Incrementalization of Graph Partitioning Algorithms

Wenfei Fan[1,2,3] Muyang Liu[1] Chao Tian[4] Ruiqi Xu[1] Jingren Zhou[4]
[1]University of Edinburgh [2]SICS, Shenzhen University [3]BDBC, Beihang University [4]Alibaba Group
{wenfei@inf., muyang.liu@, ruiqi.xu@}ed.ac.uk, {tianchao.tc, jingren.zhou}@alibaba-inc.com

ABSTRACT

This paper studies incremental graph partitioning. Given a (vertex-cut or edge-cut) partition $\mathcal{C}(G)$ of a graph G and updates ΔG to G, it is to compute changes ΔO to $\mathcal{C}(G)$, yielding a partition of the updated graph such that (a) the new partition is *load-balanced*, (b) its cut size is *minimum*, and (c) the changes ΔO are also *minimum*. We show that this tri-criteria optimization problem is NP-complete, even when ΔG has a constant size. Worse yet, it is *unbounded*, *i.e.,* there exists no algorithm that computes such ΔO with a cost that is determined only by the changes ΔG and ΔO.

We approach this by proposing to incrementalize widely-used graph partitioners $\mathcal{A}$ into *heuristically-bounded incremental* algorithms $\mathcal{A}_\Delta$. Given graph G, updates ΔG to G and a partition $\mathcal{A}(G)$ of G by $\mathcal{A}$, $\mathcal{A}_\Delta$ computes changes ΔO to $\mathcal{A}(G)$ such that (1) applying ΔO to $\mathcal{A}(G)$ produces a new partition of the updated graph although it may not be exactly the one derived by $\mathcal{A}$, (2) it retains the same bounds on balance and cut sizes as $\mathcal{A}$, and (3) ΔO is decided by ΔG alone. We show that we can deduce $\mathcal{A}_\Delta$ from both vertex-cut and edge-cut partitioners $\mathcal{A}$, retaining their bounds. Using real-life and synthetic data, we verify the efficiency and partition quality of our incremental partitioners.

PVLDB Reference Format:
Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. Incrementalization of Graph Partitioning Algorithms. *PVLDB*, 13(8): 1261-1274, 2020.
DOI: https://doi.org/10.14778/3389133.3389142

1. INTRODUCTION

Graph partitioning is to cut a graph into smaller parts of roughly "equal" size, *i.e.,* balanced, while minimizing its cut, *i.e.,* the number of edges crossing (or vertices replicated in) different parts. It is a fundamental problem in graph theory, and is crucial to parallel graph systems for supporting computations on large-scale graphs [18, 22, 23, 33, 45].

A challenge is that real-life graphs are not only large but also frequently updated. An evenly partitioned graph often becomes skewed due to updates [37]. Worse yet, graph partitioning is expensive. It is NP-complete [10, 20, 59] and is hard to approximate [8]. It is often too costly to re-partition the graphs starting from scratch in response to updates.

These highlight the need for incremental partitioning. We compute a partition $\mathcal{C}(G)$ of graph G once with a batch algorithm $\mathcal{C}$ (*a.k.a. partitioner*). Given updates ΔG to G, we compute *minimum* changes ΔO to $\mathcal{C}(G)$ such that $\mathcal{C}(G) \oplus \Delta O = \mathcal{C}(G \oplus \Delta G)$ is a partition of graph $G \oplus \Delta G$ of high quality, *i.e.,* it is *balanced* and has *minimum cut size*. Here $\oplus$ applies changes ΔO to partition $\mathcal{C}(G)$; similarly for $G \oplus \Delta G$. The need is evident because (1) changes ΔG are typically small, *e.g.,* less than 4% increase of articles for English Wikipedia in the year of 2019 [6]; and (2) when ΔG is small, ΔO is often small as well, and is much less costly to compute than to re-partition G.

Challenges. No matter how important, incremental partitioning is hard, for its intractability and unboundedness.

(1) Intractability. Incremental partitioning is a tri-criteria optimization problem, to balance the load, minimize cut sizes and minimize changes ΔO. Here $|\Delta O|$ includes the migration cost to move data among processors and balance their load. We will show that the problem is NP-complete even when the updates ΔG have a constant size, and when we do not require either load balancing or minimum $|\Delta O|$.

(2) Unboundedness. A criterion for measuring the effectiveness of incremental algorithms is boundedness [53, 44, 16]. An incremental partitioner is *bounded* if it computes ΔO such that $\mathcal{C}(G) \oplus \Delta O = \mathcal{C}(G \oplus \Delta G)$, the new partition $\mathcal{C}(G) \oplus \Delta O$ is balanced and has minimum cut size, and moreover, its cost can be expressed as a polynomial function of the size $|\mathsf{CHANGED}|$ of changes, where $|\mathsf{CHANGED}| = |\Delta G| + |\Delta O|$. It ensures that the incremental cost is decided by small $|\mathsf{CHANGED}|$ when ΔG is small, no matter how big G grows.

We show that boundedness is beyond reach for incremental partitioning: no bounded incremental algorithms exist, for either vertex-cut [31] or edge-cut [8] partitioning.

The intractability and unboundedness tell us that it is hard to balance the partition quality and the partitioning cost. While several incremental (*a.k.a.* dynamic) graph partitioners are developed, *e.g.,* [57, 48, 46, 38], none of these offers guarantees on partition quality and efficiency.

Incrementalizing partitioners. To balance the cost and partition quality, we propose to incrementalize *existing* partitioners. A number of batch partitioners are already in place, *e.g.,* [8, 26, 40, 51, 24], which have been verified effective after years of practice and are being widely used. Hence, we

Proceedings of the VLDB Endowment, Vol. 13, No. 8
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3389133.3389142

pick a successful partitioner $\mathcal{A}$ and deduce an incremental algorithm $\mathcal{A}_\Delta$ from $\mathcal{A}$. Given a graph G, updates ΔG to G and a partition $\mathcal{A}(G)$ of G computed by $\mathcal{A}$ (along with auxiliary structures), $\mathcal{A}_\Delta$ computes changes ΔO to $\mathcal{A}(G)$ such that $\mathcal{A}(G) \oplus \Delta O$ is a partition of graph $G \oplus \Delta G$, although *it may not be exactly* $\mathcal{A}(G \oplus \Delta G)$. Moreover, $\mathcal{A}_\Delta$ guarantees both efficiency and partition quality, as follows.

(1) Efficiency. We require that $\mathcal{A}_\Delta$ is *heuristically bounded*. That is, (1) the cost of $\mathcal{A}_\Delta$ is decided only by |CHANGED|, and (2) the size of changes ΔO can be expressed by a polynomial of $|\Delta G|$. This notion is less restrictive than the boundedness of [44, 16] in that $\mathcal{A}_\Delta$ is not required to return exactly the partition $\mathcal{A}(G \oplus \Delta G)$, and that the cut size of its output and the size $|\Delta O|$ are not necessarily minimum. After all, existing partitioners are mostly heuristic anyway.

Intuitively, |CHANGED| indicates the inherent updating cost of incremental partitioning. When $|\Delta G|$ is small, $|\Delta O|$ is also small since it is determined by $|\Delta G|$ alone. Hence $\mathcal{A}_\Delta$ is often more efficient than $\mathcal{A}$ when given small ΔG.

(2) Quality. This said, $\mathcal{A}_\Delta$ also warrants to achieve the same bound on the partition quality as $\mathcal{A}$. Hence, its balance and cut are comparable to those of $\mathcal{A}(G \oplus \Delta G)$ recomputed by partitioner $\mathcal{A}$. If the users of $\mathcal{A}$ are happy with its partition quality, the quality of $\mathcal{A}_\Delta$ is also acceptable to them. This is more practical than to develop an incremental partitioner starting from scratch, since for a newly developed one, its partition quality needs to be verified by years of practice.

Contributions & organization. This paper studies incrementalization of graph partitioners, from theory to practice.

(1) Fundamental results (Section 3). After formalizing the incremental partitioning problem (Section 2), we show that the problem is NP-complete, and the intractability is robust even for rather restricted special cases. Moreover, we show that this problem is unbounded for both vertex-cut and edge-cut; the unboundedness proofs are nontrivial.

(2) Incrementalization (Section 4). We propose to incrementalize (batch) partitioners $\mathcal{A}$ that have been battle-hardened in practice. This yields incremental partitioners $\mathcal{A}_\Delta$ that retain the same bounds on partition quality as $\mathcal{A}$ and are heuristically bounded. We model a class of widely-used partitioners, referred to as iterative partitioners, and provide guidelines for incrementalizing such partitioners.

As proof of concept, we incrementalize popular iterative partitioners, for both vertex-cut and edge-cut.

(3) Incrementalizing vertex-cut (Section 5). Following the guidelines, we incrementalize the distributed neighbor expansion method of [24], denoted as DNE, which is a parallel version of NE [59] and an iterative partitioner for *vertex-cut*. DNE divides edges into disjoint sets and reduces the number of replicated vertices. We incrementalize DNE into IncDNE, and show that IncDNE is heuristically bounded and moreover, retains the same partition quality bound as DNE.

(4) Incrementalizing edge-cut (Section 6). Under *edge-cut*, we pick the distributed k-way greedy graph growing partitioner KGGGP [41]. It partitions the vertices into k sets directly such that the number of crossing edges is small. We show that it can also be incrementalized, denoted by IncKGGGP, guaranteeing the heuristic boundedness and the same partition quality bound as its batch counterpart.

Along the same lines, we have also incrementalized streaming partitioners FENNEL [54] under edge-cut and HDRF [39] under vertex-cut, both heuristically bounded.

(5) Experimental evaluation (Section 7). Using real-life and synthetic graphs, we empirically verify the effectiveness of the incrementalization method. We find the following. (a) IncKGGGP and IncDNE are 5.4 and 7.9 times faster than their parallel batch counterparts, respectively, when $|\Delta G|$ accounts for 10% of $|G|$, and are 1.7 and 3.9 times faster even when $|\Delta G| = 50\%|G|$. (b) IncKGGGP and IncDNE retain comparable partition quality of their batch counterparts: in fact, 20% and 10% better on average, respectively. (c) Our (parallel) incremental partitioners scale well with $|G|$. To partition a graph with 5.8 billion edges, IncKGGGP and IncDNE take 102 and 51 seconds with 128 processors, respectively, in response to 10% updates, while existing parallel incremental partitioners take substantially longer time.

Related work. The related work is categorized as follows.

Graph partitioning. Various algorithms have been developed for edge-cut and vertex-cut partitioning (see [12, 9] for surveys), from *local* methods to *global* algorithms and *multilevel* approaches. Local methods take as input an initial partition of graph G and reduce its cut size iteratively via local search strategies, *e.g.,* KL [29], bubble framework [15], diffusion based method [36], as well as PULP [50] and MLP [55] based on label propagation [42]. In contrast, global methods start with the entire G and compute a partition directly, *e.g.,*exact algorithms [8, 32], spectral partitioning [11, 40], graph growing [21], and vertex-cut partitioners NE [59] and SBV-Cut [31]. Multilevel partitioner METIS [26, 27] splits the process into three phases, to coarsen input graph G, partition the small coarsened graph and map the partition back to G. Hash partitioners offer high efficiency with low memory cost, *e.g.,* 2D-hash [23] and degree based hash [56].

