Dynamic Scaling for Parallel Graph Computations

Wenfei Fan[1,2,4] Chunming Hu[2] Muyang Liu[1] Ping Lu[2] Qiang Yin[3] Jingren Zhou[3]

[1]University of Edinburgh [2]Beihang University [3]Alibaba Group [4]SICS, Shenzhen University

{wenfei@inf, muyang.liu@}ed.ac.uk, {hucm, luping}@buaa.edu.cn, {qiang.yq, jingren.zhou}@alibaba-inc.com

ABSTRACT

This paper studies scaling out/in to cope with load surges. Given a graph G that is vertex-partitioned and distributed across n processors, it is to add (resp. remove) k processors and re-distribute G across $n + k$ (resp. $n - k$) processors such that the load among the processors is balanced, and its replication factor and migration cost are minimized.

We show that this tri-criteria optimization problem is intractable, even when k is a constant and when either load balancing or minimum migration is not required. Nonetheless, we propose two parallel solutions to dynamic scaling. One consists of approximation algorithms by extending consistent hashing. Given a load balancing factor above a lower bound, the algorithms guarantee provable bounds on both replication factor and migration cost. The other is a generic scaling scheme. Given any existing vertex-partitioner VP of users' choice, it adaptively scales VP in and out such that it incurs minimum migration cost, and ensures balance and replication factors within a bound relative to that of VP. Using real-life and synthetic graphs, we experimentally verify the efficiency, effectiveness and scalability of the solutions.

PVLDB Reference Format:
Wenfei Fan, Chunming Hu, Muyang Liu, Ping Lu, Qiang Yin, Jingren Zhou. Dynamic Scaling for Parallel Graph Computations. *PVLDB*, 12(8): 877-890, 2019.
DOI: 10.14778/3324301.3324305

1. INTRODUCTION

In the real world, an e-commerce system often experiences load surges. For instance, its load during Christmas and Valentine's Day is often much heavier, not to mention sales triggered by unexpected hot events. This gives rise to a natural question: how many processors should we allocate to such a system? Obviously, maintaining sufficient resources just to meet peak requirements is too costly [6].

This highlights the need for dynamic scaling. It is to adaptively scale out and in, *i.e.,* add and remove processors when load jumps up and down, respectively, to improve resource utilization and reduce costs. We allocate resources on demand, instead of sticking to a one-size-fit-all configuration.

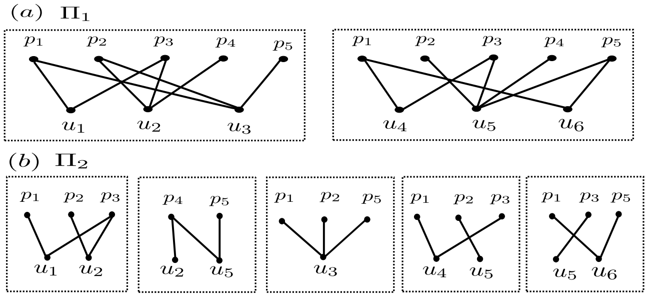


Figure 1: Partitions of a graph

Challenges. Dynamic scaling is, however, quite hard. To see this, consider an e-commerce system that employs n processors and maintains a graph G that models transactions. To maintain scalability, G is evenly partitioned into fragments and distributed across n processors for *load balancing.* Moreover, to reduce communication cost, it is often necessary to minimize the *replication factor*, *i.e.,* the copies of vertices that reside in different processors. When k processors are added or removed, it is often a must to re-partition G such that in addition to load balancing and minimum replication factor, the *migration cost* is minimized, *i.e.,* the amount of data moved from one processor to another.

Example 1: Consider the graph $G = (V, E)$ in Fig. 1 (a). It has two types of nodes: user nodes $u_1, \ldots, u_6$ and product nodes $p_1, \ldots, p_5$. In Fig. 1 (a), the edge set of G is split into two parts by a partition Π_1. Observe the following.

(1) The partition quality of Π_1 is usually measured by both balance factor and replication factor. (a) The balance factor ϵ controls that the size of each fragment is not too far from the average. Imbalanced partitions often lead to skewness and stragglers, which slow down computations. For $\epsilon \geq 0$, a partition is *ϵ-balanced* if each fragment is bounded by $\lceil(1+\epsilon)|E|/n\rceil$. For Π_1, $\epsilon = 0$ since the size of each part is at most $\lceil(1+\epsilon)|E|/2\rceil = 8$. (b) Its *replication factor* $\partial(\Pi_1) = 16/11$, defined as the ratio of total occurrences of nodes in different fragments to the total number $|V|$ of nodes in G. The smaller $\partial(\Pi_1)$ is, the better partition Π_1 is.

Consider scaling out Π_1 by adding $k = 3$ processors to $n = 2$. When $\epsilon = 0$, the size of each fragment is at most 4. Such a partition Π_2 is shown in Fig. 1 (b). To get Π_2 from Π_1, one has to move 9 edges, *e.g.,* the edges relative to u_2, u_3, u_5 and u_6, to new processors. Hence the *migration cost* from Π_1 to Π_2 is 9. Its replication factor is $\partial(\Pi_2) = 23/11$.

(2) Given ϵ, it is *not easy* to scale out Π_1 while minimizing replication factor f and migration cost m. These factors

Proceedings of the VLDB Endowment, Vol. 12, No. 8
ISSN 2150-8097.
DOI: 10.14778/3324301.3324305

interact with each other, *e.g.,* when $\epsilon = 0$, (a) to balance load, the minimum cost is 8 (different from the cost 9 for Π_2); (b) when moving 8 edges, the best f we can get is 20/11; but (c) to get an optimal $f = 18/11$, we need to move 12 edges. It is also nontrivial to identify which edges to be moved.

Moreover, graph G has to be re-partitioned *in parallel.* This is because G is already partitioned across a cluster of machines (*e.g.,* by Π_1 above); moreover, when G is large, it is not realistic to re-partition G by a single machine. □

We show that dynamic scaling is NP-complete. It remains intractable even when (a) the number k of processors added or removed during scaling is a constant, and (b) we put no restriction on either balance factor or migration cost.

While there has been work on dynamic scaling [6, 46, 42, 29, 32, 40], few of these considered how to adaptively partition graphs in scaling, and none offered guarantees on balance factor, replication factor and migration cost.

One might think that incremental graph partitioners [45, 37, 36, 41, 16, 30, 48, 8, 40] could be used for dynamic scaling. Given a partition $\mathcal{P}(G)$ of graph G and updates ΔG to G, it is to compute changes ΔO to $\mathcal{P}(G)$ such that $\mathcal{P}(G \oplus \Delta G) = \mathcal{P}(G) \oplus \Delta O$, where $\oplus$ applies changes ΔG (resp. ΔO) to G (resp. $\mathcal{P}(G)$). However, (a) the two are different problems: dynamic scaling is to re-partition graph G in response to addition or removal of k processors, not to changes ΔG to G. Moreover, (b) in practice it is often the case that $k > n$, and hence the changes ΔG and ΔO are large. It is known that when the changes are large, incremental partitioning works no better than re-partitioning the entire graph G starting from scratch. Thus incremental partitioning techniques do not apply to dynamic scaling and vice versa.

Approximation and generic methods. We propose two solutions. There are two general approaches to graph partitioning: edge-cut and vertex-cut. We focus on vertex-cut here since it has not been as well studied as edge-cut.

(1) Approximate algorithms. In light of the intractability of dynamic scaling, the best practical solution we can hope for is approximation. We develop such a solution that consists of two approximate algorithms. Given a vertex-cut partition $\Pi(n)$ of a graph G via hashing, balance factor ϵ and a number k, algorithms BVC^- and BVC^+ scale in and out $\Pi(n)$ to get a new ϵ-balanced partition $\Pi(n-k)$ and $\Pi(n+k)$, respectively, by extending consistent hashing. Better yet, we show that when ϵ is above a small threshold, the algorithms guarantee bounds on both replication factor f and migration cost m. To the best of our knowledge, the algorithms make the first solution to dynamic scaling with such bounds.

(2) A scaling scheme. While the solution above offers provable bounds on f and m, it requires to start with an initial partition based on hashing. Is it possible to scale an arbitrary vertex-cut partitioner VP of users' choice?

The answer is affirmative. We propose a generic scheme. Given an existing VP, it deduces two algorithms VP^+ and VP^- to scale VP out and in, respectively. We show that these algorithms incur minimum migration cost. Moreover, its partition quality is within a bound relative to that of VP. That is, while the scaling scheme provides no absolute bounds like the approximate algorithms above, it provides bounds relative to VP. Hence if users have been using VP, the quality of VP^+ and VP^- is acceptable to them.

Contributions & Organization. Putting these together, the paper (1) formalizes the dynamic scaling problem and establishes its complexity (Section 2); (2) provides an approximation solution with bounds on replication factor and migration cost (Section 3); and (3) proposes a generic scheme to scale existing vertex-cut partitioners with low migration cost and relative bounds on partition quality (Section 4).

(4) Experimental study (Section 5). Using real-life and synthetic graphs, we empirically verify the efficiency, partition quality and scalability of our scaling algorithms. We find the following. (a) Parallel BVC^+ (resp. BVC^-) algorithm outperforms hash-based and stream-based competitors by 7.4 and 19.7 (resp. 8.5 and 18.2) times in efficiency, respectively. (b) These algorithms also do better in replication factor than hash-based competitors by 1.94 and 2.04 times, up to 3.82 and 3.79 times. (c) Our generic scaling scheme is promising. Two stream-based scaling algorithms deduced under this scheme are able to achieve partition quality as good as re-partitioning, and are 43.8 and 40.7 times faster on average, up to 114.7 and 132.3 times. (d) Our algorithms scale well with large n, k and graphs; *e.g.,* parallel BVC^+ (resp. BVC^-) takes 9.45s (resp. 11.37s) on graphs with 440 million nodes and 14 billion edges when $n = 320$ and $k > \frac{n}{3}$.

