million vertices, with $n = 96$, ParE2H and ParV2H speed up SSSP by 13.4% on average (not shown).

Exp-2: Effectiveness of composite partitioners. Fixing the batch of algorithms as $\{$CN, TC, WCC, PR, SSSP$\}$ and $n = 96$, we next tested the effectiveness of our ParMHP partitioners, including ParME2H and ParMV2H. Table 4 and Figure 10(a) report the combined running time of CN, TC, WCC, PR and SSSP over different composite ParMHP partitions, against the respective ParHP ones. The results tell us the following.

(a) The execution time of the algorithms over composite hybrid partitions of ParME2H (resp. ParMV2H) is comparable to those of ParE2H (resp. ParV2H). Compared with their application-driven counterparts, it takes only 0.6%, 3.4%, 4.4% and 8.2% more time to run all five algorithms over MxtraPuLP, MFennel, MGrid and MNE, respectively (see Figure 10(a)). That is, composite partitions of ParME2H and ParMV2H do not come at the cost of degradation of the effectiveness of application-driven partitions, while they take much less partitioning time and storage (see more details in Exp-4).

(b) It takes much less time to run all five algorithms on our refined composite hybrid partitions than any other baseline. The total time over composite hybrid partitions ranges from 240s of HNE to 410s of HFennel. In contrast, the total time over NE and Fennel is more than 546s, which is at least 1.3 times slower.

Exp-3: Efficiency of application-driven partitioners. We next evaluated the efficiency of partitioners ParE2H and ParV2H. Varying the partition number n from 16 to 128, we evaluated the time taken by ParE2H (resp. ParV2H) in hybrid partitioners HxtraPuLP and HFennel (resp. HGrid and HNE). We find the following.

(1) As shown in Figure 9(k), for TC on Twitter ($n = 128$), ParE2H takes 19.6s and 60.3s, *i.e.,* 1.9% and 12.9% of the total partitioning time of HxtraPuLP and HFennel, respectively. On average, ParE2H takes 28.6% 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 11.4 times on average, up to 22.5 times; Exp-1). The results are consistent for the other algorithms and graphs.

(2) With a smaller partition number, the hybrid partitioner ParE2H (resp. ParV2H) takes more time to refine an edge-cut (resp. vertex-cut) to a hybrid-cut. This is because more adjustment operations are needed for a partition when the partition number decreases.

(3) 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 datasets, ParV2H takes 0.2% and 19.5% of the total partitioning time in HNE and HGrid, respectively, while improving the performance of the algorithms by 3.6 times on average, up to 7.4 times (see Exp-1).

Exp-4: Efficiency of composite partitioners. We next evaluated the time and space efficiency of our composite application-driven partitioners ParMHP, including ParME2H and ParMV2H. We find the followings.

Time Efficiency of ParMHP. Fixing $n = 96$ and the batch of algorithms as $\{$CN, TC, WCC, SSSP, PR$\}$, we evaluated the time take by our composite hybrid par-

Table 5: Accuracy and training time of cost models

	Computational Cost Function $h_{\mathcal{A}}$	MSRE	time(s)	Communication Cost Function $g_{\mathcal{A}}$	MSRE	time(s)
CN	$9.23 \times 10^{-5} d_L^+(v) d_G^+(v) + 1.04 \times 10^{-6} d_L^+(v) + 1.02 \times 10^{-6}$	0.023	46.2	$5.57 \times 10^{-5} D d_G^-$	0.028	48.6
TC	$1.8 \times 10^{-3} d_L(v) + 1.7 \times 10^{-7} d_L(v) d_G(v)$	0.11	48.3	$8.42 \times 10^{-5} d_G(v) r(v) I(v)$	0.034	47.4
WCC	$6.53 \times 10^{-6} d_L(v) + 3.46 \times 10^{-5}$	0.021	49.8	$7.51 \times 10^{-5}(1.98 r(v) - 0.97)$	0.013	47.0
PR	$4.88 \times 10^{-5} d_L^+(v) + 4 \times 10^{-4}$	0.017	43.6	$6.60 \times 10^{-4} r(v) + 1.1 \times 10^{-4}$	0.011	46.9
SSSP	$6.74 \times 10^{-4} d_L^-(v) + 1.66 \times 10^{-4}$	0.054	44.6	$1.30 \times 10^{-4} r(v) + 4.6 \times 10^{-5}$	0.026	45.2