To partition large-scale graphs, several parallel partitioners are developed. ParMETIS [28] is a distributed version of METIS; XtraPuLP [51] extends PULP with distributed-memory parallelism, and Sheep [34] transforms graphs into elimination trees for partitioning via MapReduce operations. For vertex-cut, [43] proposes a parallel local algorithm.

There has also been work on dynamic partitioning. Implemented in ParMETIS, [46] repartitions graphs by diffusion. Using logs, LogGP [57] refines the results via hyper-graph repartitioning. Leopard [25] integrates dynamic partitioning and replication for fault tolerance. Hermes [38] gives a lightweight repartitioner on top of the Neo4j platform. Spinner [35], CatchW [48] and Mizan [30] adapt partitions in the cloud. Another line of work is *streaming* partitioning, in which elements of input graphs arrive in a sequence and are processed on the fly, *e.g.,* HDRF [39] and FENNEL [54].

This work differs from the prior methods in the following.

(1) Instead of developing yet another dynamic partitioner starting from scratch, we propose to incrementalize batch partitioners. We study incremental graph partitioning from theory and methodology (Sections 3 and 4) to practical incrementalization (Sections 5 and 6). None of these has been studied in the previous work, to the best of our knowledge.

(2) Our incremental algorithms retain the same bounds on partition quality as their batch counterparts and are heuristically bounded, while most of the existing dynamic partitioners do not offer any provable performance guarantee.

(3) As opposed to streaming partitioners, the ordering of the updates has no impact on our approach. Furthermore, our incremental partitioners are capable of handling generic updates, *i.e.,* both edge insertions and deletions, while most streaming methods are developed for insertions only.

Incrementalization. There has been work on incrementalizing batch algorithms, *e.g.,* [7, 49, 58, 13]. In particular, [58, 13] incrementalize vertex-centric programs by memorizing messages and reusing the computation of vertex functions; they re-execute the entire program as long as a parameter is changed. In contrast, we incrementalize (parallel) graph partitioners *beyond the vertex-centric model* and aim to minimize recomputation when fragmenting graphs; the techniques of [58, 13] are not applicable in this setting. Moreover, our methods have provable performance guarantees on efficiency, which are not provided by [58, 13].

Related to this work is also [16], which proposes relative boundedness for assessing the quality of exact incremental algorithms. In contrast, we propose heuristic boundedness to cope with the heuristic nature of graph partitioners; we also offer guidelines for incrementalizing partitioners.

2. GRAPH PARTITIONING PROBLEMS

We first state (incremental) graph partitioning problems.

2.1 Graph Partitioning

We consider undirected graphs $G = (V, E)$, where (1) V is a finite set of vertices (nodes); and (2) $E \subseteq V \times V$ is a set of edges. For a subset $V_s \subseteq V$, we denote by $E[V_s]$ the set of edges that have both endpoints in V_s. For a subset $E_s \subseteq E$, we denote by $V[E_s]$ the vertices incident to the edges in E_s. We also write V (resp. E) in G as $V[G]$ (resp. $E[G]$).

Partitions. We consider *edge-cut* and *vertex-cut*, also known as *vertex partition* and *edge partition*, respectively.

(1) Edge-cut [8]. A *k-way edge-cut partition* $\mathcal{C}^k_{\mathcal{E}}(G)$ of graph $G = (V, E)$ is a partition $(V_1, \ldots, V_k)$ of nodes V such that $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$, for $i, j \in [1, k]$, $i \neq j$. We refer to each V_i as a *part* of partition $\mathcal{C}^k_{\mathcal{E}}(G)$.

The *cut-set* of $\mathcal{C}^k_{\mathcal{E}}(G)$ includes *cut edges* e that cross different parts, *i.e.,* $e \notin E[V_i]$ ($i \in [1, k]$). The *cut size* of $\mathcal{C}^k_{\mathcal{E}}(G)$ is $\mathcal{C}^k_{\mathcal{E}}(G).\mathsf{ct} = |E \backslash \bigcup_i E[V_i]|$, *i.e.,* the total number of cut edges. Intuitively, the larger $\mathcal{C}^k_{\mathcal{E}}(G).\mathsf{ct}$ is, the higher the communication cost is incurred to computations on the partition.

(2) Vertex-cut [31]. In contrast to edge-cut, vertex-cut may cut and replicate nodes. A *k-way vertex-cut partition* $\mathcal{C}^k_{\mathcal{V}}(G)$ of G is a partition $(E_1, \ldots, E_k)$ of the edge set E, such that

$$E_1 \cup \cdots \cup E_k = E \text{ and } E_i \cap E_j = \emptyset \text{ if } i \neq j.$$

Each E_i is referred to as a *part* of $\mathcal{C}^k_{\mathcal{V}}(G)$.

The *cut-set* of $\mathcal{C}^k_{\mathcal{V}}(G)$ consists of *cut vertices* v that have adjacent edges in different parts, *i.e.,* there exists $m \geq 1$ such that $v \in V[E_{r_0}] \cap V[E_{r_1}] \cap \cdots \cap V[E_{r_m}]$, where m is the number of *replicas* of v. The *cut size* of $\mathcal{C}^k_{\mathcal{V}}(G)$ is $\mathcal{C}^k_{\mathcal{V}}(G).\mathsf{ct} = \sum_{i \in [1,k]} |V[E_i]| - |V|$, *i.e.,* total number of the replicas.

Load balancing. We say that a k-way edge-cut (resp. vertex-cut) partition $\mathcal{C}^k_{\mathcal{E}}(G)$ (resp. $\mathcal{C}^k_{\mathcal{V}}(G)$) is *$\epsilon$-balanced* if

$$|V_i| \leq \lceil (1+\epsilon)|V|/k \rceil \text{ (resp. } |E_i| \leq \lceil (1+\epsilon)|E|/k \rceil$$

for each $i \in [1, k]$. Here ϵ is called the *balance factor*.

Intuitively, the lower ϵ is, the more balanced the partition is. Unbalanced partition leads to skewed workload and hampers the scalability of parallel graph computation.

Table 1: Notations

Symbol	Notation
$\mathcal{C}^k(G)$, ϵ	k-way partition of graph G and balance factor, resp.
$\mathcal{A}$, $\mathcal{A}(G)$	batch partitioner and a partition it computes, resp.
$\mathcal{A}_\Delta$	incrementalization of $\mathcal{A}$
ΔG	updates to graph G (edge insertions and deletions)
ΔO	updates to the old partition in response to ΔG
f, h	update function and scope function, resp.

We write both $\mathcal{C}^k_{\mathcal{E}}(G)$ and $\mathcal{C}^k_{\mathcal{V}}(G)$ as $\mathcal{C}^k(G)$ if the partition type is clear from context, and further as $\mathcal{C}(G)$ if k is clear. We also use $\mathcal{C}$ to denote a partitioning algorithm.

Graph partitioning. The *edge-cut (resp. vertex-cut) partitioning problem* aims to compute ϵ-balanced edge-cut (resp. vertex-cut) partitions while minimizing the corresponding cut sizes for a given balance factor ϵ.

- *Input*: Graph G, positive integer k and balance factor ϵ.
- *Output*: A k-way edge-cut (or vertex-cut) partition $\mathcal{C}^k(G)$ so that $\mathcal{C}^k(G)$ is ϵ-balanced and $\mathcal{C}^k(G).\mathsf{ct}$ is minimized.

2.2 Incremental Graph Partitioning

We consider *w.l.o.g.* the following *unit updates*:

- edge insertion (insert e), possibly with new nodes, and
- edge deletion (delete e), along with endpoints of degree 0.

That is, a node is removed if all its adjacent edges are deleted; and the adjacent edges of new nodes are inserted. These can catch modification if extended to property graphs.

A *batch update* ΔG to G is a sequence of unit updates.

The *incremental partitioning* problem is stated as follows.

- *Input*: Graph G, old k-way edge-cut (or vertex-cut) partition $\mathcal{C}(G)$ that is produced by a batch partitioner $\mathcal{C}$, balance factor ϵ, and a batch update ΔG to G.
- *Output*: Updates ΔO to the old partition of G such that $\mathcal{C}(G \oplus \Delta G) = \mathcal{C}(G) \oplus \Delta O$, and in addition,
  (1) new partition $\mathcal{C}(G \oplus \Delta G)$ is ϵ-balanced;
  (2) $\mathcal{C}(G \oplus \Delta G).\mathsf{ct}$ is minimized, *i.e.,* $\mathcal{C}(G \oplus \Delta G).\mathsf{ct} \leq \mathcal{C}'(G \oplus \Delta G).\mathsf{ct}$ for other ϵ-balanced $\mathcal{C}'(G \oplus \Delta G)$; and
  (3) the size of ΔO is minimized, *i.e.,* $|\Delta O| \leq |\Delta O'|$ for any updates $\Delta O'$ to $\mathcal{C}(G)$ under conditions (1) and (2).

Intuitively, $|\Delta O|$ contains the *migration* cost of incrementally partitioning G, for moving data among processors.

This is a tri-criteria optimization problem. Its objective is to ensure partition quality (conditions (1) and (2)) and reduce cost (condition (3)) when computing new partitions.

The notations of this paper are summarized in Table 1.

3. FUNDAMENTAL PROBLEMS

We next investigate the complexity and boundedness of the incremental partitioning problem. The results are mostly negative, telling us that the problem is challenging.

(1) Complexity. We first settle its conventional complexity. Its decision problem, also referred to as incremental partitioning, is to decide, given graph G, partition $\mathcal{C}(G)$, updates ΔG, balance factor ϵ, a bound η for cut sizes and another bound λ on migration cost, whether there exists a partition $\mathcal{C}(G \oplus \Delta G) = \mathcal{C}(G) \oplus \Delta O$ such that (1) $\mathcal{C}(G \oplus \Delta G)$ is ϵ-balanced, (2) $\mathcal{C}(G \oplus \Delta G).\mathsf{ct} \leq \eta$, and (3) $|\Delta O| \leq \lambda$.

It is known that both edge-cut partitioning and vertex-cut partitioning are NP-complete [10, 20, 59]. We show that their incremental counterparts are also intractable.

To understand what factor dominates the complexity, we drop one of the three conditions and investigate the problem. We find that the intractability is robust: the problem remains NP-hard as long as the bound on cut sizes is imposed, when the constraint on either balance or migration cost is not required, even when ΔG have a constant size.

Theorem 1: *For both edge-cut and vertex-cut,*
(1) incremental graph partitioning is NP*-complete and remains* NP*-hard even when (a) $|\Delta G|$ is a constant, and (b) either ϵ or λ is ∞,* i.e., *condition (1) or (3) is dropped; and (2) it is in* PTIME *when η is ∞ (without condition (2)).* □

Proof: (1) We give an NP algorithm for incremental edge-cut (resp. vertex-cut) partitioning that works as follows: (i) guess a set ΔO of updates to the old partition $\mathcal{C}(G)$; and (ii) check whether the three conditions of incremental partitioning are satisfied by ΔO and $\mathcal{C}(G)$. The algorithm is in NP since the checking in step (ii) can be done in PTIME.

For both edge-cut and vertex-cut, the NP-hardness of incremental partitioning with $\lambda = \infty$ (resp. $\epsilon = \infty$) is verified by reduction from the decision problem of graph partitioning stated in Section 2.1 [20, 59] (resp. the CLIQUE problem [19], which decides whether there is a clique of size K or more). Each reduction uses a unit edge insertion with two new nodes as ΔG. When reducing from CLIQUE, we analyze the impact on cut sizes if all nodes (resp. edges) of the K-clique are put into the same part of the edge-cut (resp. vertex-cut) partition, from which the one-to-one mapping is established.