This work is among the first treatments of dynamic scaling, from approximation to scaling of existing partitioners.

Related work. We summarize the related work as follows.

Graph partitioning. Vertex-cut was proposed in [15]. It was shown in [3] that it is NP-complete to minimize the replication factor f when evenly partitioning a graph. It is NP-hard even when the balance factor is fixed [47]. A simple vertex-cut strategy is to assign edges to fragments randomly by hashing. However, this usually leads to bad locality since it ignores the structures of input graphs [5]. 2DHash [44] mitigates this problem by maintaining a $2\sqrt{n}-1$ bound on f, where n is the number of fragments. Degree-based hash partitioning [43] assigns edges based on vertex degrees and favors cutting vertices with relatively large degrees. HDRF [31] also replicates (or cuts) high-degree vertices in streaming partition. Apart from these, several heuristics were developed, *e.g.,* [5, 25, 47].

This work differs from the prior work in the following.

(1) As a special case of Theorem 1 ($k = 0 \wedge m = \infty$), we show that vertex-cut partitioning is NP-hard even when we put no constraint on the balance factor ϵ. This is analogous to its edge-cut counterpart [14]. This is not implied by the results of [3, 47], and cannot be improved by further restricting ϵ.

Moreover, we settle the complexity of dynamic scaling and reveal what dominates the cost (Theorem 1). To the best of our knowledge, no previous work has studied this issue.

(2) For partition quality, algorithms BVC^+ and BVC^- guarantee both a bound on the replication factor and the balance of partitions. The bound differs from the one of the degree-based approach in [43] by only a small factor, a small price for balancing, which is not guaranteed by [44, 43, 31, 25].

(3) BVC^+ and BVC^- adopt consistent hashing to prepare for dynamic scaling, which allows us to adjust an existing partition in response to adding or removing processors, without re-partitioning the graph starting from scratch. It was not studied in the prior work [44, 15, 43, 31, 3, 47].

Consistent hashing. The method was proposed in [17] to reduce the movement of hashed clients when the size of hash table changes (see Section 3.1). As shown in [33, 27], when there are far more clients than servers as in real-life dynamic scaling, simple consistent hashing [17] suffers from imbalanced load. In [27], a simple linear probing technique was integrated into consistent hashing to deal with load balancing.

A popular variant is DHT (distributed hash table), *e.g.,* CAN [34] and Chord [38]. DHT employs consistent hashing to store key-value pairs in a distributed setting, for users to locate a key-value pair with a given key, via "hashing".

Closer to this work are [34, 28, 24, 18, 19] for adding or removing servers (analogous to fragments) in DHT, and [9, 23] for balancing the workload of servers in DHT. When adding a new server, CAN [34] bisects a randomly picked zone, which plays the same role as an "interval", and assigns one of the half zones to the new server. A bucket solution was given in [28, 24] to handle server removal, and multiple-choice algorithms were used in [28, 19] to add servers. Servers are evenly distributed over a unit circle for load balancing [9]. Upper and lower bounds for workload are used to guide interval adjustments [9].

Our work differs from the prior work in the following.

(1) In contrast to [17, 27] that hash fragments, we assign the fragments in a different way to ensure that its distribution is as uniform as possible. This also helps us balance load when used together with the technique of [27].

(2) We propose a strategy to add or remove fragments for dynamic scaling. (a) To add fragments, we bisect a largest interval, rather than randomly picking one [34, 28]; (b) we define an order in which fragments are removed; and (c) we add or remove fragments, but do not move fragments as in [28, 24, 18]. These help us guarantee provable bounds on load balance, replication factor and migration cost.

(3) We integrate a degree-based approach [43] with consistent hashing, to leverage the coherence of edges (or clients) and bound the replication factor. In contrast, consistent hashing often treats all clients equally and thus ignores their coherence. Directly adopting such approaches in our setting fails to provide a bound on the replication factor.

Scaling. The study of dynamic scaling has mostly focused on how to allocate virtual machines (VMs) when load varies in cloud computing [6, 46, 42, 29], or how to reduce energy consumption when workload is low [21, 22].

The scaling problems studied in the prior work differ from $\mathsf{DS}(\epsilon, f, m)$ (Section 2) in that it does not consider graph partitioning, not to mention its three objectives (ϵ, f, m).

Closer to this work are [32, 40, 7, 12], which study graph partitioning in dynamic scaling; these focus on edge-cut partitioning. A greedy heuristic was developed in [32] to migrate vertices when scaling; [40] randomly picks vertices based on a given probability, and moves the vertices to other fragments in response to changes to the graphs; [7] adopts a lazy strategy: when a worker is added, necessary vertices are moved to it only when the worker processes a query; [12] uses a bin-packing model to balance workers after scaling.

This work differs from [32, 40, 7, 12] as follows. (a) We study scaling with vertex-cut partition, which is not yet well studied, as opposed to edge-cut. (b) None of [32, 40, 7, 12] guarantees partition quality as we do. In particular, [7] accumulates vertices at new workers and is not load balanced.

2. THE DYNAMIC SCALING PROBLEM

We first state the problem and settle its complexity.

Preliminaries. We consider (un)directed graphs $G = (V, E)$, where V is the set of vertices, and $E \subseteq V \times V$ is the set of edges. Denote by (a) $v(e) = \{u, w\}$ the set of two end-points of an edge e, and (b) $v(E') = \bigcup_{e \in E'} v(e)$ the set of vertices that are on the edges in a set $E' \subseteq E$.

Partitions. A *vertex-cut n-partition* of graph $G = (V, E)$ is $\Pi(n) = (E_1, E_2, \ldots, E_n)$, which partitions the edge set E into n disjoint sets. We refer to E_i as a fragment of $\Pi(n)$.

A n-partition $\Pi(n)$ induces n subgraphs $G_1, G_2, \ldots, G_n$ of G, where $G_i = (v(E_i), E_i)$, such that $V = \bigcup_{i \in [1,n]} v(E_i)$ and $E = \bigcup_{i \in [1,n]} E_i$. To simplify the presentation, we assume *w.l.o.g.* that each E_i is nonempty in the sequel.

There are two criteria to evaluate the quality of $\Pi(n)$.

(a) Balance factor. Given $\epsilon \geq 0$, $\Pi(n)$ is called *ϵ-balanced* if

$$\max\{|E_1|, \ldots, |E_n|\} \leq \lceil (1+\epsilon)|E|/n \rceil.$$

That is, no E_i is substantially larger than the average.

(b) Replication factor. The *replication factor* of $\Pi(n)$ is

$$\partial(\Pi(n)) = \frac{1}{|V|} \sum_{i=1}^{n} |v(E_i)|.$$

Intuitively, the larger $\partial(\Pi(n))$ is, the higher the communication cost is for synchronization in a distributed setting.

Scaling. Given an integer $k \in (-n, \infty)$ and a n-partition $\Pi(n)$ of G, we want to reconfigure $\Pi(n)$ to a new partition $\Pi(n+k)$. This is called *scaling in* if $-n<k<0$ by reducing $|k|$ processors; and *scaling out* if $k > 0$ by adding k processors.

The *migration cost* from $\Pi(n)$ to $\Pi(n+k)$ is the number of edges moved to get $\Pi(n+k)$, including (a) edges migrated from $G_1, \ldots, G_n$ to the (new) fragments of $\Pi(n+k)$, and (b) edges moved among $G_1, \ldots, G_{n+k}$ to be rebalanced.

The *dynamic scaling problem* is stated as follows.

- *Input*: A n-partition $\Pi(n)$ of G, an integer $k > -n$, a balance factor ϵ, a replication factor f, and a bound m.
- *Question*: Does there exist an ϵ-balanced vertex-cut $(n+k)$-partition $\Pi(n+k)$ of G such that $\partial(\Pi(n+k)) \leq f$ and migration cost from $\Pi(n)$ to $\Pi(n+k)$ is at most m?

That is, under balance factor ϵ and replication factor f, it aims to minimize the migration cost of dynamic scaling.

Complexity. The dynamic scaling problem bears three criteria: a balance factor ϵ, a replication factor f and a bound m on moving cost. We denote it as $\mathsf{DS}(\epsilon, f, m)$ or simply DS.

To identify the impact of the three criteria on the complexity, we also study three variants of $\mathsf{DS}(\epsilon, f, m)$, when one of the three criteria is dropped. Denote by $\mathsf{DS}(f, m)$, $\mathsf{DS}(\epsilon, m)$ and $\mathsf{DS}(\epsilon, f)$ the three variants when dropping constraints on balance factor ϵ, replication factor f and migration cost m, respectively. For example, $\mathsf{DS}(f, m)$ asks whether there exists a partition $\Pi(n+k)$ of G such that $\partial(\Pi(n+k)) \leq f$ and migration cost from $\Pi(n)$ to $\Pi(n+k)$ is at most m, no longer requiring $\Pi(n+k)$ to be load balanced.

It is not surprising that $\mathsf{DS}(\epsilon, f, m)$ is NP-complete. We show that the intractability is quite robust: it remains NP-hard as long as f is one of the optimization goals, even when the number of processors added or removed is fixed.

Theorem 1: *(1) Each of* DS, $\mathsf{DS}(f, m)$ *and* $\mathsf{DS}(\epsilon, f)$ *is* NP-*complete, and remains* NP-*hard even when k is a constant.*

(2) $\mathsf{DS}(\epsilon, m)$ *is in* PTIME*; and* $\mathsf{DS}(f, m)$ *is in* PTIME *when both* k *and* n *are fixed and when* m *is* ∞ *(unrestricted).* □

Proof: (1) An NP algorithm for DS works as follows: it first guesses a $(n + k)$-partition and then checks in PTIME whether the three constraints are satisfied. Hence DS is in NP, and so are its special cases $\mathsf{DS}(f, m)$ and $\mathsf{DS}(\epsilon, f)$.