titioners ParMHP. We find that composite partitioner ParMHP saves partitioning time, compared with partitioning via application-driven hybrid partitioners ParHP for each algorithm separately. As shown in Fig. 10(b), the ParMHP partitioners are 109%, 104%, 19% and 111% faster than the ParHP counterparts for HxtraPuLP, HFennel, HGrid and HNE, respectively.

Space efficiency of ParMHP. Fixing $n = 96$ and graph algorithms as {CN, TC, WCC, SSSP, PR}, we tested the space cost of our composite partitions generated by ParME2H and ParMV2H. We find the followings.

(1) Composite partitions save space cost. As opposed to ParHP that stores five different hybrid partitions separately, the composite partitions produced by ParMHP take 55%, 51%, 61% and 67% less space than HxtraPuLP, HFennel, HGrid, and HNE, respectively.

(2) The initial partitions of xtraPuLP, Fennel, Grid and NE are space-efficient, since one partition fits all five applications. The composite partitions of ParMHP take 15%, 17%, 57% and 58% extra space than the initial static partitions for xtraPuLP, Fennel, Grid and NE, respectively. In exchange for the small extra space cost, ParME2H and ParMV2H improve the performance the algorithm batch {CN, TC, WCC, SSSP, PR} by 7.5 times on average, up to 18 times (see Exp-2).

Exp-5: 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, ParV2H, and the composite variants ParME2H and ParMV2H.

(1) For ParE2H and ParV2H, as shown in Fig. 9(l) for CN, we find that (a) they both scale well: when graph G grows, ParE2H (resp. ParV2H) takes from 12.2s to 59.7s (resp. from 5.7s to 32.5s); (b) 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; and (c) the results are consistent for TC, WCC and PR. Note that the point of ParV2H in HNE is missing for $5|G|$ in Fig. 9(l), since NE ran out-of-memory there.

(2) Fixing the batch of algorithms as {CN, TC, WCC, SSSP, PR}, the results of ParME2H and ParMV2H are quite similar to those of ParE2H and ParV2H (not shown). (a) ParME2H and ParMV2H scale well: they take at most 92s and 35s, respectively, when $|G|$ is up to 6.6 billion. (b) Partitioning for all five applications incurs only 22% and 17% extra time than partitioning for CN alone, on average. (c) Skewed input partition leads to longer partitioning time for both ParME2H and ParMV2H. That is, the scalability of ParME2H and ParMV2H is consistent with their non-composite variants.

Exp-6: Learning accuracy and efficiency. Table 5 reports the cost functions $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$, prediction accuracy and training time of each cost model. Note that the coefficients of $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ in Table 5 can be related to system characteristics of our experiment setting, *e.g.,* inter-process message latency and bandwidths. Nonetheless, the metric variables involved in the cost models and important factors are robust to different systems. We adopt MSRE as the accuracy metric. The smaller MSRE of a model is, the more accurate the model is. As shown in Table 5, the MSRE of regression model for CN, WCC, PR and SSSP 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. This optimization deteriorates h_{TC} since node degrees are not informative enough for cost prediction.

Remark. It is worth mentioning that small datasets like liveJournal can be processed directly without partitioning by GPU analytical runtime like Gunrock [49]. In the same settings of Exp-6 and using the provided implementations [1], Gunrock takes 221.4s, 38.9s, 148.4s and 22.3s for applications TC, WCC, SSSP and PR[1] to converge over liveJournal. Datasets Twitter and UKWeb are too large to fit into the 16GB GPU memory, and hence cannot be processed by Gunrock. Observe that

[1] We do not include the result of CN since there exists no official implementation for CN with Gunrock.

the learning cost in Exp-6 is higher than directly running WCC and PR with Gunrock (47.0s and 46.9s vs. 38.9s and 22.3s). This said, we remark the following.