(2) When $\eta = \infty$, we repeatedly move one vertex (resp. edge) from the part having the maximum elements to a minimum one until the edge-cut (resp. vertex-cut) partition is ϵ-balanced ("yes") or λ changes are made ("no"), in PTIME. □

(2) Boundedness. As suggested by [44], the effectiveness of incremental algorithms can be evaluated by boundedness. An incremental partitioner $\mathcal{A}_\Delta$ is *bounded* if it can compute updates ΔO such that $\mathcal{C}(G \oplus \Delta G) = \mathcal{C}(G) \oplus \Delta O$ and the new partition satisfies the three conditions stated above, and moreover, its cost can be expressed as a function of $|\mathsf{CHANGED}|$, where $|\mathsf{CHANGED}| = |\Delta G| + |\Delta O|$. The incremental graph partitioning problem is *bounded* if there exists such a bounded $\mathcal{A}_\Delta$, and is called *unbounded* otherwise.

Following [44], we consider *locally persistent* algorithms $\mathcal{A}_\Delta$, in which each node maintains a block of storage including pointers to its neighbors. It starts from an update in ΔG and checks G following the pointers, where the choice of which one to follow depends only on the information accumulated in the current step. It does not allow global information such as pointers to nodes other than neighbors.

Unfortunately, no bounded incremental partitioner $\mathcal{A}_\Delta$ exists at all for either vertex-cut or edge-cut.

Theorem 2: *For both edge-cut and vertex-cut, the incremental graph partitioning problem is unbounded, even under a constant number of unit updates.* □

Proof: We give an elementary proof for each. (1) We first show that incremental partitioning is unbounded for 2-way edge-cut partitions, *i.e.,* bisections under a constant number of *either unit edge insertions or deletions.* We construct an instance of the edge-cut partitioning problem, where $\epsilon = 0$ and graph G has a bisection of cut size 0. Then we prove by contradiction that there exists no bounded incremental algorithm that can derive a new bisection $\mathcal{C}(G \oplus \Delta G)$ of $G \oplus \Delta G$ satisfying the three constraints in response to constant-size updates ΔG. (2) Similarly, we construct an instance of 2-way vertex-cut partitioning problem, and show that any new partition satisfying the three conditions cannot be computed by any bounded incremental vertex-cut partitioner. The result holds when ΔG consists of edge insertions only or edge deletions only, even if ΔG is of constant size. □

4. INCREMENTALIZING PARTITIONERS

In light of Theorems 1 and 2, we propose to incrementalize widely-used graph partitioners $\mathcal{A}$. The objective is to both (1) ensure small migration cost, and (2) retain the partition quality of $\mathcal{A}$, which has been verified by years of practice.

We formalize incrementalization (Section 4.1) and present guidelines for incrementalizing partitioners (Section 4.2).

4.1 Incrementalization of Batch Algorithms

Pick a batch partitioner $\mathcal{A}$ that has proven effective in practice. An *incrementalization of* $\mathcal{A}$, denoted by $\mathcal{A}_\Delta$, is an incremental partitioner that (1) is heuristically bounded, and (2) retains the partition quality of $\mathcal{A}$, defined as follows.

(1) Heuristically bounded. Given a graph G, updates ΔG to G, a balance factor ϵ, the old partition $\mathcal{A}(G)$ produced by a run of $\mathcal{A}$ on G with possibly auxiliary structures $D_\mathcal{A}$ of $\mathcal{A}$, a *heuristically bounded* incrementalization $\mathcal{A}_\Delta$ computes changes ΔO to $\mathcal{A}(G)$ such that $\mathcal{A}(G) \oplus \Delta O$ is a partition of graph $G \oplus \Delta G$, its cost can be expressed as a polynomial function in $|\mathsf{CHANGED}| = |\Delta G| + |\Delta O|$, and moreover, the size $|\Delta O|$ can be expressed as a polynomial of $|\Delta G|$.

As opposed to the boundedness adopted in Theorem 2, $\mathcal{A}_\Delta$ is not required to be exact, *i.e.,* it does not necessarily return $\mathcal{A}(G \oplus \Delta G)$. After all, almost all practical partitioners are non-deterministic heuristics. Hence we only require $\mathcal{A}_\Delta$ to return a partition of $G \oplus \Delta G$, which has to retain the quality of $\mathcal{A}$ (see below). As will be seen in Sections 5 and 6, this relaxed notion allows us to deduce partitioners that incur only the cost necessary for incremental partitioning.

(2) Relative bound on partition quality. It is required that the new partition $\mathcal{A}(G) \oplus \Delta O$ is also balanced *w.r.t.* the same balance factor as $\mathcal{A}(G)$, and moreover, that $\mathcal{A}_\Delta$ retains the same bound on the cut sizes as $\mathcal{A}$.

As opposed to Theorem 2, $\mathcal{A}_\Delta$ does not have to minimize the cut sizes. Instead, it guarantees partition quality "as good as" that of $\mathcal{A}$. Hence if $\mathcal{A}$ is widely used in practice, then the quality of $\mathcal{A}_\Delta$ is also acceptable to the users of $\mathcal{A}$.

4.2 An Incrementalization Approach

Given a batch partitioner $\mathcal{A}$, how could we deduce its heuristically bounded incrementalization $\mathcal{A}_\Delta$? Below we first characterize a class of iterative partitioners $\mathcal{A}$ that are commonly used in practice. We then provide general guidelines for incrementalizing such partitioners.

Iterative partitioners. Consider a batch partitioner $\mathcal{A}$. When running $\mathcal{A}$ on graph G, $\mathcal{A}$ often builds auxiliary structures $D_\mathcal{A}$, which extends G with status variables associated with the nodes and edges in G, and part sizes of partition $\mathcal{A}(G)$. Structure $D_\mathcal{A}$ keeps track of the computation and often evolves during the entire process. Hence we characterize the partitioning process of $\mathcal{A}$ as *continuous updates* to $D_\mathcal{A}$.

In practice, the continuous updates are often carried out in iterations by chronological orders. Denote by $D^t_\mathcal{A}$ the status

of $D_{\mathcal{A}}$ after t rounds of iterations. Then $D_{\mathcal{A}}^{t+1}$ is obtained from $D_{\mathcal{A}}^{t}$ by applying the changes computed with update function f on certain update region H_t:

$$D_{\mathcal{A}}^{t+1} = D_{\mathcal{A}}^{t} \oplus f(H_t),$$
$$H_t = h(D_{\mathcal{A}}^{t}).$$

As will be seen shortly, the *update function* f is decided by the logic of $\mathcal{A}$. The update region H_t for iteration $t+1$ consists of certain status variables in $D_{\mathcal{A}}$, which are determined by a *scope function* h of $\mathcal{A}$, taking the latest status $D_{\mathcal{A}}^{t}$ as input. The computation proceeds until it reaches a fixpoint, *i.e.,* $D_{\mathcal{A}}^{t_0+1} = D_{\mathcal{A}}^{t_0}$ at some iteration t_0. At this point $D_{\mathcal{A}}^{t_0}$ subsumes the resulting partition $\mathcal{A}(G)$.

We say that $\mathcal{A}$ is an *iterative partitioner* if its workflow complies with the iterative computation model given above.

Intuitively, the scope function h identifies update region by gathering information from the last iteration, *i.e.,* status variables of $D_{\mathcal{A}}$ to be updated. The update function f deduces actual changes to those status variables, *e.g.,* deciding the part allocation for unassigned nodes and edges. That is, the computation in an iterative partitioner $\mathcal{A}$ is guided by the changes at runtime (change propagation).

Example 1: Consider the graph growing [21] that produces 2-way edge-cut partitions. It selects a starting node from a graph and grows one part around it via breadth-first search (BFS), until half of the nodes are included. The remaining are put into the other part. For graph G of Fig. 1 (consider solid edges only), graph growing can pick w_0' as the starting node and put $v_1, \ldots, v_{19}$, $u_1', \ldots, u_{19}'$, w_0' and w_0 to part V_1, while the rest of nodes in G are allocated to part V_2.

Graph growing is an iterative partitioner. Its status variables include the part id given to each node and part sizes. In each iteration, its update function f takes as input a set of unassigned vertices, a candidate part id and the size of that part. It assigns the part id to these vertices and updates the size accordingly. The scope function h returns unallocated neighbors of the nodes that are newly assigned, *i.e.,* BFS. □

As will be seen in the later sections, the operations in update function f and scope function h can be conducted in parallel, *e.g.,* allocating different vertices or edges at the same time within a single parallel iteration.

One can verify that the following are iterative partitioners: (1) graph growing [21], greedy growing [26], KGGGP [41] and bubble methods [15] for edge-cut; (2) NE [59] and DNE [24] for vertex-cut; and (3) FENNEL [54], HDRF [39], Ginger [14] and Greedy [52] for the streaming setting; while the streaming partitioners are developed for insertions only, they can still be incrementalized to handle generic updates.

However, edge-cut partitioner METIS [26, 27] and vertex-cut Sheep [34] are not in this class since they mutate the topological structure of graphs during partitioning, beyond the expressive power of the iterative computation model.

Approach. To deduce an efficient incrementalization $\mathcal{A}_\Delta$ from an iterative partitioner $\mathcal{A}$, we identify *essential* changes *pertaining to* updates ΔG, and apply such changes by using (revised) update and scope functions of $\mathcal{A}$, as follows.

(a) Resuming iteration. Given updates ΔG and the final status $D_{\mathcal{A}}^{T}$ after a batch run of $\mathcal{A}$, $\mathcal{A}_\Delta$ first finds changes to a (small) region covered by ΔG, and enforces them on $D_{\mathcal{A}}^{T}$ to get the new status $D_{\mathcal{A}}^{T+1}$, *i.e.,* the partitioning process is *resumed* with a new iteration. These are called *essential changes pertaining to* updates. More specifically, the essential changes $f(H_T)$ at the initial new iteration $T+1$ is computed by the update function f of $\mathcal{A}$, where $H_T \subseteq \Delta G$ is an update region determined by a scope function h' revised from function h of $\mathcal{A}$. The essential changes are applied to the status variables in H_T as in the batch partitioner $\mathcal{A}$.

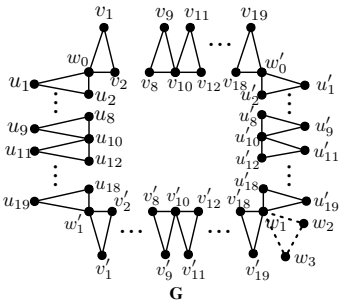


Figure 1: Example graph

Based on the new status $D_{\mathcal{A}}^{T+1}$, the original scope function h of $\mathcal{A}$ then identifies another update region H_{T+1} for the next new iteration $T+2$, such that the essential changes pertaining to H_{T+1} can be determined as above. The update function f is then applied to H_{T+1}. The two steps iterate in $\mathcal{A}_\Delta$ until no more changes can be made with functions f and h along the same lines as that in the batch $\mathcal{A}$, yielding the new final status $D_{\mathcal{A}}^{T'}$ for this incremental run.