We verify the NP-hardness of DS and $\mathsf{DS}(\epsilon, f)$ by reduction from the 3-partition problem [2], and $\mathsf{DS}(f, m)$ by reduction from the maximal clique problem (cf. [13]). The reductions are constructed with constant k.

(2) For $\mathsf{DS}(\epsilon, m)$, the PTIME algorithm below suffices. Each time it moves one edge from the largest fragment to a minimum one until either (a) the balance factor gets back to ϵ (Yes); or (b) the migration cost exceeds the bound m (No).

When neither ϵ nor m is bounded and both n and k are constants, we first show that there is a partition such that its replication factor is minimal, and the number of cut nodes is bounded by a constant. Based on this property, we give a PTIME algorithm for $\mathsf{DS}(f, m)$ with $m{=}\infty$: enumerate all possible sets of cut nodes; check whether any of the associated partitions has replication factor no larger than f. □

3. APPROXIMATION ALGORITHMS

In light of the intractability of $\mathsf{DS}(\epsilon, f, m)$, the best practical solutions are approximate algorithms. We now develop such a solution. It consists of algorithms BVC^+ and BVC^- to scale out and in a partition $\Pi(n)$ of a graph to an ϵ-balanced partition $\Pi(n + k)$, respectively (Section 3.2). Given any balance factor ϵ above a small threshold, both algorithms guarantee bounds on replication factor f and migration cost m. We parallelize these algorithms (Section 3.3), retaining the same bounds. We are not aware of other dynamic scaling solutions that offer such bounds.

Our solution extends consistent hashing [17, 27] and hash-based partitioning [43]. We remark the following (see Section 1 for details). (1) None of the prior algorithms works on dynamic scaling, especially for deciding which fragments to be removed or added while ensuring a bound on replication factor f. (2) As observed in [4, 18, 44, 33, 27], consistent hashing does no better than random hash partitioning and gives no guarantee on partition quality. (3) In particular, the algorithms of [17, 27] have no guarantee on replication factor f, and [43] gives no guarantee on balance factor ϵ.

3.1 Consistent Hashing and Extension

We first review consistent hashing, and then outline our extension to cope with dynamic scaling. Consider mapping M balls to N bins. Consistent hashing [17] is a hash-style solution, using two different hash functions h_M and h_N, with the same range. The range is modeled as a hash ring, a unit circle $\mathcal{C}$. It first hashes the balls and bins to locations on $\mathcal{C}$ by applying h_M and h_N, respectively. Each ball is then mapped to the nearest bin on $\mathcal{C}$ in the clockwise order.

Its advantage is that when the number of bins changes dynamically, the number of balls that need remapping is small. When removing a bin from $\mathcal{C}$ (scale in), only the balls in the deleted bin are remapped to the next bin on $\mathcal{C}$ in the clockwise order. When adding a new bin on $\mathcal{C}$ (scale out), it first finds certain balls that are hashed to locations between the new bin and its previous bin in the clockwise order. It then remaps these balls to the new bin.

For dynamic scaling, we can model edges as balls and fragments of a partition as bins, and apply consistent hashing. However, we need to address the following challenges.

(1) Replication factor. Consistent hashing treats all balls equally. This is equivalent to hashing edges by a random hash function, which, as observed by [44], often leads to poor locality. To rectify this, we employ degree-based hashing proposed in [43], which favors cutting vertices with relatively large degrees. Intuitively, the replication factor gets smaller when more vertices with large degrees are cut.

(2) Load balance. By hashing balls, a bin may have far more balls than the others. Moreover, when $M \gg N$, the maximum load may deviate from the average by $\sqrt{\frac{2M \log N}{N}}$ [33], where M and N are the number of balls and bins, respectively. One might want to add virtual workers to mitigate the unbalance [17], but it works only when $M = O(N \log N)$ [33]. For graph partitioning, the number of balls is much larger than the number of bins, *i.e.,* $M \gg N$, and adding virtual workers (a fragment is mapped to multiple positions in circle $\mathcal{C}$) cannot make the bins balanced.

To balance the workload, we enforce a given balance factor as a *hard constraint*, and rebalance partitions by using a linear probing technique [27]. In addition, we adopt degree-based hashing and extend consistent hashing to weighted consistent hashing, which was not studied in [17, 27].

(3) Migration cost. Consistent hashing maps fragments as bins on the circle $\mathcal{C}$ by hash functions. However, when $M \gg N$, which is typically the case in our setting, this usually incurs heavy cost in graph partition. This is because when balls are not distributed evenly, some bins may be overfull, and balancing the bins increases the migration cost.

To minimize the cost, we propose a *fragment placement strategy.* Instead of hashing the fragments, we first evenly distribute the fragments on the circle $\mathcal{C}$ [9]. When scaling in or out, our placement strategy selects fragments to be removed or added, and places the fragments on $\mathcal{C}$ as uniformly as possible. We will see that this allows us to bound the migration cost. It also helps us improve partition quality.

Notations. We will use the following notations. Consider a graph $G = (V, E)$ in which each vertex $v \in V$ has a unique global id $v.\mathsf{id}$. Given a unit circle $\mathcal{C}$ and a constant c, we divide it into 2^c segments, and use it as the hash ring. We use only one hash function h_M that maps the id's of vertices to the locations of $\mathcal{C}$, *i.e.,* to the set $\{0, 1, \ldots, 2^c - 1\}$.

We consider power-law graphs. A graph follows power-law if the probability that a vertex has degree d is given by

$$\Pr(d) \propto d^{-\alpha},$$

where α is the *power-law constant* that controls the "skewness" of degree distribution. Many real-life graphs follow the power law and have a power-law constant around 2 [15]. The power-law constant helps us bound replication factor, but it has no impact on the bound on migration cost.

3.2 Algorithms for Scaling Out and In

We now present algorithms BVC^+ and BVC^- for dynamic scaling out and in, respectively. Given a partition $\Pi(n) = (E_1, \ldots, E_n)$ of graph G and a number $k > -n$, BVC^+ and BVC^- adjust $\Pi(n)$ to get a new partition $\Pi(n + k)$. As remarked earlier, the algorithms extend consistent hashing. Below we first show how to obtain an initial partition, to which BVC^+ and BVC^- are applied. We then present our scaling algorithms and prove the performance guarantees.

Initial partition. Given a graph G and a number n, we extend consistent hashing to compute an initial partition $\Pi(n) = (E_1, \ldots, E_n)$ of G. In contrast to classical consistent hashing, (i) we use degree-based hashing to improve replication factor; and (2) we evenly distribute the fragments on the unit circle $\mathcal{C}$ to reduce migration cost. More specifically, $\Pi(n) = (E_1, \ldots, E_n)$ is computed as follows.

(1) We first evenly distribute the fragments $E_1, \ldots, E_n$, *i.e.,* bins, initially empty, on the circle $\mathcal{C}$. This is done by allocating each E_i ($i \in [1, n]$) at position $i\lceil \frac{2^c-1}{n} \rceil$ on $\mathcal{C}$.

(2) We then hash each edge $e \in E$ by using its vertex with a relatively smaller degree. More specifically, the hash value $e.\mathsf{hash}$ of an edge $e = (u, v)$ is defined by

$$e.\mathsf{hash} = \begin{cases} h_M(v.\mathsf{id}) & \mathsf{deg}(v) < \mathsf{deg}(u), \\ h_M(u.\mathsf{id}) & \mathsf{otherwise}. \end{cases}$$

This favors cutting vertices with relative large degrees. Edge e is then assigned to the nearest fragment clockwise. More specifically, denote by $L_1, L_2, \ldots, L_n$ the positions of E_1, $\ldots$, E_n on $\mathcal{C}$ respectively, we assign e to $E_{\text{next\_par}(e,\mathcal{C})}$, where

$$\text{next\_par}(e, \mathcal{C}) = \text{argmin}_{i \in [1,n]}((L_i - e.\mathsf{hash}) \mod 2^c).$$

Example 2: For graph G of Fig. 1 (a), let $c = 5$, *i.e.,* to divide circle $\mathcal{C}$ into 2^5 segments. Assume that hash function h_M maps vertices onto $\mathcal{C}$: $p_1 \to 2$, $p_2 \to 20$, $p_3 \to 22$, $p_4 \to 29$, $p_5 \to 30$, $u_1 \to 10$, $u_2 \to 12$, $u_3 \to 5$, $u_4 \to 21$, $u_5 \to 26$, $u_6 \to 25$. Let $n = 2$, then the initial partition $\Pi(2) = (E_1, E_2)$ obtained as above is $E_1 = \{e_{1,1}, e_{1,3}, e_{2,2}, e_{2,3}, e_{3,1}, e_{3,2}, e_{3,5}\}$, and $E_2 = \{e_{2,4}, e_{4,1}, e_{4,3}, e_{5,2}, e_{5,3}, e_{5,4}, e_{5,5}, e_{6,1}, e_{6,5}\}$. □

Overview of BVC^+ and BVC^-. Given a number $k > -n$, a balance factor ϵ, and a partition $\Pi(n)$ that is an initial partition obtained as above, algorithms BVC^+ and BVC^- adjust $\Pi(n)$ to $\Pi(n+k)$ in three steps as follows.

(1) Step (1) updates fragment placement on the circle $\mathcal{C}$. Suppose that for $i \in [1, n]$, fragment E_i is placed at location L_i before scaling starts. Given k, step (1) identifies $|k|$ locations to remove (scale in) or add (scale out) fragments.