(1) Cost model learning is conducted offline. Once the cost model for a given application is obtained, it can be readily applied over other graphs, small or large. Compared with the performance gains it brings (see Exp-1), the model learning cost is trivial.

(2) Graph partitioning is a must to process large-scale graphs. Our application-driven approach learns a cost model for a given application $\mathcal{A}$ and speeds up the parallel execution of $\mathcal{A}$ over large-scale graphs like Twitter and UKWeb. Without partitioning, Gunrock is hindered by the memory limitation of GPU, and hence fails to handle datasets Twitter and UKWeb.

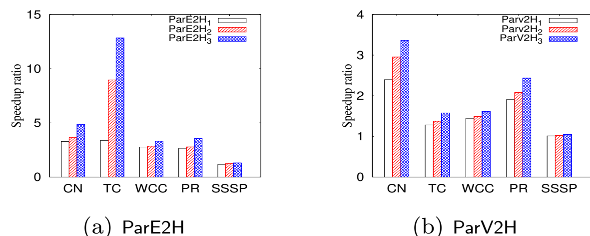
Summary. We find the following. (1) Hybrid partitioner ParE2H (resp. ParV2H) speeds up CN, TC, WCC, PR and SSSP by 11.4, 7.3, 3.0, 3.4 and 1.5 times (resp. 3.6, 2.0, 1.3, 1.8 and 1.2 times) on average, respectively, up to 22.5, 21.2, 7.5, 6.9 and 1.8 times when varying n form 16 to 128. These verify the effectiveness of application-driven partitioners. (2) ParE2H and ParV2H are efficient. In the same setting as (1), on average ParE2H (resp. ParV2H) takes 11.5% (resp. 11.1%) of the total partitioning time of Twitter. (3) The partitioners scale well. On graphs with $500M$ vertices and $6B$ edges, algorithm ParE2H (resp. ParV2H) takes 59.7s (resp. 32.5s) to extends edge-cut (resp. vertex-cut) to a hybrid partition, with 96 workers. (4) The composite extensions of ParE2H and ParV2H are effective. Fixing the batch of algorithms as $\{\text{CN}, \text{TC}, \text{WCC}, \text{SSSP}, \text{PR}\}$, composite partitioners ParME2H and ParMV2H retain comparable performance for each algorithm in the batch, *i.e.,* the gap is less than 3.4% and 8.2%, compared with their performance over partitions of ParE2H and ParV2Hn, respectively. ParME2H (resp. ParMV2H) also speeds up the partitioning process by at least 104% (resp. 19%), while saving up to 55% (resp. 67%) space. (5) The learned functions accurately estimate computational and communication costs. The MSRE of $h_{\mathcal{A}}$ and $g_{\mathcal{A}}$ is below 0.11 for all the algorithms tested. Moreover, the training time is small, *e.g.,* at most 47s for WCC.

8 Conclusion

We have proposed an application-driven partitioning strategy that, as opposed to conventional partitioners, aims to reduce the parallel computation and communication costs of graph algorithms. 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 the cost patterns of $\mathcal{A}$ and speed up parallel execution of $\mathcal{A}$. We have also extended the strategy to support multiple graph algorithms on the same partition at the same time, such that the partition fits the cost patterns of each and every of the algorithms. Our experimental study has verified that application-driven partitioning is effective and efficient.

This work deals with static graph partitioning. One topic for future work is to develop incremental algorithm that maintains application-driven partitions in response to updates to graphs. Another topic is to adapt the application-driven partitioning strategy to multi-core parallelism, a setting in which the communication cost has different characteristics.