(b) Rebalancing. The input updates ΔG may make some parts of the old edge-cut (resp. vertex-cut) partition $\mathcal{A}(G)$ become overweight. When this happens, $\mathcal{A}_\Delta$ needs to remove a set U_i of nodes (resp. edges) from each overweight V_i (resp. E_i) to make it satisfy the balance constraint, and then *reallocates* U_i using the same strategy as that in (a) above. That is, the union of such sets U_i's are also *treated as updates* in $\mathcal{A}_\Delta$ at the beginning and are combined with the input updates ΔG to compute essential changes.

In fact, $\mathcal{A}_\Delta$ reassigns some vertices or edges that are either covered by the input updates or picked for rebalancing. As verified in our experiments (see Section 7), this helps refine the old partition since $\mathcal{A}_\Delta$ can make use of *all the information of the old partition.* In some cases, this yields partition quality even better than repartitioning with the batch $\mathcal{A}$.

Example 2: Continuing with Example 1, consider a batch update ΔG that deletes four edges (w_0, v_1), (w_0, v_2), (w_1, v_{18}') and (w_1, v_{19}') from graph G.

Here the incrementalization $\mathcal{A}_\Delta$ of graph growing finds the initial new update region directly using the revised scope function h', which finds the set of nodes covered by ΔG, *i.e.,* $\{w_0, v_1, v_2, w_1, v_{18}', v_{19}'\}$; $\mathcal{A}_\Delta$ makes them unassigned, updates the sizes of the corresponding parts, and deduces the candidate part id's based on the allocation of their neighbors. It next reallocates these nodes iteratively using the original update and scope functions of graph growing (Example 1), putting w_0 (resp. w_1) into part V_2 (resp. V_1), and keeping the allocation of all other nodes unchanged. That is, $\mathcal{A}_\Delta$ swaps the original allocation of just two nodes. □

Justification. The key idea of the incrementalization $\mathcal{A}_\Delta$ is to adopt *the original semantics of* $\mathcal{A}$ by means of continuous updates to $D_{\mathcal{A}}$, and to restrain changes to the status variables related to input updates and load balancing only.

It ensures heuristic boundedness when (1) the update function f and the (revised) scope functions h and h' are *incrementally computable* in polynomial time in the size of

updates. That is, $f(d \oplus \Delta d)$ (resp. $h(d \oplus \Delta d)$, $h'(d \oplus \Delta d)$) can be computed from $f(d)$ (resp. $h(d)$, $h'(d)$) and Δd without accessing the entire d; and (2) the cost for identifying each set U_i from overweight parts is a polynomial in $|U_i|$.

Intuitively, some aggregate functions, *e.g.,* sum and avg, are incrementally computable, but the function for computing eigenvectors in spectral partitioning method [40] is not.

Indeed, $\mathcal{A}_\Delta$ only re-evaluates update functions on those regions with changes involved, and the size of changes is bounded by a polynomial in $|\Delta G|$ and $\sum_{i \in [1,k]} |U_i|$ when the functions are incrementally computable. One can verify that the new balance constraint is satisfied after rebalancing if $|U_i| \geq |\Delta G|$ for $i \in [1, k]$; and $\sum_{i \in [1,k]} |U_i|$ can be bounded by $O(|\Delta G|)$. Note that rebalancing is conducted once when $\mathcal{A}_\Delta$ starts. Thus both the overall cost of $\mathcal{A}_\Delta$ and the size $|\Delta O|$ are decided by $|\Delta G|$ *alone* when the two conditions above hold, *i.e.,* $\mathcal{A}_\Delta$ is heuristically bounded. Moreover, if $\mathcal{A}$ achieves balance *w.r.t.* a given factor ϵ, then $\mathcal{A}_\Delta$ ensures balance by means of the same update and scope functions, since the balance constraint is checked in these functions.

We should remark that there exist other classifications for graph partitioning algorithms. We study incremetalization of the iterative partitioners since they cover quite a few widely-used methods and are formulated using simple update and scope functions, which can be extended to express other graph algorithms beyond partitioners.

As proof of concept, we next incrementalize vertex-cut and edge-cut partitioners in Sections 5 and 6, respectively. It is shown there that the incrementalization $\mathcal{A}_\Delta$ is able to retain the same bound on cut sizes as that of $\mathcal{A}$.

5. INCREMENTALIZING VERTEX-CUT

Following the guidelines of Section 4.2, we incrementalize the distributed neighbor expansion (DNE) [24] for vertex-cut. DNE parallelizes sequential partitioner NE [59], known for its high partition quality. We review DNE in Section 5.1.

The main result of this section is the following. We will incrementalize the iterative partitioner DNE and provide a constructive proof of Theorem 3 in Section 5.2.

Theorem 3: *There exists a heuristically bounded incrementalization of the vertex-cut partitioner* DNE *that retains the same bound on partition quality as* DNE. □

5.1 DNE: A Batch Vertex-Cut Partitioner

As vertex-cut partitioners, DNE and NE aim to minimize the total number of replicas of cut vertices while keeping the k-way edge partition $(E_1, \ldots, E_k)$ of a graph G balanced.

NE creates each E_i iteratively following a neighbor expansion heuristic. Each time it selects a node v from the set of boundary vertices of E_i (or randomly if $E_i = \emptyset$), such that v has *minimal* unassigned adjacent edges. Here the *boundary vertices* include those nodes in $V[E_i]$ that have unassigned adjacent edges. It (a) allocates the remaining adjacent edges of v to E_i, and (b) puts each (u, w) into E_i if (v, u) is assigned in (a) and $w \in V[E_i]$. The set E_i is expanded in this way until violating the balance constraint, followed by building E_{i+1}. Observe that NE minimizes replicas in (a); and the allocation of edges within two hops of boundary vertices in (b) does not introduce any new replicas.

To scale with large-scale graphs, the parallel version DNE [24] of NE performs the expansion of each edge set *in parallel*, using a greedy heuristic similar to that of NE. It adopts the generic shared-nothing model. Assume that k processors $P_1, \ldots, P_k$ are available and are connected by a network of channels. Initially the graph G is distributed among k processors via a *random* vertex-cut partitioner. Algorithm DNE computes a k-way vertex-cut partition for G, *i.e.,* the number of parts in the partition is *fixed* as the number of processors used. It employs two kinds of distributed processes: *expansion process* and *allocation process*. The selection of boundary vertex is done in expansion processes in a local-optimal way as in NE. *i.e.,* checking minimal unassigned adjacent edges. The allocation processes detect the one-hop and two-hop neighbor edges of the selected vertices, as well as the corresponding new boundary vertices, which will be sent to expansion processes for actual allocation.

The computation of DNE starts with k expansion processes and k allocation processes; each one is deployed to one of the processors distinctly. It expands edge sets iteratively in synchronized rounds, in which each expansion process begins with a randomly selected vertex and is responsible for maintaining the boundary vertices and the edges of one part of the partition, and for checking the balance constraint. Different from NE, DNE can select multiple boundary vertices during each iteration, *e.g.,* top-K minimal vertices for a fixed K. DNE allocates the edges following the same semantics as NE, except that CAS (compare-and-swap) operation is performed to resolve conflicts when different threads attempt to allocate the same edge to different parts.

Example 3: Recall graph G from Example 1 and assume that $k = 4$, balance factor $\epsilon = 0.001$, and v_{10}, v'_{10}, u_{10} and u'_{10} are initially picked as the boundary vertices for four parts E_1 to E_4 by the expansion processes of DNE, respectively. Suppose that DNE selects top-2 minimal boundary vertices.

In the first iteration, the allocation process for E_1 detects adjacent edges of v_{10} and creates a candidate part id 1 for them. It also finds two-hop neighbor edges (v_8, v_9) and (v_{11}, v_{12}) as assigning them to E_1 does not increase the cut size. Moreover, v_8 and v_{12} will be selected as new boundary vertices of E_1 since they both have 2 unassigned adjacent edges, which are local-minimal. All these are sent to the expansion process for E_1 to do the actual allocation and start the next iteration. This is done in parallel with the expansions of E_2, E_3 and E_4, which include one-hop and two-hop neighbor edges of v'_{10}, u_{10} and u'_{10} in relevant processes.

DNE ends up with four parts, *i.e.,* edge sets $E_i = \{(x_j, x_{j+1}) \mid j \in [1, 18]\} \cup \{(x_{2j}, x_{2(j+1)}) \mid j \in [1, 8]\} \cup \{(y, x_1), (y, x_2), (y', x_{18}), (y', x_{19})\}$ of graph G. Here we use variables $\langle x, y, y' \rangle$ to represent $\langle v, w_0, w'_0 \rangle$, $\langle v', w'_1, w_1 \rangle$, $\langle u, w_0, w'_1 \rangle$ and $\langle u', w'_0, w_1 \rangle$ for each $i \in [1, 4]$, respectively. □

5.2 Incrementalization of DNE

As a proof of Theorem 3, we next develop a heuristically bounded incrementaliztion IncDNE for DNE. The challenges include the following: (1) the old partitions derived by the batch DNE can become skewed due to batch updates, and (2) different unit updates may have impacts on each other and hence change the course of the computation of DNE.

Overview. To retain the same bound on partition quality as DNE, we apply essential changes pertaining to the updates as suggested by Section 4.2. As shown there, the idea is to *identify and reuse* the update function and scope function of the iterative algorithm DNE.

Update function. Recall that DNE generates partitions by updating the status in terms of part assignments for edges, the number of unassigned adjacent edges of each boundary vertex, the boundary vertices for each part, and part sizes (for checking balance) in parallel supersteps, *i.e.,* iterations. Thus it fits into the paradigm of iterative computation.

DNE has an update function f to revise the status variables above, which takes as input a set of unassigned edges with their candidate part id's and corresponding part sizes, and a set of new boundary vertices for different parts. It updates the allocation of these edges using their candidate part id's, adjusts the part sizes according to the latest allocation, and updates the number of unallocated adjacent edges for each new boundary vertex v, referred to as the *score* $\mathsf{sc}(v)$ of v. Based on the newly updated scores, it also selects the minimal boundary vertices for the next iteration.

In fact, all the work of update function f is done by the expansion processes of DNE in each synchronized round.

Scope function. The scope function h of DNE identifies unallocated one-hop and two-hop neighbor edges of each newly selected boundary vertex v, and derives a candidate part id for them from the parts in which v resides. These edges will be finally assigned by update function. If an edge (u, w) is detected here with a candidate part id i and if node u has not been covered by the edge set E_i, scope function h also marks u as a new boundary vertex for part E_i. All these are conducted in parallel in the allocation processes of DNE.

Auxiliary structures. We next present auxiliary structures used by IncDNE. We keep the following. (1) The assignment $\mathsf{part}(e)$ for each edge e. (2) For each node v, a set $\mathsf{S}(v)$ to store the id's of the parts that cover v together with the number of v's adjacent edges assigned to these parts. (3) The final size $W(i)$ of each part E_i for $i \in [1, k]$. All the structures are obtained as byproducts when running DNE.

Incrementalization. The incrementalization IncDNE is shown in Algorithm 1. Following the guidelines of Section 4.2, it discovers an initial new update region to resume the iteration of the batch run and achieve load balancing. This is done by a function revised from the scope function of DNE, using a strategy to identify edges from overweight parts and ΔG. Starting from this region, it applies the original update and scope functions of DNE to allocate edges.