To minimize the migration cost in the next steps, we propose a strategy to place the fragments uniformly. Let I_1, $\ldots$, I_n be the n intervals on $\mathcal{C}$ induced by $E_1, \ldots, E_n$, *i.e.,*

$$I_i = (L_{\mathsf{next}(i)} - L_i) \mod 2^c$$

where $\mathsf{next}(i) = (i+1) \mathsf{ mod } n$. Denote by $I_{\max} = \mathsf{max}\{I_i\}_{i=1}^n$ and $I_{\min} = \mathsf{min}\{I_i\}_{i=1}^n$. We select $|k|$ locations for dynamic scaling, and ensure the following *interval invariant*:

$$I_{\max} \leq 2I_{\min}, \tag{1}$$

i.e., the maximum interval has size at most twice the size of the minimum one. As will be seen shortly, this interval invariant will be used to bound both migration cost and replication factor. Note that the initial partition satisfies the interval invariant. Starting from an evenly distributed placement of fragments, we will propose a strategy to maintain the interval invariant during scaling.

(2) It then employs consistent hashing to update edge assignments as we did in the initial partition construction.

(3) It restores balance via linear probing [27] (see below).

We will see that when ϵ is not too small, BVC^- and BVC^+ guarantee bounds on migration cost and replication factor.

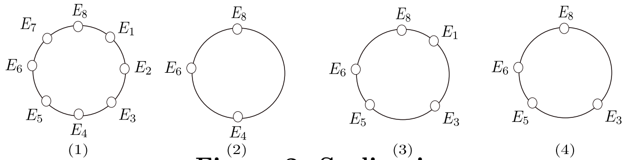


Figure 2: Scaling in

Fragment placement. We use a stack to keep track of the order of locations when the circle $\mathcal{C}$ is adjusted by removing or adding fragments. When we remove a fragment, we remove the one on the top of the stack, and when we add a new fragment, we push its location onto the stack.

Initial stack. The stack is initialized with the n fragments $E_1, \ldots, E_n$ when the initial partition is constructed. We decide a specific order such that we do not remove two consecutive fragments at the same time when scaling in, since otherwise it may triple the size of the intervals and violate the invariant. Indeed, the fragments are evenly distributed on $\mathcal{C}$, and the size of the smallest interval is $\lceil \frac{2^c-1}{n} \rceil$. When we remove two consecutive fragments, *e.g.,* fragments located at $i\lceil \frac{2^c-1}{n} \rceil$ and $(i+1)\lceil \frac{2^c-1}{n} \rceil$, we get an interval from $(i-1)\lceil \frac{2^c-1}{n} \rceil$ to $(i+2)\lceil \frac{2^c-1}{n} \rceil$, and its size is $3\lceil \frac{2^c-1}{n} \rceil$, which triples the size of the smallest intervals.

More specifically, suppose that $E_1, \ldots, E_n$ are located in the clockwise order on $\mathcal{C}$. We start from E_1, walk the circle clockwise, and pick every other fragment. We proceed until no fragment is left. This yields an order $E_1, E_3, \ldots, E_t$. We push their locations onto the stack in the reverse order, *i.e.,* E_1 is on top of the stack, and E_t is at its bottom.

We next give our strategy to remove and add fragments.

Removing fragments. To remove $|k|$ fragments from the circle $\mathcal{C}$, we simply pop up $|k|$ locations from the stack one by one, and remove their corresponding fragments.

Adding fragments. To add a new fragment E', we find the largest interval on $\mathcal{C}$, place E' in the middle of the interval, and push the location of E' onto the stack. If there exist multiple largest intervals of the same size, we randomly pick one. To add k fragments, we repeat the process k times.

Lemma 2: *The interval invariant holds when fragments are added or removed as described above.* □

Proof: We show that if the invariant holds before scaling, then it also holds after it. Observe that after adding fragments, the size of the largest interval decreases; and after removing fragments, the smallest interval increases. For scaling out, $I_{\max} \leq 2I_{\min}$ because we bisect the largest interval, and obtain two smallest intervals. For scaling in, we merge two smallest intervals and generate a largest one. □

Example 3: Suppose that we initially have 8 fragments as shown in Fig. 2 (1). We show how to remove 5 fragments.

(1) Based on the strategy, the fragments in Fig. 2 (1) are ordered as $E_1 \to E_3 \to E_5 \to E_7 \to E_2 \to E_6 \to E_4 \to E_8$. We remove the first 5 fragments (E_1, E_3, E_5, E_7 and E_2) in the order, yielding Fig. 2 (2). The intervals have size $\frac{1}{2} \times 2^c$, $\frac{1}{4} \times 2^c$ and $\frac{1}{4} \times 2^c$, respectively. The invariant holds.

(2) One might want to remove fragments also by picking the smallest intervals. However, this may violate the invariant. For instance, if we remove fragments surrounded by two minimum intervals, *e.g.,* E_2, E_4 and E_7 from Fig. 2 (1), we end up with Fig. 2 (3), and can no longer remove more frag-

```
Algorithm BVC$^+$
Input: A partition $\Pi(n) = (E_1, \ldots, E_n)$ of $G$,
        a number $k > 0$, and a balance factor $\epsilon$.
Output: An $\epsilon$-balanced new partition $\Pi(n+k)$=$(E_1, \ldots, E_{n+k})$.

/* Step (1): Adjust fragments on $\mathcal{C}$ */
1.  identify $k$ locations $L_{n+1}$, ..., $L_{n+k}$ for fragments to add;
2.  add $k$ new fragment such that $E_{n+j}$ at $L_{n+j}$ for $j \in [1, k]$;
/* Step (2): Reallocate edges via consistent hashing */
3.  for each $e \in \bigcup_{i=1}^{n} E_i$ do
4.      $i^*$=next_par($e$.hash, $\mathcal{C}$); /*get the next fragment on $\mathcal{C}$*/
5.      if $i^* \in \{n+1, \ldots, n+k\}$ then
6.          move $e$ to fragment $E_{i^*}$;
/* Step (3): Balancing */
7.  $w \leftarrow \lceil (1+\epsilon)\frac{|E|}{n+k} \rceil$;
8.  while there exists some $E_i$ with $|E_i| > w$ do
9.      $\Delta E_i \leftarrow$ select $(|E_i| - w)$ edges from $E_i$;
10.     $E_i \leftarrow E_i \setminus \Delta E_i$;
11.     next $\leftarrow (i+1) \mod n$;
12.     migrate $\Delta E_i$ to fragment $E_{\text{next}}$;
13.     $E_{\text{next}} \leftarrow E_{\text{next}} \cup \Delta E_i$;
```

```
Algorithm BVC$^-$
Input: A partition $\Pi(n) = (E_1, \ldots, E_n)$ of $G$,
        a number $0 < k < n$, and a balance factor $\epsilon$.
Output: A new partition $\Pi(n) = (E'_1, \ldots, E'_{n-k})$ of $G$.

1.  identify and remove fragments $E_{j_1}, \ldots E_{j_k}$, with a stack;
2.  for each edge $e \in \bigcup_{i=1}^{k} \{E_{j_i}\}$ do
3.      $i$ = next_par($e$.hash, $\mathcal{C}$); /*get the next fragment on $\mathcal{C}$ */
4.      move $e$ to $E_i$;
5.  $\{E'_1, E'_2, \ldots, E'_{n-k}\} \leftarrow \{E_1, \ldots, E_n\} \setminus \{E_{j_1}, \ldots, E_{j_k}\}$;
6.  balance $E'_1, \ldots, E'_{n-k}$ by linear probing as Algorithm BVC$^+$;
```

Figure 3: Algorithm for scaling out/in

ment without violating the invariant. Indeed, if we further remove E_1, we end up with Fig. 2 (4), in which the distance between E_8 and E_3 triples the distance between E_5 and E_6. Removing other fragments also inflicts violation. □

We now present algorithms BVC$^+$ and BVC$^-$ in Fig. 3.

Algorithm BVC$^+$ Given $\Pi(n)$, ϵ and $k > 0$, BVC$^+$ extends $\Pi(n)$ to $\Pi(n+k)$ in three steps. (1) It first adds new fragments on circle $\mathcal{C}$ as remarked earlier, maintaining the interval invariant. (2) It then re-allocates edges by a degree-based approach to improve locality, and maps edges to fragments as in consistent hashing. (3) Finally it adjusts the partition to make it balanced. Steps (2) and (3) integrate consistent hashing [17, 27] and the degree-based approach [43].

(1) It first identifies k locations with the placement strategy above, and adds k new fragments at the locations (lines 1-2).

(2) It then identifies edges belonging to the new fragments based on consistent hashing and moves them to the corresponding new fragments (lines 3-6).

(3) Finally, it applies linear probing [27] to balance the partition (lines 7-13). For each fragment E_i, if it is not balanced $(|E_i| > \lceil (1+\epsilon)\frac{|E|}{n+k} \rceil)$, then it forwards $|E_i| - \lceil (1+\epsilon)\frac{|E|}{n+k} \rceil$ edges to the next fragment in the clockwise order.

Remark. (a) BVC$^+$ terminates when all fragments are balanced. This is assured by that each edge is migrated at most $n+k$ times, and at most $|E|$ edges need to be moved.

(b) The initial partition step can be done by BVC$^+$, denoted by BVC. Indeed, it is a special case when the graph is given as a fragment, and BVC$^+$ adds another $n-1$ fragments.

Example 4: We show how BVC$^+$ extends the partition $\Pi(2)$ of Example 2 to a new partition $\Pi(5) = (E_1, \ldots, E_5)$. It first identifies 3 locations on circle $\mathcal{C}$ to place the new fragments E_3, E_4 and E_5. It then finds edges that belong to the new fragments, and moves them to the right place. We get $E_1 = \{e_{1,1}, e_{1,3}, e_{2,2}, e_{2,3}\}$, $E_2 = \{e_{2,4}, e_{5,4}, e_{5,5}\}$, $E_3 = \{e_{3,1}, e_{3,2}, e_{3,5}\}$, $E_4 = \{e_{4,1}, e_{4,3}, e_{5,2}\}$ and $E_5 = \{e_{5,3}, e_{6,1}, e_{6,5}\}$. This yields balanced $\Pi(5)$ of Fig. 1 (b). □

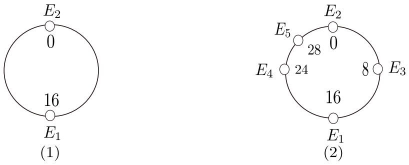


Figure 4: Scaling out

Algorithm BVC$^-$ Given a balance factor ϵ, a number k such that $-n<k<0$, and a partition $\Pi(n) = (E_1, \ldots, E_n)$ of G such that E_i's are placed on a unit circle $\mathcal{C}$, BVC$^-$ adjusts $\Pi(n)$ to $\Pi(n+k)$ as follows. It first identifies $|k|$ fragments E_{j_1}, ..., $E_{j_{|k|}}$ on the top of the stack, and removes them from circle $\mathcal{C}$ (line 1). As assured by Lemma 2, after the removal, the circle $\mathcal{C}$ still satisfies the interval invariant.