Appendix: More experimental study



(a) ParE2H (b) ParV2H

Fig. 11: Phase decomposition of ParE2H and ParV2H

Impact of different phases. We tested 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 speedup gain of the k-th phase of ParE2H by comparing ParE2H$_{k-1}$ and ParE2H$_k$; similarly for ParV2H. Figures 11(a) and 11(b) 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%, 83.5%, 74.4% and 89.2% of the total speedup of CN, TC, WCC, PR and SSSP, respectively. (b) ESplit alone improves CN and TC by 1.1 and 2.7 times, respectively. For WCC, PR and SSSP, its impact is smaller, since CN and TC are more sensitive to workload imbalance. The impact of ESplit on CN over Twitter is smaller, since we filtered large-degree vertices for CN. Without filtering, ESplit improves CN over liveJournal by 1.9 times. (c) MAssign accounts for another 22.3%, 30.1%, 13.8%, 21.9% and 6.3% of the speedup of CN, TC, WCC, PR and SSSP, respectively.

(2) ParV2H. (a) Phase VMigrate contributes the most to the speedup of CN, TC, WCC, PR and SSSP, which account for about 71.2%, 81.2%, 87.1%, 78.2% and 96.7% of the total speedup, respectively. (b) By merging v-cut nodes into e-cut nodes, VMerge contributes 16.5%, 5.8%, 2.6%, 7.1% and 1.2% of the total speedup for the five algorithms tested, respectively. (c) Phase MAssign contributes 9.9% on average.

References

1. Gunrock. *https://github.com/gunrock/gunrock/tree/master/examples*
2. Livejournal. *http://snap.stanford.edu/data/soc-LiveJournal1.html*
3. Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml*
4. Twitter. *http://twitter.com/*
5. UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05*(2006)
6. Graphscope. *https://graphscope.io/* (2020)
7. Andreev, K., Racke, H.: Balanced graph partitioning. TCS **39**(6) (2006)
8. Avdiukhin, D., Pupyrev, S., Yaroslavtsev, G.: Multi-dimensional balanced graph partitioning via projected gradient descent. PVLDB **12**(8), 906–919 (2019)
9. Bang-Jensen, J., Gutin, G.Z.: Digraphs: Theory, Algorithms and Applications. Springer (2008)
10. Bichot, C.E., Siarry, P.: Graph partitioning. John Wiley & Sons (2013)
11. Bishop, C.M.: Pattern recognition and machine learning. springer (2006)
12. Bourse, F., Lelarge, M., Vojnovic, M.: Balanced graph edge partition. In: SIGKDD, pp. 1456–1465 (2014)
13. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. WWW pp. 107–117 (1998)
14. Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent advances in graph partitioning. In: Algorithm Engineering - Selected Results and Surveys, pp. 117–158 (2016)
15. Chandrashekar, G., Sahin, F.: A survey on feature selection methods. Computers & Electrical Engineering **40**(1), 16–28 (2014)
16. Chen, R., Shi, J., Chen, Y., Chen, H.: PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In: EuroSys, pp. 1:1–1:15 (2015)
17. Chvatal, V.: A greedy heuristic for the set-covering problem. Mathematics of operations research **4**(3), 233–235 (1979)
18. Cukierski, W., Hamner, B., Yang, B.: Graph-based features for supervised link prediction. In: INCC, pp. 1237–1244. IEEE (2011)
19. Dai, D., Zhang, W., Chen, Y.: IOGP: An incremental online graph partitioning algorithm for distributed graph databases. In: HPDC, pp. 219–230 (2017)
20. Fan, W., Jin, R., Liu, M., Lu, P., Luo, X., Xu, R., Yin, Q., Yu, W., Zhou, J.: Application driven graph partitioning. In: SIGMOD, pp. 1765–1779. ACM (2020)
21. Fan, W., Liu, M., Lu, P., Yin, Q.: Graph algorithms with partition transparency. IEEE Transactions on Knowledge and Data Engineering (2021)
22. Fan, W., Yu, W., Xu, J., Zhou, J., Luo, X., Yin, Q., Lu, P., Cao, Y., Xu, R.: Parallelizing sequential graph computations. TODS **43**(4), 18:1–18:39 (2018)
23. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company (1979)
24. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: Distributed graph-parallel computation on natural graphs. In: OSDI, pp. 17–30 (2012)