More specifically, a designated coordinator (processor) P_c collects from all processors the statistics of old assignments at the beginning of IncDNE (line 1), *i.e.,* how many edges at processor P_i were allocated to part of id j for $i, j \in [1, k]$. Using this information, P_c deduces a list $(r_1, \ldots, r_k)$ of numbers for each overweight part of id j, such that *canceling* the allocation of r_i many edges to part E_j at processor P_i can make this part satisfy the new balance constraint (lines 2-3). The numbers are combined with the updated part sizes, and sent to corresponding processors via messages (lines 4-5). Coordinator P_c also distributes the updates ΔG, in which the update of edge (u, v) is posted to one of the processors that contain the old node u or v (line 6). Thereafter, procedure LocalAdjE is executed at each processor to identify new update region and continue allocation (line 7).

Procedure LocalAdjE. Upon receiving messages from coordinator, LocalAdjE at each processor P_i first adjusts the part sizes and local graph accordingly (lines 1-2). Then it *randomly* selects certain number of edges, whose old allocation need to be canceled as guided by the messages, for rebalancing. These edges are collected into a set $\mathcal{E}^\Delta$ and their part assignments are marked $\perp$, denoting the status of unallocated (lines 4-6). Moreover, the assignments for some edges having endpoints in the local update ΔG_i are assigned $\perp$ (line 6), which will also be reallocated regarding the new status. These edges are filtered by procedure FilterE (not shown), which returns edge e if an endpoint of e has a small number of adjacent edges assigned to part E_j and $\mathsf{part}(e) = j$. Intuitively, reallocating such edges may help improve the partition quality. It next computes score value $\mathsf{sc}(v)$ and adjusts the set $\mathsf{S}(v)$ for each old node in $\mathcal{E}^\Delta$ and ΔG_i (lines 7-8). By the definitions of sc and S, this can be performed *incrementally* by checking $\mathcal{E}^\Delta \cup \Delta G_i$ and the filtered edges only. All these constitute the operations of the revised scope function and rebalancing.

Algorithm 1: IncDNE

Input: Graph G fragmented by vertex-cut, integer k, balance factor ϵ, auxiliary information and batch update ΔG.
Output: Updated auxiliary information, including $\mathsf{part}(\cdot)$.

// executed at the coordinator P_c
1 collect the statistics of the old allocation;
2 **foreach** overweight part of id j **do**
3 select a list $(r_1, \ldots, r_k)$ of non-negative integers *s.t.* $\sum_i r_i \geq W(j) - (1+\epsilon)|E[G \oplus \Delta G]/k|$;
4 $W(j) := W(j) - \sum_i r_i$; post $W(j)$ to all processors;
5 **foreach** $i \in [1, k]$ *s.t.* $r_i > 0$ **do** post $\langle r_i, j \rangle$ to processor P_i;
6 distribute ΔG to corresponding processors;
7 invoke procedure LocalAdjE at each processor;

Procedure LocalAdjE // executed at each P_i, in parallel
Input: Local auxiliary information, messages from coordinator.
Output: Updated local auxiliary information.
1 update the part sizes according to $W(j)$'s received from P_c;
2 apply the update ΔG_i received from P_c on the local graph;
3 $\mathcal{E}^\Delta := \emptyset$;
4 **foreach** pair $\langle r_i, j \rangle$ received from P_c **do**
5 collect r_i edges that are allocated to part of id j into $\mathcal{E}^\Delta$;
6 **foreach** edge $e \in \mathcal{E}^\Delta \cup \mathsf{FilterE}(V[\Delta G_i])$ **do** $\mathsf{part}(e) := \perp$;
7 collect into $\mathcal{L}^\Delta$ the old nodes that are covered by $\mathcal{E}^\Delta \cup \Delta G_i$;
8 **foreach** node $v \in \mathcal{L}^\Delta$ **do**
compute score $\mathsf{sc}(v)$, update $\mathsf{S}(v)$;
9 **if** $\mathcal{L}^\Delta \neq \emptyset$ **or** there are newly inserted edges **then**
10 launch corresponding expansion processes and allocation processes using $\mathcal{L}^\Delta$ as initial new boundary vertices;

LocalAdjE then determines the assignments for unallocated edges by employing the update function (expansion processes) and scope function (allocation processes) of DNE, where nodes in $\mathcal{L}_\Delta = \mathcal{E}^\Delta \cup \Delta G_i$ are treated as initial new boundary vertices and are given to update function (lines 9-10, see details of the processes in Section 5.1 and [24]). That is, the iteration resumes with the new update region $\mathcal{L}_\Delta$.

Example 4: Continuing with Example 3, consider a batch update ΔG that deletes all edges incident to w_0, v_2, v_3, u_2, u_3, and inserts (w_1, w_2), (w_1, w_3) and (w_2, w_3). Now parts E_2 and E_4 become overweight and hence IncDNE selects two sets $U_2 = \{(w_1', v_2'), (w_1', v_1')\}$ and $U_4 = \{(w_0', u_1'), (w_0', u_2')\}$ of edges and cancels their original allocation at corresponding processors. Moreover, it computes the scores for w_0', u_1', u_2', w_1', v_1', v_2' and w_1; based on the results, it removes the part id of 4 (resp. 2) from $\mathsf{S}(w_0')$ (resp. $\mathsf{S}(w_1')$) since they now have no adjacent edges assigned to E_4 (resp. E_2).

IncDNE next allocates the newly inserted edges and the edges in $U_3 \cup U_4$ using the same update and scope functions

of the batch DNE. Here the old nodes w_0', u_1', u_2', w_1', v_1', v_2' and w_1 are taken as new boundary vertices. For instance, edge (w_0', u_1') is allocated to E_1 in the first new iteration, since w_0' is a minimal boundary vertex of E_1 with $\mathsf{sc}(v) = 2$ and part E_4 cannot take any more edge due to the balance constraint. Finally $U_2 \cup \{(w_2, w_3)\}$ is included in E_3, and $U_4 \cup \{(w_1, w_2), (w_1, w_3)\}$ is added to E_1 in IncDNE. □

IncDNE provides a constructive proof for Theorem 3.

Proof of Theorem 3: Algorithm IncDNE adjusts the status $\mathsf{part}(\cdot)$, $\mathsf{S}(\cdot)$ and $W(\cdot)$ related to new update region, via original update and scope functions of DNE. Based on their semantics, one can verify that these functions are incrementally computable (Section 4.2) with the auxiliary structures. In fact, apart from newly inserted edges and those selected for reallocation, the part assignments of all other edges remain stable and they trigger no change to $\mathsf{S}(\cdot)$ or $W(\cdot)$.

Moreover, the cost for rebalancing and identifying initial new update region can be expressed as a polynomial in $|\Delta G|$, since (1) IncDNE selects edges from each overweight part randomly for assignment cancellation, and the total number of such edges is bounded by a polynomial in $|\Delta G|$; and (2) procedure FilterE only inspects edges adjacent to the nodes covered by ΔG to cancel part assignments. Putting these together, IncDNE is heuristically bounded.

For the cut sizes, note that IncDNE resumes the iterative computation of DNE; hence we treat those uncanceled allocation by DNE as if being decided in the same rounds of IncDNE. We check the total number of vertices in all parts to analyze the cut. This number is bounded by a function as in [24, 59] (to be given below). Denote by E_r^t (resp. S_i^t) the set of unassigned edges (resp. nodes covered by E_i) at the end of round t in DNE and IncDNE. Initially $S_i^0{=}\emptyset$ and E_r^0 includes all edges of the graph. The function is given as $\phi_t{=}|E_r^t|{+}\sum_i |S_i^t|$. Denote by c_t the number of boundary vertices randomly selected at round t.

One can verify that $\phi_t - \phi_{t-1} \leq c_t$ for $t \geq 1$ since (a) each edge assignment introduces at most one replica of cut vertex except for those randomly selected ones. Moreover, (b) there are at most $|V| + k$ random selections of boundary vertices, since each node can be randomly selected as the boundary vertex at most once when it has unassigned adjacent edges. Observe that IncDNE adopts the original semantics of DNE when allocating edges, and arguments (a) and (b) hold for both the two methods. Indeed, IncDNE always puts (u, v) into a part that is the same as the assignment for one of u's or v's allocated adjacent edges if exist; hence reallocation of old edges in IncDNE does not affect the upped bound for the number of random selections. Thus $\phi_T \leq \phi_0 + \sum_{t\in[1,T]} c_t$ when DNE and IncDNE terminate at round T. That is, $\sum_i |S_i^T| - |V[G \oplus \Delta G]| \leq |E[G \oplus \Delta G]| + k$ when partitioning the updated graph by DNE and IncDNE, *i.e.,* the two achieve the same upper bound on cut sizes and the bound coincides with the one given in [24] for DNE. □

6. INCREMENTALIZING EDGE-CUT

We next incrementalize the k-way greedy graph growing method KGGGP [41] under edge-cut. KGGGP can be used to partition the coarsened graphs in ParMETIS [28] and METIS [26], two widely-used vertex partitioners for decades.

Below we first review parallel KGGGP (Section 6.1). We then show that KGGGP can be incrementalized, and prove the following performance bounds (Section 6.2).

Theorem 4: *There exists a heuristically bounded incrementalization of the edge-cut partitioner* KGGGP *that retains the same bound on partition quality as* KGGGP. □

6.1 KGGGP: A Batch Edge-Cut Partitioner

KGGGP generates k-way edge-cut partitions $(V_1, \ldots, V_k)$ of an input graph G via greedy best-first search, which picks the most promising vertices for each part V_i by certain rules. As an extension of the greedy graph growing method [26] for bi-partitioning, it computes the k-way partitions directly. KGGGP starts with a randomly selected *seed* vertex v_r and expands each part V_i iteratively by adding *boundary vertices* in the frontier of the best-first search from v_r. The inclusion of a vertex v in V_i is guided by a greedy score function

$$\mathsf{sc}(v, i) = \sum_{v' \in V_i, (v,v') \in E} |v'| - v.\mathsf{unalloc},$$

where v.unalloc denotes the number of unallocated neighbors of v. Each time a boundary vertex v with the *maximal* $\mathsf{sc}(v, i)$ in the frontier is allocated to V_i as long as the balance constraint is not violated. This is followed by continuing the search at v and updating the scores for the neighbors of v. Function $\mathsf{sc}(\cdot, \cdot)$ is essentially a heuristic that selects vertices with the largest decrease of the cut sizes.

KGGGP can be easily parallelized in a way similar to how DNE parallelizes NE (Section 5.1). That is, different parts are expanded *in parallel rounds* via distinct greedy best-first searches; and multiple boundary vertices with top-K maximal scores can be added to the same part in each synchronized round. We consider parallel KGGGP in the sequel.

The parallel partitioner works on a graph G that is initially fragmented across k processors $(P_1, \ldots, P_k)$ via a *random* edge-cut partitioner, where each P_i maintains a fragment, *i.e.,* a subgraph including all vertices allocated to P_i and their adjacent edges. It computes a k-way edge-cut partition for G. The number of parts of the partition is again *fixed* as the number of processors. Similar to DNE, it adopts *expansion process* and *allocation process*. Here (1) each expansion process manages the frontier of a best-first search to select top-K maximal boundary vertices for expanding one part, where a node is picked at random as the initial boundary vertex (seed of the search); and (2) allocation processes update the scores, perform the best-first searches in a distributed manner and generate potential allocation requests for boundary nodes, which are transmitted to expansion processes for allocation. If multiple best-first searches touch the same vertex v, it is selected as the boundary vertex and allocated to only one part in accordance with the maximum score associated with v.