After these steps, BVC$^-$ remaps the edges in E_{j_1}, ..., $E_{j_{|k|}}$ to the remaining fragments based on consistent hashing (lines 2-4). More specifically, for each edge e in a removed fragment, it finds the next fragment on $\mathcal{C}$ in the clockwise order (line 3) and moves e to it (line 4). At last it balances the fragments via linear probing as in BVC$^+$ (lines 5-6).

Analysis. We show that when the balance factor is not too small, BVC$^+$ and BVC$^-$ guarantee bounds on both replication factor and migration cost. Since each edge is hashed by its vertices, denote by $h_{\max}$ the maximum number of times of a vertex used for hashing. Here $h_{\max}$ is usually much smaller than the maximum degree of the graph, as for a vertex it is unlikely that most of its edges are hashed using its id.

Given $k > -n$, we have the following starting from an initial partition with BVC$^+$, in which $\beta_k^1 = \frac{8(n+k)h_{\max}}{|E|} \log((n+k)\sqrt{|E|+1})$, $\beta_k = \sqrt{\beta_k^1}(\sqrt{\beta_k^1} + \sqrt{2})$, and $\theta = d_{min} \times \frac{\alpha-1}{\alpha-2} - d_{min} \times \frac{\alpha-1}{2\alpha-3} + \frac{1}{2}$, where d_{min} is the minimal node degree in a power-law graph, and α is its power-law constant [43].

Theorem 3: *If $k > -n$ and $\epsilon > 1 + 2\beta_k$, then (1) the expected value of migration cost when scaling out (resp. in) from $\Pi(n)$ to $\Pi(n+k)$ via* BVC$^+$ *(resp.* BVC$^-$*) is at most $O(k\frac{|E|}{n+k})$ (resp. $O(k\frac{|E|}{n})$); and (2) the expected value of the replication factor is at most $(n+k)(1-(1-2\frac{1}{n+k})^{\theta}) + \frac{2}{|V|}$.* □

Observe the following about Theorem 3.

(1) The lower bound β_k for balance factor is not very restrictive, since in the real world it is common to find that $|E| \gg n$. Taking Twitter as an example (see Section 5), $\beta_k \leq 0.009$ for n=64, where $|E|$ is approximately 1.5 billion.

(2) Edge selection in linear probing affects neither migration cost [27] nor the upper bound for replication factor.

(3) The bound for migration cost holds on general graphs, but not the replication factor f_e. On a power-law graph G, f_e of degree-bashed hashing would decrease when G gets more skewed [43]; this does not hold on general graphs.

Proof: We only give a proof sketch for the bounds for BVC$^-$; the proof for BVC$^+$ is similar.

(1) The migration cost of BVC$^-$ includes (a) the cost of moving edges from removed fragments to fragments that remain; and (b) the cost of rebalancing fragments. For cost (a), since

each fragment has at most $\lceil (1+\epsilon)\frac{|E|}{n} \rceil$ edges, and k fragments are removed, at most $O(k\frac{|E|}{n})$ edges are migrated. Thus the migration cost for (a) is bounded by $O(k\frac{|E|}{n})$.

For cost (b), we show that the expected number of edges in each fragment E_i to be forwarded is bounded by $O(\frac{1}{n^2})$, by using Bernstein's inequality [10]. Since each edge can be forwarded at most n times, the migration cost for balancing each fragment is at most $O(\frac{1}{n})$. Hence total migration cost for balancing all n fragments is bounded by $O(1)$.

(2) Suppose that V_i is the set of vertices contained in fragment E_i ($i \in [1, n{+}k]$) after BVC^- terminates. To bound the replication factor, by its definition, we only need to bound the expected value of $|V_i|$ for all $i \in [1, n+k]$. Note that $|V_i|$ can be bounded by the number of vertices hashed to E_i plus the number of vertices forwarded to E_i during the rebalancing step. The number of vertices hashed to E_i can be bounded by $|E|(1-(1-\frac{2}{n+k})^{\theta})$ using the technique of [43], since the fragments are such placed that the invariant holds, and the probability that an edge is hashed to E_i is bounded by $\frac{2}{n+k}$. For the number of vertices forwarded to E_i, since the total number of forwarded edges is bounded by $O(1)$ as proved above, and each edge has two associated vertices, the number of vertices forwarded to E_i can also be bounded. □

3.3 Parallelization

Dynamic scaling has to be conducted in parallel. It starts with a partition when a graph is already fragmented and distributed across a cluster of processors. To scale out/in, all processors involved need to work together in parallel. Moreover, when dealing with large graphs, it is not practical for a single-machine to compute a balanced partition.

In light of this, we next parallelize BVC^+ and BVC^-, and develop their parallel versions ParBVC^+ and ParBVC^-, respectively. We show that these parallel algorithms retain the same performance guarantees as their serial counterparts.

Parallel setting. Our parallel algorithms run in a shared-nothing distributed setting, as commonly used nowadays.

(a) Initially, a graph $G = (V, E)$ is partitioned into n fragments $E_1, \ldots, E_n$, which are distributed to n processors P_1, $\ldots$, P_n, respectively, referred to as workers.

(b) The workers run under the BSP model [39], which separates scaling into supersteps. In a superstep, each worker conducts computation of ParBVC^+ or ParBVC^- to refine its own fragment and exchanges updates via messages.

(c) When adding or deleting $|k|$ fragments ($k{>}{-n}$), $|k|$ additional workers are added or $|k|$ existing workers are deleted.

Parallel algorithms. We only present ParBVC^+; ParBVC^- is similar. As opposed to its serial counterpart (Section 3.2), the algorithm conducts *in parallel* (a) the computation of hash values and edge assignments, and (b) edge migration and linear probing for load balancing, by all workers.

Algorithm ParBVC^+. Given a partition $\Pi(n)$ of G placed on a unit circle $\mathcal{C}$, a balance factor ϵ and a number $k > 0$, ParBVC^+ scales out $\Pi(n)$ to an ϵ-balanced partition $\Pi(n{+}k)$. Like BVC^+, it first adds k new fragments on the circle $\mathcal{C}$, maintaining the interval invariant. It then identifies edges that belong to the new fragments by consistent hashing, and migrates them to the corresponding fragments. As opposed to BVC^+, ParBVC^+ does these *in parallel*: for each existing fragment E_i ($1 \leq i \leq n$), its worker P_i identifies and moves out the related edges in E_i. Finally ParBVC^+ balances the resulting partition, *in parallel* via linear probing.

Analysis. We show that ParBVC^+ retains the same bounds on replication and migration cost as BVC^+ (Theorem 3).

(a) Bounds for ParBVC^+. Since ParBVC^+ and BVC^+ use the same hash function for edges, the distribution of edges among fragments is the same for both ParBVC^+ and BVC^+. Moreover, both algorithms maintain the same interval invariant (Lemma 2). Hence the same bounds of Theorem 3 can be deduced for both of them, although ParBVC^+ migrates edges in parallel, while BVC^+ does it sequentially.

(b) Running time. For BVC^+, the migration cost is bounded by $O(|k|\frac{|E|}{n+k})$. For ParBVC^+, the expected running time is in $O(\frac{|E|}{n+k})$, since edge migration from existing fragments to new ones dominates the cost, and ParBVC^+ conducts it in parallel. By Theorem 3, only a small number of edges need to be moved in the linear probing step for rebalancing.

4. A GENERIC SCALING SCHEME

The approximation solution above requires an initial partition that places fragments on a hash ring and satisfies the interval invariant. In practice, however, users often start with a partition computed by a partitioning algorithm VP of their own choice. Is there a method that scales any existing vertex-cut partitioner VP in response to load surges?

We next develop such a generic solution and show that it guarantees minimum migration cost and a relative bound on partition quality (Section 4.1). As proof of concept, we scale two existing vertex-cut partitioners (Section 4.2).

4.1 Dynamic Scaling Scheme

Given a vertex-cut partitioning algorithm VP, we deduce algorithms VP^+ and VP^-. Given a n-partition $\Pi(n) = (E_1, \ldots, E_n)$ generated by VP and an integer $k > -n$, VP^+ and VP^- compute partition $\Pi(n+k)$ for scaling out and in, respectively, depending on whether $k > 0$. To simplify the presentation, we assume *w.l.o.g.* that $\epsilon = 0$ in this section.

Scaling scheme. The scheme computes $\Pi(n{+}k)$ by selecting a minimum number of edges to move, employing VP to re-assign these edges, and retaining the edge assignments of VP as much as possible. This allows us to minimize migration cost and achieve partition quality comparable to VP. More specifically, VP^+ and VP^- work as follows.

Scaling out. From each fragment E_i ($i \in [1, n]$), VP^+ (a) selects a subset $E'_i \subseteq E_i$ of edges such that $|E'_i| = \frac{k|E_i|}{n+k}$, and (b) applies VP to the set $\bigcup_{i=1}^{n} E'_i$ of all selected edges, and obtains a k-partition $(E''_{n+1}, \ldots, E''_{n+k})$. (c) These yield a $(n+k)$-partition $(E_1 \setminus E'_i, \ldots, E_n \setminus E'_n, E''_{n+1}, \ldots, E''_{n+k})$.