25. Huang, J., Abadi, D.: LEOPARD: Lightweight edge-oriented partitioning and replication for dynamic graphs. PVLDB **9**(7) (2016)
26. Huang, L., Jia, J., Yu, B., gon Chun, B., Maniatis, P., Naik, M.: Predicting execution time of computer programs using sparse polynomial regression. In: NIPS (2010)
27. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. SIAM Journal on Computing **7**(4), 413–423 (1978)
28. Jain, N., Liao, G., Willke, T.L.: Graphbuilder: scalable graph etl framework. Graph Data Management Experiences and Systems (2013)
29. Karypis, G.: Metis and parmetis. In: Encyclopedia of Parallel Computing, pp. 1117–1124 (2011)
30. Karypis, G., Kumar, V.: Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0 (1995)
31. Karypis, G., Kumar, V.: Metis: A software package for partitioning unstructured graphs. Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version **4** (1998)
32. Karypis, G., Kumar, V.: Multilevelk-way Partitioning Scheme for Irregular Graphs. JPDC **48**(1), 96–129 (1998)
33. Kim, M., Candan, K.S.: SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. DKE **72**, 285–303 (2012)
34. Krauthgamer, R., Naor, J., Schwartz, R.: Partitioning graphs into balanced components. In: SODA (2009)
35. Li, D., Zhang, Y., Wang, J., Tan, K.: TopoX: Topology refactorization for efficient graph partitioning and processing. PVLDB **12**(8), 891–905 (2019)
36. Liben-Nowell, D., Kleinberg, J.: The link prediction problem for social networks. CIKM (2003)
37. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: SIGMOD (2010)
38. Margo, D.W., Seltzer, M.I.: A scalable distributed graph partitioner. PVLDB **8**(12), 1478–1489 (2015)
39. Mondal, J., Deshpande, A.: Managing large dynamic graphs efficiently. In: SIGMOD, pp. 145–156 (2012)
40. Newman, M.E., Watts, D.J., Strogatz, S.H.: Random graph models of social networks. Proceedings of the National Academy of Sciences **99**(suppl 1), 2566–2572 (2002)
41. Park, H., Stefanski, L.: Relative-error prediction. Statistics & probability letters **40**(3), 227–236 (1998)
42. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in PyTorch. In: NIPS Autodiff Workshop (2017)
43. Petroni, F., Querzoni, L., Daudjee, K., Kamali, S., Iacoboni, G.: HDRF: Stream-based partitioning for power-law graphs. In: CIKM (2015)
44. Pothen, A., Simon, H.D., Liou, K.P.: Partitioning sparse matrices with eigenvectors of graphs. SIMAX **11**(3), 430–452 (1990)
45. Raz, R., Safra, S.: A sub-constant error-probability low-degree test, and a sub-constant error-probability pcp characterization of np. In: STOC, pp. 475–484 (1997)

46. Slota, G.M., Rajamanickam, S., Madduri, K.: Pulp/xtrapulp: Partitioning tools for extreme-scale graphs. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2017)
47. Tsourakakis, C.E., Gkantsidis, C., Radunovic, B., Vojnovic, M.: FENNEL: Streaming graph partitioning for massive scale graphs. In: WSDM, pp. 333–342 (2014)
48. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
49. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: A high-performance graph processing library on the gpu. In: Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming, pp. 1–12 (2016)
50. Watts, D.J., Strogatz, S.H.: Collective dynamics of ‘small-world’networks. nature **393**(6684), 440 (1998)
51. Wikipedia: Stone–Weierstrass Theorem. *https://en.wikipedia.org/wiki/Stone-Weierstrass_theorem*
52. Yang, S., Yan, X., Zong, B., Khan, A.: Towards effective partition management for large graphs. In: SIGMOD, p. 517 (2012)
53. Zhang, C., Wei, F., Liu, Q., Tang, Z.G., Li, Z.: Graph edge partitioning via neighborhood heuristic. In: KDD (2017)
54. Zhu, X., Chen, W., Zheng, W., Ma, X.: Gemini: A computation-centric distributed graph processing system. In: OSDI, pp. 301–316 (2016)