There are k expansion processes and k allocation processes launched in KGGGP, each one for a distinct processor. They work as the update function and scope function of the iterative partitioner KGGGP (see details below).

Example 5: Consider graph G and balance factor ϵ of Example 3. Let $k = 4$ and v_{10}, v_{10}', u_{10} and u_{10}' be the seed vertices chosen by KGGGP to expand parts V_1 to V_4, respectively. The seeds are firstly assigned to distinct parts by expansion processes, as they are the only boundary vertices. After this, the neighbors v_8, v_9, v_{11} and v_{12} of v_{10} are detected as new boundary vertices for V_1 in allocation processes, *i.e.,* best-first search for V_1 is continued at v_{10}. In addition, they trigger allocation requests for v_9 and v_{11} to V_1 if KGGGP finds top-2 maximal boundary vertices. Note that the latest scores $\mathsf{sc}(v_9, 1) = \mathsf{sc}(v_{11}, 1) = 0$ are now top-2

maximal. Similarly, allocation requests are generated for boundary vertices v'_9 and v'_{11} (resp. u_9 and u_{11}, u'_9 and u'_{11}) to V_2 (resp. V_3, V_4) simultaneously in the same synchronized round. All these will be assigned in the second iteration.

KGGGP terminates with a 4-way vertex partition of $V_1 = \{v_1, \ldots, v_{19}\} \cup \{w_0\}$, $V_2 = \{v'_1, \ldots, v'_{19}\} \cup \{w_1\}$, $V_3 = \{u_1, \ldots, u_{19}\} \cup \{w'_1\}$ and $V_4 = \{u'_1, \ldots, u'_{19}\} \cup \{w'_0\}$. □

6.2 Incrementalization of KGGGP

We next present a heuristically bounded incrementalization IncKGGGP of the iterative algorithm KGGGP and prove Theorem 4. Along the same lines as Section 5.2, we start with the update function and scope function of KGGGP along with the auxiliary structures adopted.

Update function. Since KGGGP creates partitions by updating the status (*i.e.,* scores, part assignments and part sizes) in parallel rounds, it can be modeled as an iterative partitioner. Given a set of allocation requests, *i.e.,* boundary nodes with candidate part id's and scores, and the part sizes, the update function f of KGGGP applies the assignments for those nodes with top-K maximal scores from the requests and updates part sizes in response to the allocation.

In fact, expansion processes of KGGGP serve the purpose of the update function. Each of the k expansion processes is launched to expand one part V_i, as shown in Algorithm 2. It maintains a set $\mathcal{B}_i$ of boundary vertices for V_i with their scores, and adjusts $\mathcal{B}_i$ according to the allocation requests received from allocation processes in each iteration (lines 1-3), *i.e.,* adding or removing vertices, or updating scores. The boundary vertices with top-K maximal scores are selected from $\mathcal{B}_i$ and are allocated to part of id i if $\mathcal{B}_i$ is nonempty; otherwise an unassigned vertex is randomly picked for allocation (lines 4-6). It also updates the part size $W(i)$ and synchronizes the new allocation with allocation processes by broadcasting (lines 6-7). The termination of the process is decided by whether part V_i is fully filled.

Scope function. Based on the newly allocated vertices, the scope function h of KGGGP adjusts the score values of their neighbors, and initiates allocation requests for the unassigned neighbors, *i.e.,* new boundary vertices.

The allocation processes in KGGGP carry out the functionality of h. Each allocation process (shown in Algorithm 2) first updates the part assignments upon receiving the new allocation (line 1). It then inspects the neighbors of newly assigned ones, updates their scores and generates allocation requests (lines 3-6). Each request consists of an unassigned neighbor v', the maximal score $\max_j \mathsf{sc}(v', j)$ for v' and a part id j_m from $\arg\max_j \mathsf{sc}(v', j)$; this will be sent to the expansion process related to V_{j_m} (line 7). If there has been another request created for v' with a different part id j'_m, the allocation process also notifies the expansion process *w.r.t.* $V_{j'_m}$ to remove v' from the boundary vertices maintained there. Therefore, v' can be assigned to a single part only.

Auxiliary structures. We maintain the following. (1) For each vertex v, $\mathsf{part}(v)$ to track its assigned part. (2) The score value $\mathsf{sc}(v, i)$ at the end of the batch partitioning process. (3) The final part size $W(j)$ ($j \in [1, k]$). All these are readily obtained when running KGGGP.

Incrementalization. The incrementalization IncKGGGP of KGGGP is outlined in Algorithm 2, which finds essential changes to status concerning updates ΔG and load balancing.

Algorithm 2: IncKGGGP

Input: Graph G fragmented by edge-cut, integer k, balance factor ϵ, auxiliary information and batch update ΔG.
Output: Updated auxiliary information, including $\mathsf{part}(\cdot)$.

// executed at the coordinator P_c
1 $\mathcal{I} := \mathsf{ReBalance}(\Delta G)$;
2 distribute $\mathcal{I}$ and ΔG to corresponding processors;
3 invoke procedure LocalAdjV at each processor;

Procedure LocalAdjV // executed at each P_i, in parallel
Input: Local auxiliary information, messages M from P_c.
Output: Updated local auxiliary information.
1 update $W(\cdot)$ and $\mathsf{part}(\cdot)$ based on M;
2 adjust local fragment with update ΔG_i received from P_c;
3 **foreach** node v in $\mathsf{FilterV}(V[\Delta G_i])$ **do** $\mathsf{part}(v) := \perp$;
4 $\mathcal{L}_\Delta := \{v \mid \mathsf{part}(v) = \perp \wedge \exists (v, v')\ s.t.\ \mathsf{part}(v') \neq \perp\}$;
5 **foreach** node $v \in \mathcal{L}_\Delta$ **do**
6 update $\mathsf{sc}(v, \cdot)$; create allocation request for v;
7 collect into R all the new allocation requests generated;
8 **if** there are unassigned nodes **then**
9 launch corresponding expansion processes and allocation processes using R as initial new allocation requests;

Expansion Process // launched for each part V_i, in parallel
1 set $\mathcal{B}_i := \emptyset$;
2 **while** part of id i can take in vertices **and** there are unassigned nodes **do**
3 $\mathcal{B}_i$.adjust(RecevieAllocationRequest());
4 **if** $\mathcal{B}_i \neq \emptyset$ **then** $T_{max} := \mathcal{B}_i$.DeleteTopKMax();
5 **else** $T_{max} :=$ RandomVertex();
6 Allocate(T_{max}, i); update $W(i)$;
7 SynchronizeAllocation(T_{max});

Allocation Process // launched at each P_i, in parallel
1 $T_{recv} :=$ RecevieAllocation(); update $\mathsf{part}(\cdot)$ based on T_{recv};
2 $\mathcal{B}_{send} := \emptyset$;
3 **foreach** edge (v, v') with v in T_{recv} **do**
4 update $\mathsf{sc}(v', \cdot)$;
5 **if** v' is not assigned **then**
6 $\mathcal{B}_{send} := \mathcal{B}_{send} \cup \mathsf{Request}(v', \arg\max_j \mathsf{sc}(v', j))$;
7 SendAllocationRequest($\mathcal{B}_{\mathsf{send}}$);

Similar to IncDNE, IncKGGGP has two phases. (1) It first cancels a set of old assignments to satisfy the new balance constraint and extracts an initial new update region related to ΔG. (2) It then applies the original update and scope functions of KGGGP to resume the iterative computation.

The rebalancing phase is performed in a way analogous to that in IncDNE, in which coordinator P_c invokes procedure ReBalance (line 1, not shown) to compute the number of assignments to be canceled at each processor and the updated part sizes. These cancellation requests and input updates ΔG are posted to different processors (line 2), followed by executing procedure LocalAdjV in parallel (line 3).

Procedure LocalAdjV adjusts the old allocation, part sizes and local graph in response to the messages received from P_c (lines 1-2). Depending on the cancellation requests, the vertices v's are also selected *randomly* and each $\mathsf{part}(v)$ is assigned the status $\perp$. In addition, it computes a set of nodes covered by the input updates using FilterV, and cancels their old allocation (line 3). A vertex v is returned by FilterV if it was allocated to part of j and $\mathsf{sc}(v, j)$ is relatively small. Note that such old allocation introduces a large number of cut edges and are badly needed to be re-inspected. LocalAdjV next updates the scores and creates allocation requests for those unassigned vertices $\mathcal{L}_\Delta$ that are connected to allocated ones (lines 4-6). Here the scores are adjusted *incrementally w.r.t.* the edges that are either involved in the input updates or covered by nodes with unassigned status.

Table 2: Real-world graphs

Dataset	Abbr.	Type	$\|V\|$	$\|E\|$
Road-CA[4]	RCA	road network	1.97M	2.77M
Wiki-de[5]	WK	hyperlink, dynamic	2.17M	86.3M
LiveJ.[3]	LJ	social network	4.84M	68.5M
FriendSter[2]	FS	social network	65.6M	1.80B
UK-Web[1]	UK	hyperlink	1.06B	3.72B

Finally the update function and scope function of KGGGP are executed to continue the iteration, where nodes $\mathcal{L}_\Delta$ are picked as the restarting points of best-first searches (line 9).

Example 6: Recall the graph G, balance factor ϵ and vertex partition $(V_1, \ldots, V_4)$ from Example 5, and the updates ΔG from Example 4. IncKGGGP first finds that parts V_2 and V_4 are overweight. It then cancels the allocation for two sets U_2=$\{v_1'\}$ and $U_4 = \{w_0'\}$ of vertices due to rebalancing. Since w_0', v_1', w_2 and w_3 are the only nodes that remain unassigned and are linked to assigned noes, IncKGGGP adjusts the scores associated with these four nodes and generates new allocation requests for them. For example, a request for putting w_0' into V_1 is created as $\mathsf{sc}(w_0', 1) = 2$ and is maximum for w_0'. Similarly, it generates another request for allocating v_1' to V_3. Note that V_2 and V_4 cannot take in any more nodes after the rebalancing. Based on the new status, IncKGGGP only adds $\{w_0', w_2\}$ to V_1 and $\{v_1', w_3\}$ to V_3. □

Proof of Theorem 4: The heuristic boundedness of partitioner IncKGGGP can be verified along the same lines as its counterpart for IncDNE given in the proof of Theorem 3.

For the cut sizes, recall that each vertex is assigned in the batch KGGGP when it is touched by the best-first searches as a *non-starting node* or is taken randomly as the *starting node* of a new search. A detailed analysis reveals that the cut sizes of the partitions created by KGGGP is dominated by the number N_s of starting nodes; intuitively, all the adjacent edges of a starting node can be cut edges, while each non-starting node is incident to *at least one* non-cut edge.

To see the number N_s of starting nodes in the partitions produced by IncKGGGP, note that it adopts the same random selection strategy of starting nodes as that in KGGGP, even for the nodes chosen for reallocation in the rebalancing phase and procedure FilterV. Indeed, the best-first searches of KGGGP in the batch run are resumed in IncKGGGP without adding any extra starting nodes for reasons other than that in KGGGP. That is, each node with a canceled assignment will be reallocated to one of the parts assigned for its neighbors by IncKGGGP; and v is a starting node selected by IncKGGGP if and only if it can be chosen as a starting node in KGGGP. Consequently, the two partitioners have the same upper bound on the number N_s of starting nodes and thus achieve the same upper bound on cut sizes. □

7. EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we conducted experiments to evaluate the (1) efficiency, (2) scalability, (3) partition quality of our incrementalized partitioners; and (4) their effectiveness on running distributed graph applications.