That is, it employs the original partitioner VP to re-assign the selected edges. It only moves edges from E_i to the k new fragments, *not* between existing fragments E_i ($i \in [1, n]$).

Scaling in. VP^- randomly selects $|k|$ fragments E_{i_1}, $\ldots$, $E_{i_{|k|}}$ to remove, and then employs VP to reassign edges of $\bigcup_{j=1}^{|k|} E_{i_j}$ to the remaining fragments E_{j_1}, $\ldots$, $E_{j_{n+k}}$.

VP^+ and VP^- incur the minimum migration cost, since they move the minimum number of edges to make the new partition balanced with $\epsilon = 0$. VP^+ only moves edges from original fragments to newly added ones, and VP^- reassigns edges from the removed fragments to the remaining ones. Neither moves edges among existing fragments.

Proposition 4: *Given a balanced partition $\Pi(n)$, the migration cost of* VP^+ *(resp.* VP^-*) is* $O(\frac{k|E|}{n+k})$ *(resp.* $O(\frac{|k||E|}{n})$*) when adding (resp. removing) $|k|$ fragments.* □

Edges selection. We next show that the algorithms also offer relative bounds on replication factor f. Below we focus on VP^+; the analysis of VP^- is similar and simpler.

Observe that VP^+ only selects edges from overfull fragments and moves them to newly added ones. VP^+ uses the following edge selection strategy: from each fragment E_i ($i \in [1, n]$), VP^+ selects $\frac{k}{n+k}|E_i|$ edges from E_i such that the number of vertices on the selected edges is minimum.

We now give an upper bound on the replication factor of VP^+. Denote by τ_i the average vertex degree in fragment E_i.

Proposition 5: *The replication factor after* VP^+ *is at most* $F + k \cdot \frac{k}{n+k} \frac{2|E|}{\mathsf{min}\{\tau_i\}_{i=1}^{n} \cdot |V|}$ *with the edge selection strategy above. Here* $\mathsf{min}\{\tau_i\}_{i=1}^{n}$ *is the minimum average vertex degree of all fragments, and F is the replication factor before scaling.* □

Proof: This is deduced from the following: (a) the replication factor of the original fragments after the scaling is at worst F; (2) the number of vertices on selected edges from fragment E_i is at most $\frac{k}{n+k}\frac{2|E_i|}{\tau_i}$; and (3) each selected vertex can be assigned to at most k new fragments. □

In practice, the replication factor is expected to be better than this upper bound, because (1) when we remove edges from a fragment E_i, its replication factor is decreased and is often smaller than F; and (2) when we use VP to distribute the selected edges, the replication factor of the new fragments is often smaller than the second term in Proposition 5, since each vertex unlikely appears in all new fragments.

4.2 Scaling Stream Partitioners

As case studies, we next scale HDRF [31] and Greedy (Powergraph [15]), two well-known vertex-cut partitioners.

Both partitioning algorithms are *stream-based*, which processes edges in a one-pass fashion. Consider a vertex-cut partition $\Pi(n) = (E_1, \ldots, E_n)$ generated so far. An incoming edge e is assigned to a fragment E_i based on scores $S(e, E_i)(i \in [1, n])$, which aggregates edges assigned to E_i so far. More specifically, edge e is assigned to E_{i^*}, where

$$i^* = \text{argmax}_{i \in \{1,\ldots,n\}} S(e, E_i),$$

i.e., the fragment that maximizes the score. Partitioners HDRF and Greedy use different score functions.

HDRF. We start with HDRF, which favors replicating vertices with relatively large degrees. Given an edge $e = (u, v)$, it computes a score $S(u, v, E_i)$ *w.r.t.* each fragment E_i:

$$S(u, v, E_i) = S_{\text{REP}}(u, v, E_i) + S_{\text{BAL}}(E_i), \tag{2}$$

where $S_{\text{REP}}(u, v, E_i)$ is a *replication score* of e *w.r.t.* E_i and $S_{\text{BAL}}(E_i)$ is a *balance score* of E_i, defined as follows. To replicate vertices with higher degrees first, HDRF defines $S_{\text{REP}}(u, v, E_i) = g(u, v, E_i) + g(v, u, E_i)$, where

$$g(v, u, E_i) = \begin{cases} 1 + \frac{\mathsf{deg}(u)}{\mathsf{deg}(v) + \mathsf{deg}(u)} & \text{if } v \in V_i, \\ 0 & \text{otherwise.} \end{cases}$$

Here $\mathsf{deg}(u)$ and $\mathsf{deg}(v)$ are the degrees of u and v, respectively. Let $\textsc{Maxsize}$ and $\textsc{Minsize}$ be the maximum and minimum size of all fragments when processing edge e, respectively, then the balance score $S_{\text{BAL}}(E_i)$ of e is defined as

$$S_{\text{BAL}}(E_i) = \lambda \frac{\textsc{Maxsize} - |E_i|}{1 + \textsc{Maxsize} - \textsc{Minsize}},$$

where λ is a user-defined parameter that controls the impact of the balance score. HDRF sets the default value of λ as 2.

Edge selection of HDRF. We focus on edge selection for scaling out, since there is no much flexibility for scaling in. A naive method is to randomly select edges from overfull fragments. However, this usually leads to degeneration of partition quality. Instead, we introduce two strategies based on score and timestamp of stream HDRF.

(1) Score based. Intuitively, a larger HDRF score $S(e, E_i)$ of e indicates better locality of e *w.r.t.* fragment E_i. Hence it is natural to move out edges with relatively lower scores. However, we cannot simply use the score assigned to e when it comes in, since it only reflects the fragment information at that moment. Hence for each edge e, we compute a new score $S(e, E_i \setminus \{e\})$ by treating e as a new edge for E_i. Edges with relatively lower new scores are selected for scaling out.

(2) Timestamp based. Intuitively, edges that are processed earlier are more likely to be assigned to "wrong" fragments, since their scores are computed with less information and may not be accurate. In HDRF, $\mathsf{deg}(u)$ and $\mathsf{deg}(v)$ used in the score function cannot be computed in advance and thus are approximated by their partial degrees, *i.e.,* the number of processed edges that are attached to u and v, respectively. The degrees used in the score computation for earlier edges are not as accurate as those of later edges.

This suggests that we revise the assignment of early coming edges and retain the assignment of later ones. Hence when running HDRF, we associate with each edge e a timestamp recording when it is added to its fragment. We select edges with relatively smaller timestamp for scaling out.

Based on these, we deduce HDRF^+ and HDRF^- as follows.

HDRF^+. From each fragment E_i, HDRF^+ selects $\frac{k}{n+k}|E_i|$ edges based on one of the edge selection strategies above. It merges these edges as a new stream and invokes HDRF to assign these edges to the k newly added fragments.

HDRF^-. This case is simpler. HDRF^- randomly selects $|k|$ fragments and merges their edges as a new stream. It then uses HDRF to reassign the edges to the remaining fragments.

As will be demonstrated in Section 5, HDRF^+ and HDRF^- scale partition with quality comparable to re-partitioning the entire graphs by HDRF starting from scratch, while they incur the minimum migration cost (Proposition 4).

Replication factor. We show that with the two simple edge-selection strategies above, HDRF^+ still guarantees bounded replication factor relative to partitioner HDRF.

We use the following notations. Denote by (a) $E'_1, \ldots, E'_n$ the sets of edges selected from partition $(E_1, \ldots, E_n)$ by one of the strategies; (b) $E''_1, \ldots, E''_n$ the edges remaining in the n fragments; and (c) f' and f'' the replication factor of $(E'_1, \ldots, E'_n)$ and $(E''_1, \ldots, E''_n)$, respectively.

Observe that f'' is at least as good as the replication factor of the original $(E_1, \ldots, E_n)$. For the k new fragments, we show that the replication factor is comparable to f'. To simplify the analysis, we adopt $\lambda = 1$ as in [31].

Proposition 6: *The replication factor after* HDRF$^+$ *is bounded by (1)* $f''+\frac{2k^2}{n+k}|E|$ *with the score-based strategy, and (2)* $f'+f''+\frac{k}{n+k}\frac{|E|}{|V|}-\frac{|V_1|}{2\cdot|V|}$ *for timestamp-based when* $\lambda=1$, *where* V_1 *is the number of vertices in the selected edges.* □

Proof: We verify statement (1); the proof for statement (2) is similar. Observe that the replication factor of the resulting partition is the sum of the replication factor of n remaining fragments $\Pi(n)' = (E''_1, \ldots, E''_n)$ and that of the partition $\Pi(k)$ of k new fragments with edges $E'_1, \ldots, E'_n$. The replication factor of $\Pi(n)'$ is at worst f''. From a detailed analysis of the new score $S(e, E_i \setminus \{e\})$ it follows that the replication factor of $\Pi(k)$ is bounded by $\frac{2k^2}{n+k}|E|$. □

Greedy. Greedy is a stream-based partitioner adopted by Powergraph [15]. It can be seen as a special case of HDRF. It also uses Eq. (2) to compute edge scores. It differs from HDRF in that it (a) uses 1 as the default value for λ to balance score; and (b) it does not include the impact of degrees in the replication score and defines $g(v, u, E_i)$ by

$$g(v,u,E_i) = \begin{cases} 1 & \text{if } v \in V_i, \\ 0 & \text{otherwise.} \end{cases}$$

The edge selection strategies for HDRF also work for Greedy. Denote by Greedy$^+$ and Greedy$^-$ the scaling algorithms deduced from Greedy along the same lines. Then the bounds for migration cost and replication factor of HDRF$^+$ and HDRF$^-$ also hold on Greedy$^+$ and Greedy$^-$, respectively.

Parallelization. Following [35], we parallelize HDRF$^+$ and HDRF$^-$ (resp. Greedy$^+$ and Greedy$^-$) in a mini-batch fashion as follows. Each worker maintains a shared state that includes the information of degrees and locations of processed vertices. The edge assignment is conducted in rounds. In a round, each worker handles a small batch of edges *in parallel*, as in HDRF or Greedy; workers communicate with each other at the end of each round to synchronize the shared state. The process terminates when all edges are processed.

5. EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we conducted four sets of experiments to evaluate our scaling algorithms for their (1) efficiency, (2) partition quality, (3) scalability, and (4) impact on the performance of graph analysis tasks.

Experimental setting. We start with the setting.

Datasets. We used three real-life power-law graphs: (a) PLD [26], an undirected graph with 39 million nodes and 623 million edges, in which each node represents a pay-level domain and each edge indicates a hyperlink between a pair of domains; (b) Twitter [20], a social network with 42 million users and 1.5 billion links; and (c) UKWeb [1], a large Web graph with 106 million nodes and 3.7 billion edges.

We also generated synthetic graphs with size up to 440 million vertices and 14 billion edges, to test scalability.

Algorithms. We implemented approximate ParBVC$^-$ and ParBVC$^+$ (Section 3), and parallel HDRF$^+$, HDRF$^-$, Greedy$^+$ and Greedy$^-$ (Section 4), all in C++, compared with the following: (1) CH [17], a consistent-hashing partitioner; in contrast to ParBVC$^+$ and ParBVC$^-$, CH takes edge id as hashing key and hashes fragments to a unit circle; it also uses a virtual-sever method to balance load; (2) 2DHash [44], a widely used hash-based vertex partitioner; (3) Libra [43], a state-of-the-art degree-based hashing algorithm; and (4) stream partitioners HDRF and Greedy (Section 4). Since 2DHash, Libra, HDRF and Greedy do not support dynamic scaling, we mainly consider their partition quality.

To evaluate the effectiveness of our edge selection strategies of our generic scaling scheme, we implemented variants of HDRF$^+$ and Greedy$^+$, also in C++. Denote by HDRF$^+_{\mathrm{S}}$ and HDRF$^+_{\mathrm{T}}$ the implementations of HDRF$^+$ with edge selection based on score and timestamp, respectively; similarly for Greedy$^+_{\mathrm{S}}$ and Greedy$^+_{\mathrm{T}}$. The results reported for HDRF$^+$ and Greedy$^+$ take the average of two strategies. We also implemented a strategy that randomly chooses edges for scaling out, denoted by HDRF$^+_{\mathrm{R}}$ and Greedy$^+_{\mathrm{R}}$, respectively. We parallelized the algorithms as described in Section 4.2. The mini-batch size is set to 256 by default.

The experiments were conducted on GRAPE, a parallel graph processing engine [11], deployed on an HPC cluster of up to 36 machines, each with 12 cores powered by Intel Xeon 2.2GHz and 128GB memory, with a 10Gbps link between machines. Each experiment was repeated 5 times and the average is reported here.

Experimental results. We next report our findings.

Exp-1: Efficiency. We first evaluated the scaling time and migration cost of the algorithms. For ParBVC$^+$ and ParBVC$^-$, we set balance factor $\epsilon = 0.1$; the other algorithms do not take ϵ as a hard constraint on load balance.

Varying k. Fixing $n = 96$, we varied k from 20 to 100 (resp. 10 to 50) for scaling out (resp. in). We find the following.

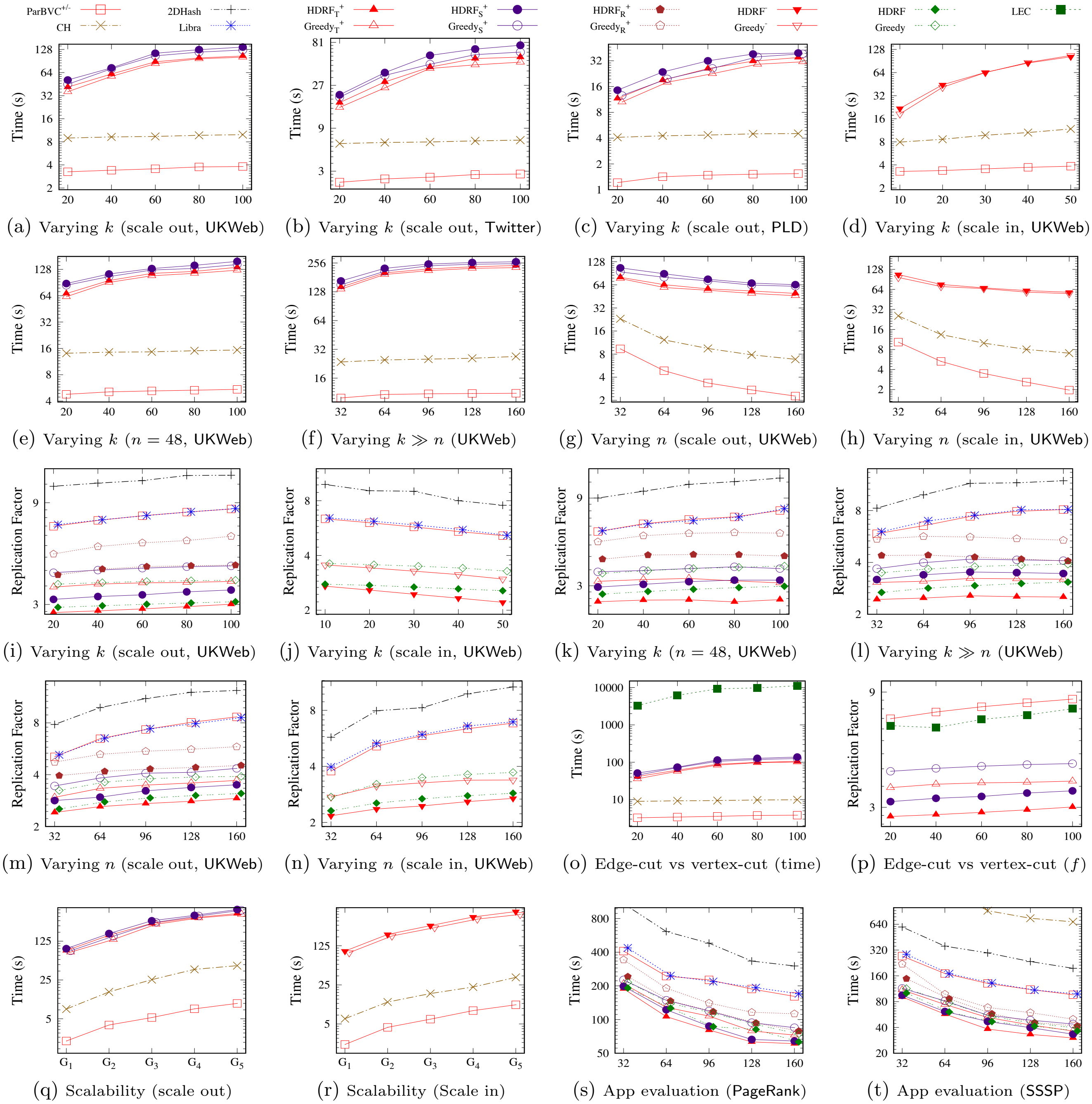
(1) As shown in Figures 5(a)-5(c), ParBVC$^+$ performs the best in time efficiency. It outperforms CH, HDRF$^+$ and Greedy$^+$ by 2.7, 20.3 and 18.5 times, respectively, up to 3.4, 36.1 and 33.1 times. All algorithms take longer when k gets larger, as expected. However, ParBVC$^+$ and CH are less sensitive to the change of k than HDRF$^+$ and Greedy$^+$, since they incur less synchronization overhead during scaling.

(2) 2DHash, Libra, HDRF and Greedy do not support dynamic scaling, and have to re-partition graphs. ParBVC$^+$ is 8.9, 7.4, 926.5 and 763.6 times faster than these methods, respectively, up to 13.1, 11.2, 1406.8 and 1224.8 times (not shown). This is because the re-partitioning methods need to (a) recompute edge assignments, and (b) move most edges (their migration cost is 2.9 times larger than ParBVC$^+$).

(3) The results for scaling in are consistent with scaling out. As shown in Fig. 5(d), on average ParBVC$^-$ outperforms CH, HDRF$^-$ and Greedy$^-$ on UKWeb by 2.7, 18.6 and 17.4 times, respectively, up to 3.1, 26.7 and 27.8 times. The results on Twitter and PLD are consistent (not shown).

(4) CH incurs larger migration cost, on average 1.1 (resp. 1.2) times more than ParBVC$^+$ (resp. ParBVC$^-$). It is 2.7 (resp. 2.7) times slower than ParBVC$^+$ (resp. ParBVC$^-$) (see (1)), since CH generates unbalanced partitions (Exp-2), which yield stragglers and slow down scaling. This verifies the effectiveness of our fragment placement strategy (Section 3.2).

(5) HDRF$^+$ and Greedy$^+$ (resp. HDRF$^-$ and Greedy$^-$) incur minimum migration cost. These are 1.37 and 1.37 (resp. 1.40 and 1.40) times better than ParBVC$^+$ (resp. ParBVC$^-$) on average, respectively. Nevertheless, they are slower than ParBVC$^+$ and ParBVC$^-$. This is because during scaling they need to (a) compute the score *w.r.t.* all fragments to decide



(a) Varying k (scale out, UKWeb) (b) Varying k (scale out, Twitter) (c) Varying k (scale out, PLD) (d) Varying k (scale in, UKWeb)

(e) Varying k ($n = 48$, UKWeb) (f) Varying $k \gg n$ (UKWeb) (g) Varying n (scale out, UKWeb) (h) Varying n (scale in, UKWeb)

(i) Varying k (scale out, UKWeb) (j) Varying k (scale in, UKWeb) (k) Varying k ($n = 48$, UKWeb) (l) Varying $k \gg n$ (UKWeb)

(m) Varying n (scale out, UKWeb) (n) Varying n (scale in, UKWeb) (o) Edge-cut vs vertex-cut (time) (p) Edge-cut vs vertex-cut (f)

(q) Scalability (scale out) (r) Scalability (Scale in) (s) App evaluation (PageRank) (t) App evaluation (SSSP)

Figure 5: Performance Evaluation

the assignment of an edge, and (b) synchronize shared state. (6) HDRF$^+$ and Greedy$^+$ are on average 53.1 and 46.1 times faster than HDRF and Greedy, respectively. However, they take longer than hash-based CH for the same reason given in (5) above; similarly for HDRF$^-$ and Greedy$^-$.