Experimental setting. Five real-life graphs were used and are summarized in Table 2. Among them, WK is a dynamic graph whose edges are labeled by timestamps to record when they were inserted or deleted. RCA is a road network with a large diameter of 865, while the others are skewed graphs following power-law degree distribution.

Table 3: Baseline partitioners

Method	Model	Parallel	Incremental
KGGGP [41]	edge-cut	yes	no
DNE [24]	vertex-cut	yes	no
FENNEL [54]	edge-cut	no	no
HDRF [39]	vertex-cut	no	no
ParMETIS [47]	edge-cut	yes	yes
Hermes [38]	edge-cut	yes	yes

We also developed a generator to produce synthetic graphs G, controlled by the number of vertices $|V|$ (up to 250 million) and the number of edges $|E|$ (up to 6 billion).

Updates. The updates to WK were extracted from the real timestamped changes by limiting certain periods of time. For other graphs, we generated random updates controlled by the size $|\Delta G|$, and they were mixed with equal amount of edge insertions and deletions, unless stated otherwise.

Partitioners. In addition to IncDNE and IncKGGGP, following the same guidelines, we also incrementalized stream partitioners HDRF [39] and FENNEL [54] into IncHDRF and IncFENNEL, respectively. They are heuristically bounded, but not parallel. Table 3 lists the baselines that we compared with. We used the official package for ParMETIS, and implemented the rest of methods in C++ with openMPI.

We evaluated an incrementalization $\mathcal{A}_\Delta$ of a batch partitioner $\mathcal{A}$ as follows. Given graph G, updates ΔG and partition $\mathcal{A}(G)$, we compared the efficiency and quality of $\mathcal{A}_\Delta$ in computing the new partitions based on $\mathcal{A}(G)$ against their batch counterparts obtained by running $\mathcal{A}$ on updated graph $G \oplus \Delta G$ starting from scratch. Since $|G| \approx |G \oplus \Delta G|$ for mixed updates ΔG, the pre-partitioning time for G is approximately the same as for partitioning $G \oplus \Delta G$ using batch algorithms; the latter is reported. For a fair comparison, ParMETIS and Hermes incrementally process partitions of G created by KGGGP in response to updates ΔG.

We conducted all experiments on an HPC cluster. For each test, we used up to 24 servers, each with 16 processors of 2.20 GHz and 64 GB memory, up to 384 processors. Each experiment was repeated 5 times. The average is reported.

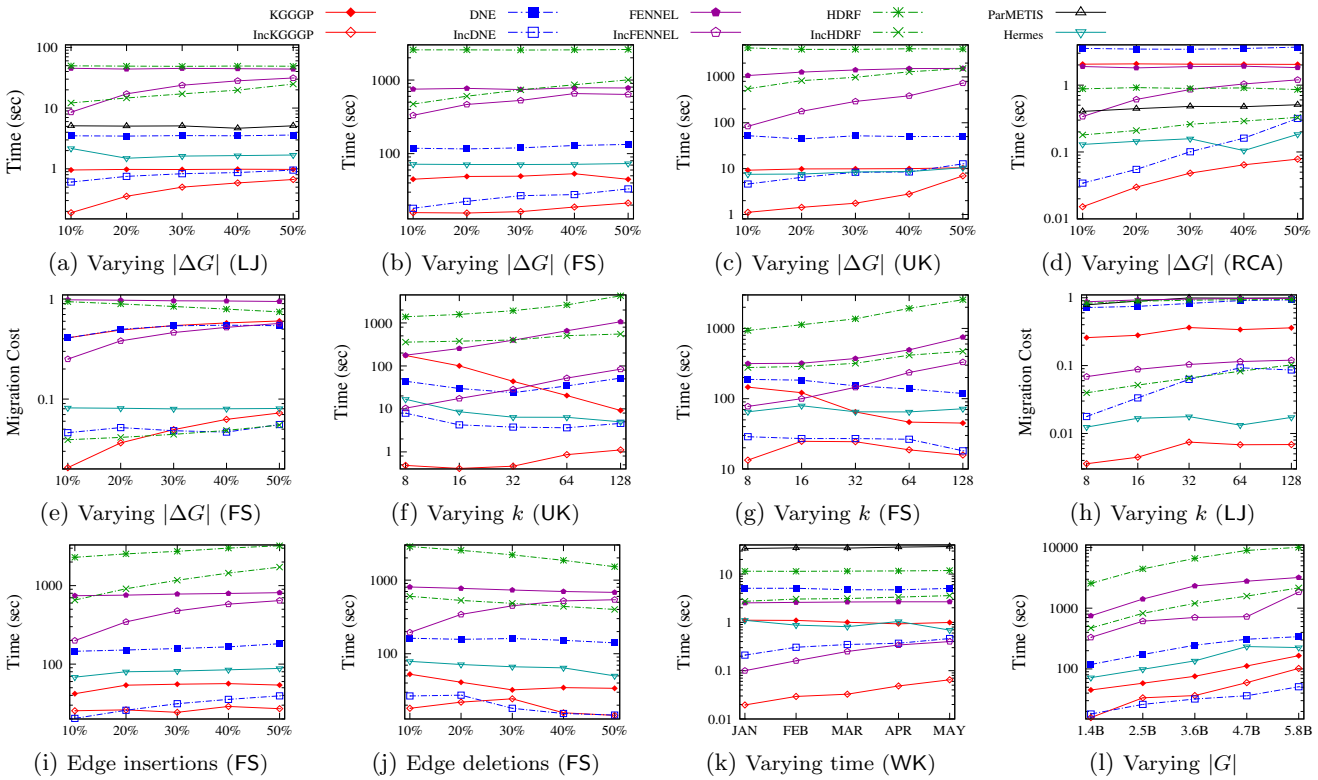
Experimental results. We next report our findings. We found that ParMETIS failed to handle large FS and UK. We ensure that all output partitions are ϵ-balanced (ϵ=0.1) and report partitioning time excluding that for loading inputs.

Exp-1: Efficiency. We first evaluated the efficiency of (incremental) partitioners *w.r.t.* update size $|\Delta G|$, parts (processors) k and update type. We also report *normalized migration cost*, *i.e.,* the ratio of the number of nodes (resp. edges) that switch parts to the number of overlapped nodes $|V[G] \cap V[G \oplus \Delta G]|$ (resp. edges $|E[G] \cap E[G \oplus \Delta G]|$), which indicates the cost of re-distributing G due to ΔG.

(1) Varying $|\Delta G|$. Fixing $k = 128$ for LJ, FS and UK and $k = 8$ for (small) RCA, we varied $|\Delta G|$ from 10% to 50% of $|G|$. The results in Figures 2(a) to 2(e) tell us the following.

(a) IncKGGGP, IncDNE, IncFENNEL and IncHDRF consistently outperform their (parallel) batch counterparts even when $|\Delta G|$ is up to 50% of $|G|$. Over LJ, FS and UK, on average they are 5.4, 7.9, 6.8 and 5.8 times faster when $|\Delta G|$ accounts for 10% of $|G|$, respectively, and are 1.7, 3.9, 1.6 and 2.4 times faster even when $|\Delta G| = 50\%|G|$.

(b) The four heuristically bounded incremental partitioners take less time on smaller updates. In contrast, ParMETIS and Hermes are not sensitive to $|\Delta G|$.



(a) Varying $|\Delta G|$ (LJ) (b) Varying $|\Delta G|$ (FS) (c) Varying $|\Delta G|$ (UK) (d) Varying $|\Delta G|$ (RCA)

(e) Varying $|\Delta G|$ (FS) (f) Varying k (UK) (g) Varying k (FS) (h) Varying k (LJ)

(i) Edge insertions (FS) (j) Edge deletions (FS) (k) Varying time (WK) (l) Varying $|G|$

Figure 2: Elapsed time and migration costs

(c) IncKGGGP and IncDNE consistently outperform other methods when $|\Delta G| \leq 30\%|G|$. IncKGGGP (resp. IncDNE) is 27 and 11 (resp. 8.4 and 3.6) times faster than ParMETIS and Hermes, respectively, on LJ with 10% updates.

(d) Our heuristically bounded incremental partitioners also work well on non-skewed RCA. As shown in Figure 2(d), they are at least 1.5 times faster than their batch counterparts when $|\Delta G|$ is up to 50% of $|G|$.

(e) The incremental partitioners incur small migration costs. As shown in Figure 2(e), IncKGGGP, IncDNE, IncFENNEL and IncHDRF incur such costs less than 21%, 4.6%, 25% and 3.9% on FS with $10\%|G|$ updates, respectively. In contrast, these are 41%, 41%, 98% and 94% by batch counterparts.

These justify the need for incremental partitioners.

(2) Varying k. Fixing $|\Delta G| = 10\%|G|$, we varied k from 8 to 128, to test the efficiency with different numbers of parts. We find the following from Figures 2(f) to 2(h).

(a) All sequential partitioners take longer with larger k, due to the increased workloads. However, the response time of parallel partitioners, *e.g.,* IncKGGGP, may decrease as k gets larger, since k is also the number of the processors used.

(b) Over FS and UK, IncKGGGP, IncDNE, IncFENNEL and IncDNE consistently outperform their batch counterparts. They are efficient: on UK with k=8, they take 0.5, 7.9, 10.3 and 357 seconds, respectively. IncKGGGP is particularly fast on UK(with loading time of 21.5s) as it reuses 98% of batch partitioning result on vertices. These nodes cover 98% edges in $G \oplus \Delta G$ as UK is a heavily skewed graph. IncKGGGP is faster than Hermes by at least 3.7 times; and it incurs only 4.5% of the migration cost of ParMETIS on LJ (Fig. 2(h)).

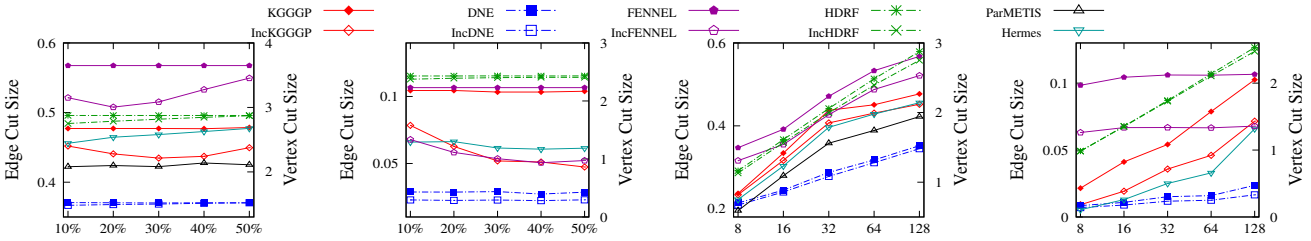
(3) Type of Updates. Apart from the random mixed updates adopted above, we also studied the impact of different types of updates. The results are shown in Figures 2(i) to 2(k).

(a) Fixing $k = 128$, we generated random updates of edge insertions only and edge deletions only over FS, respectively, with $|\Delta G|$ varying from 10% to 50% of $|G|$. We find that (a) the response time of incremental partitioners increases with the size of edge insertions, as expected; (b) not all incremental partitioners take longer when given more edge deletions, since the sizes of updated graphs may shrink substantially with large amount of deletions; and (c) compared to batch ones, our incrementalization achieves speedup of at least 1.26 times for both 50% insertions and deletions,

(b) Fixing k=8, we extracted real-life updates for WK by inspecting its status over 5 months in 2011 (Figure 2(k)). The result is consistent with the results on randomly generated updates: all our incremental partitioners are substantially faster than their batch counterparts; IncKGGGP (resp. IncDNE) is at least 11 (resp. 1.5) times faster than Hermes.

Exp-2: Scalability. We evaluated the scalability of our incremental partitioners with larger synthetic graphs. Fixing $|\Delta G|$=10%$|G|$ and k=128, we ran partitioners on 5 synthetic graphs by varying $|G|=|V|+|E|$ from 1.4 billion to 5.8 billion. As shown in Figure 2(l), (a) our incremental partitioners scale well with $|G|$. When $|G|$ is increased by 4.1 times, using the same amount of computing resources, KGGGP, IncKGGGP, DNE and IncDNE take 3.7, 6.4, 2.9 and 2.8 times longer. (b) When $|G|$ is 5.8 billion, IncKGGGP and IncDNE take only 102 and 51 seconds, to update partitions, respectively. In contrast, Hermes, IncFENNEL and IncHDRF take 225, 1839 and 2162 seconds, respectively, while ParMETIS failed to complete. (c) Heuristically bounded incremental partitioners outperform their batch counterparts by up to 8.5 times; these are consistent with the results in Exp-1.

Exp-3: Quality. We also tested the quality of partitions computed. Since we have ensured all partitions to be ϵ-



(a) Varying $|\Delta G|$ (LJ) (b) Varying $|\Delta G|$ (UK) (c) Varying k (LJ) (d) Varying k (UK)

Figure 3: Normalized cut sizes

Table 4: Running distributed graph applications

Method	SSSP		WCC		PageRank	
	Time	Comm.	Time	Comm.	Time	Comm.
KGGGP	80.8	18.6	33.4	21.9	127	108
IncKGGGP	63.4	15.9	31.0	19.9	105	107
FENNEL	72.5	11.8	36.6	21.1	135	87.6
IncFENNEL	66.4	8.86	31.5	20.1	113	82.2
Hermes	79.2	14.5	61.9	20.9	145	86.9
DNE	102	20.8	16.6	12.3	43.7	112
IncDNE	92.8	21.2	13.4	11.7	39.4	109
HDRF	216	50.8	20.1	21.0	47.8	188
IncHDRF	208	49.6	18.7	19.9	42.2	179

balanced ($\epsilon = 0.1$), we only compared the normalized cut sizes for different edge-cut (resp. vertex-cut) partitioners, *i.e.,* $\mathcal{C}^k_{\mathcal{E}}(G).\mathsf{ct}/|E|$ (resp. $\mathcal{C}^k_{\mathcal{V}}(G).\mathsf{ct}/|V|$).

(1) Varying $|\Delta G|$. In the same setting as Exp-1(1), Figures 3(a) and 3(b) report the normalized cut sizes on LJ and UK, respectively. We find that (a) compared to batch ones, our incremental partitioners compute partitions with comparable or even lower cut sizes for reasons given in Section 4.2. Over LJ and UK, the cut sizes of IncKGGGP, IncDNE, IncFENNEL and IncHDRF are on average 26%, 16%, 27% and 2% lower than their batch counterparts. (b) The edge cut sizes of IncKGGGP are on average 10% and 5.3% lower than Hermes on UK and LJ, respectively. (c) While the edge cut sizes of ParMETIS are on average 12% lower than those of IncKGGGP on LJ, ParMETIS cannot handle large graphs like UK. These experimentally verify Theorems 3 and 4, *i.e.,* the efficiency of heuristically bounded incrementalization does not come at a price of lower partition quality.

(2) Varying k. In the same setting as Exp-1(2), we evaluated the impact of the number k of parts on partition quality. As shown in Figures 3(c) and 3(d), (a) for all partitioners, their cut sizes increase with larger k, as expected. (b) The cut sizes of IncKGGGP, IncDNE, IncFENNEL and IncHDRF are on average 80%, 89%, 81% and 98% of their batch counterparts. That is, our incremental partitioners retain the batch partition quality. (c) Over LJ, the cut sizes of IncKGGGP are on average 2.4% and 13% higher than those of Hermes and ParMETIS, respectively, but IncKGGGP is substantially faster, and ParMETIS cannot handle large graphs.

Exp-4: Effectiveness on running distributed applications. Finally, we studied the impact of different partitioning results on the performance of three common distributed graph applications, including Single Source Shortest Path (SSSP), Weakly Connected Components (WCC) and PageRank. We used the graph-centric programs implemented on GRAPE [18, 17] under both vertex-cut and edge-cut models. The partitions were computed for FS, with $|\Delta G| = 10\%|G|$ and $k = 128$. ParMETIS again failed to complete. The elapsed time (in second) and communication cost (in GB) of the applications are reported in Table 4. It shows that (a) for all three applications, the elapsed time over our incremental partitions is at least 4% less than their batch counterparts. (b) The communication cost on our incremental partitions is at least 5.8% (resp. 1.2%) lower for WCC (resp. PageRank), and at most 1.8% higher for SSSP. (c) Although Hermes partition incurs less communication than IncKGGGP partition, its elapsed time is longer. These verify that our incrementalized partitioners retain the partition quality of the batch ones, consistent with Exp-3.

Summary. We find that our heuristically bounded incremental partitioners substantially outperform their batch counterparts in efficiency, and at the same time, achieve comparable or even better partition quality. (1) IncKGGGP, IncDNE, IncFENNEL and IncHDRF are 5.4, 7.9, 6.8 and 5.8 times faster than the batch ones when $|\Delta G| = 10\%|G|$, respectively, and are 1.7, 3.9, 1.6 and 2.4 times faster even when $|\Delta G|$ is $50\%|G|$. Moreover, the smaller the updates are, the faster they run. These verify the effectiveness of heuristic boundedness. (2) On average, IncKGGGP, IncDNE, IncFENNEL and IncHDRF incur cut sizes that are 20%, 10%, 22% and 2% lower than their batch counterparts, respectively. They also reduce at least 4% of runtime when running distributed applications. (3) Parallel IncKGGGP and IncDNE scale well with k, $|\Delta G|$ and $|G|$. It takes IncKGGGP and IncDNE only 102 and 51 seconds, respectively, to partition a graph with 5.8 billion edges in response to 10% updates, using 128 processors. (4) IncKGGGP and IncDNE outperform existing incremental ParMETIS and Hermes in efficiency with comparable cut sizes. Over LJ, when $k = 128$, IncKGGGP (resp. IncDNE) is on average 13 and 4.9 (resp. 6.4 and 2.2) times faster than ParMETIS and Hermes, respectively.

8. CONCLUSION

While incremental graph partitioning is intractable and unbounded, we have proposed an approach to develop practical incremental partitioners. The novelty of the work consists of a new approach to incrementalizing partitioners, and a notion of heuristic boundedness to characterize the effectiveness of incremental heuristic algorithms. We have verified the effectiveness of the new approach by incrementalizing (parallel) batch partitioners, both edge-cut and vertex-cut, and by conducing an experimental study.

One topic for future work is to generalize the incrementalization approach to graph algorithms beyond partitioners.

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

9. REFERENCES

[1] UKWeb. *http://law.di.unimi.it/webdata/uk-2007-05/*, 2011.

[2] Friendster. *https://snap.stanford.edu/data/com-Friendster.html*, 2012.

[3] LiveJournal. *http://snap.stanford.edu/data/com-LiveJournal.html*, 2012.

[4] Road-ca. *http://snap.stanford.edu/data/roadNet-CA.html*, 2012.

[5] Wiki-de. *http://konect.uni-koblenz.de/networks/link-dynamic-dewiki*, 2012.

[6] Size of Wikipedia. *https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia*, 2020.

[7] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, CMU, 2005.

[8] K. Andreev and H. Racke. Balanced graph partitioning. *Theory Comput. Syst.*, 39(6):929–939, 2006.

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

[10] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *KDD*, 2014.

[11] T. N. Bui and C. Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *PPSC*, 1993.

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

[13] Z. Cai, D. Logothetis, and G. Siganos. Facilitating real-time graph mining. In *CloudDB*, 2012.

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

[15] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26(12):1555–1581, 2000.

[16] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.

[17] W. Fan, M. Liu, R. Xu, L. Hou, D. Li, and Z. Meng. Think sequential, run parallel. *LNCS*, 11180:1–25, 2018.

[18] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *SIGMOD*, 2017.

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

[20] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some simplified NP-complete graph problems. *Theor. Comput. Sci.*, 1(3):237–267, 1976.

[21] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981.

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

[23] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[24] M. Hanai, T. Suzumura, W. J. Tan, E. S. Liu, G. Theodoropoulos, and W. Cai. Distributed edge partitioning for trillion-edge graphs. *PVLDB*, 12(13):2379–2392, 2019.

[25] J. Huang and D. Abadi. LEOPARD: Lightweight edge-oriented partitioning and replication for dynamic graphs. *PVLDB*, 9(7):540–551, 2016.

[26] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SISC*, 20(1):359–392, 1998.

[27] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[28] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, 1998.

[29] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

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

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

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

[33] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

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

[35] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. Spinner: Scalable graph partitioning in the cloud. In *ICDE*, 2017.

[36] H. Meyerhenke, B. Monien, and S. Schamberger. Graph partitioning and disturbed diffusion. *Parallel Computing*, 35(10-11):544–569, 2009.

[37] M. E. Newman. Clustering and preferential attachment in growing networks. *Physical review E*, 64(2):025102, 2001.

[38] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, 2015.

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

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

[41] M. Predari and A. Esnard. A k-way greedy graph partitioning with initial fixed vertices for parallel applications. In *PDP*, 2016.

[42] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[43] F. Rahimian, A. H. Payberah, S. Girdzijauskas, and S. Haridi. Distributed vertex-cut partitioning. In *DAIS*, 2014.

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

[45] S. Salihoglu and J. Widom. GPS: A graph processing system. In *SSDBM*, 2013.

[46] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput.*, 47(2):109–124, 1997.

[47] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[48] Z. Shang and J. X. Yu. Catch the wind: Graph workload balancing on cloud. In *ICDE*, 2013.

[49] A. Shankar and R. Bodík. DITTO: Automatic incrementalization of data structure invariant checks (in Java). In *PLDI*, 2007.

[50] G. M. Slota, K. Madduri, and S. Rajamanickam. Pulp: Scalable multi-objective multi-constraint partitioning for small-world networks. In *Big Data*, 2014.

[51] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, 2017.

[52] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, 2012.

[53] T. Teitelbaum and T. W. Reps. The Cornell program synthesizer: A syntax-directed programming environment. *CACM*, 24(9), 1981.

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

[55] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *ICDE*, 2014.

[56] C. Xie, L. Yan, W. Li, and Z. Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In *NIPS*, 2014.

[57] N. Xu, L. Chen, and B. Cui. LogGP: A log-based dynamic graph partitioning method. *PVLDB*, 7(14):1917–1928, 2014.

[58] T. A. K. Zakian, L. Capelli, and Z. Hu. Automatic incrementalization of vertex-centric programs. In *IPDPS*, 2019.

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