(7) We also evaluated the impact of different initial partition numbers n. Fixing $n = 48$, we varied k from 20 to 100 (resp. 10 to 40) for scaling out (resp. scaling in). As shown in Fig. 5(e) on UKWeb, its scaling-out performance pattern is consistent with Fig. 5(a) when $n = 96$. The (scaling-in) results on Twitter and PLD are consistent (not shown).

Varying $k \gg n$. Fixing $n = 32$, we varied k from 32 to 160 on UKWeb to evaluate scaling-out algorithms when $k \gg n$. As shown in Fig. 5(f), the results are consistent with Figures 5(a)–5(c). (a) When k gets larger, all algorithms take longer. (b) ParBVC$^+$ is on average 2.4, 20.4 and 19.6 times faster than CH, HDRF$^+$ and Greedy$^+$, respectively. (c) HDRF$^+$ and Greedy$^+$ beat HDRF and Greedy by 16.9 and 15.4 times, respectively. (d) ParBVC$^+$ beats re-partitioning methods 2DHash, Libra, HDRF and Greedy by 10.3, 8.4, 348.4 and 302.5 times, respectively. (e) ParBVC$^+$ and CH are not as sensitive to k as HDRF$^+$ and Greedy$^+$, since they are easy to parallelize and incur less synchronization cost. The (scaling in) results on Twitter and PLD are consistent.

Varying n. Fixing $k/n = 1/3$, we varied n from 32 to 160 on UKWeb. The results on Twitter and PLD are consistent.

As shown in Fig. 5(g), (1) ParBVC$^+$ beats CH, HDRF$^+$ and Greedy$^+$ by 2.8, 18.5 and 17.4 times on average, respectively. (2) HDRF$^+$ and Greedy$^+$ are 59.4 and 54.9 times faster than HDRF and Greedy, respectively (HDRF and Greedy are not shown). (3) When n is larger, all algorithms take less time. (4) HDRF$^+$ and Greedy$^+$ are not very sensitive to n as when n increases, so does their communication cost. ParBVC$^+$ and CH have better *parallel scalability*: they are 4.3 and 3.4 times faster when n varies from 32 to 160, respectively. This is because (a) consistent hashing reduces migration cost; and (b) the hash computation can be efficiently parallelized.

As shown in Fig. 5(h), the results for scaling in are consistent with Fig. 5(g). In particular, ParBVC$^-$ outperforms CH, HDRF$^-$ and Greedy$^-$ by 2.9, 19.5 and 18.4 times on average, respectively. When n increases from 32 to 160, ParBVC$^-$ and CH are 5.3 and 3.6 times faster, respectively.

Exp-2: Partition quality. We next evaluated (a) the replication factor f, and (b) balance factor ϵ. We also evaluated (c) the effectiveness of the edge selection strategies (Section 4.2) for stream partitioners. We used UKWeb; the results on Twitter and PLD are consistent (not shown).

Replication factor. In the same setting as Exp-1, Figures 5(i)-5(n) report replication factors of the algorithms.

(1) Varying k. As shown in Fig. 5(i), the replication factors of all algorithms for scaling out become larger when n or k increases. Moreover, observe the following.

(a) HDRF$^+_{\mathrm{T}}$ has the best replication factor among the scaling out algorithms over all datasets. On average, it outperforms HDRF$^+_{\mathrm{S}}$, Greedy$^+_{\mathrm{T}}$, Greedy$^+_{\mathrm{S}}$, ParBVC$^+$ and CH by 1.1, 1.2, 1.4, 1.8 and 5.9 times, respectively, up to 1.2, 1.3, 1.6, 2.8 and 10.4 times. When $k = 100$, HDRF$^+_{\mathrm{T}}$ beats these algorithms by 1.2, 1.3, 1.5, 2.7 and 10.4 times, respectively. That is, HDRF$^+_{\mathrm{T}}$ performs well even when the configuration is changed substantially (when $k > n$). This is because HDRF$^+_{\mathrm{T}}$ (i) retains data locality as HDRF by assigning edges to where their vertices are located and cutting vertices with large degrees; and (ii) rectifies "bad edge assignments" by reassigning edges based on the information of graphs.

(b) HDRF$^+_{\mathrm{T}}$ also does better than re-partitioning algorithms Libra, 2DHash and Greedy on average by 1.8, 2.5 and 1.3 times, respectively. It is even better than HDRF in most cases, which re-partitions graphs starting from scratch. This is because (i) early incoming edges incur bad locality since their assignments by HDRF use little information of graphs; and (ii) HDRF$^+_{\mathrm{T}}$ utilizes more information, *e.g.,* the degrees of processed vertices, and rectifies the "bad" assignments when scaling out. This shows that our generic scaling scheme does not come with a price of partition quality.

(c) The replication factor of CH is on average larger than 20 (not shown). ParBVC$^+$ and Libra have comparable replication factors, since both of them employ a degree-based approach and hence retain good locality. On average, they outperform other hash-based algorithms CH and 2DHash by 3.4 and 1.4 times, respectively, up to 3.8 and 1.6 times.

(d) The results of scaling in are consistent. As shown in Fig. 5(j), on average HDRF$^-$ outperforms Greedy$^-$, ParBVC$^-$, CH, Libra, 2DHash, HDRF and Greedy by 1.3, 2.3, 8.8, 2.4, 3.5, 1.1 and 1.4 times, respectively. As opposed to scaling out, the replication factors of all algorithms for scaling in decrease when k increases.

(e) The timestamp based edge selection strategy works the best. On average the replication factor of HDRF$^+_{\mathrm{T}}$ (resp. Greedy$^+_{\mathrm{T}}$) is 1.1 and 1.4 (resp. 1.1 and 1.2) times better than HDRF$^+_{\mathrm{S}}$ and HDRF$^+_{\mathrm{R}}$ (resp. Greedy$^+_{\mathrm{S}}$ and Greedy$^+_{\mathrm{R}}$).

Table 1: Balance factor

Alg/Dataset	UKWeb	Twitter	PLD
ParBVC$^+$	0.1	0.1	0.1
HDRF$^+$	0.003	< 0.001	< 0.001
HDRF	0.043	< 0.001	< 0.001
Greedy$^+$	0.085	0.013	0.023
Greedy	0.503	0.201	0.119
CH	3.21	3.06	3.15
Libra	0.012	0.008	0.011
2DHash	1.13	1.16	1.04

(f) As in Exp-1, we also tested the case when $n = 48$. As shown in Fig. 5(k), the results are consistent with Fig. 5(i). This shows that our algorithms have a stable performance pattern regardless of the initial partition number n.

(2) Varying $k \gg n$. As in Exp-1, we also set $n = 32$ and varied k from 32 to 160. As shown in Fig. 5(l), the replication factors of all scaling-out algorithms except the stream-based variants, *i.e.,* HDRF$^+$, HDRF$^+_{\mathrm{R}}$, Greedy$^+$, and Greedy$^+_{\mathrm{R}}$, increase when k gets larger. (a) When k varies from 32 to 160, the replication factor of HDRF$^+$ increases from 2.8 to 3.0. It beats Greedy$^+$, ParBVC$^+$, CH, Libra and 2DHash by 1.2, 2.5, 8.9, 2.5 and 3.7 times, respectively. (b) The replication factors of HDRF$^+$, HDRF$^+_{\mathrm{R}}$, Greedy$^+$ and Greedy$^+_{\mathrm{R}}$ get slightly smaller when $k > 96$. This is because (i) when $k > 96$, most of edges have to be moved; (ii) these algorithms rectify edges assignment during scaling. (c) HDRF$^+$ (resp. Greedy$^+$) has comparable replication factor to HDRF (resp. Greedy).

(3) Varying n. Fixing $k/n = 1/3$, as shown in Figures 5(m) and 5(n), the replication factors of all algorithms become larger when n increases. (a) When n varies from 32 to 160, the replication factor of HDRF$^+$ varies from 2.6 to 3.2. On average it beats Greedy$^+$, ParBVC$^+$, CH, Libra and 2DHash by 1.3, 2.6, 9.0, 2.6 and 3.9 times, respectively. (b) The results for scaling in are consistent. On average, HDRF$^-$ beats Greedy$^-$, ParBVC$^-$, CH, Libra, 2DHash and Greedy by 1.3, 2.3, 7.7, 2.3, 3.4 and 1.4 times, respectively. (c) HDRF$^+$ and HDRF$^-$ achieve replication factors comparable to HDRF.

Balance factor. We next evaluated the balance factor. Table 1 shows the balance factors for scaling out when $n = 96$ and $k = 40$ on average over the three real-life graphs.

(1) HDRF$^+$ does the best in most cases. Its balance factor is as small as 0.003. The balance factor of Greedy$^+$ varies from 0.001 to 0.095. It is not as balanced as HDRF$^+$ since (a) it puts less weight on balance score than HDRF$^+$ (see Section 4.2) and (b) it may assign edges based on high-degree vertices and cut vertices with relatively low degree. Even so, Greedy$^+$ still does better than Greedy in balance.

(2) ParBVC$^+$ enforces a user-defined balance factor $\epsilon = 0.1$ by its rebalancing stage (Section 3.2). In contrast, CH and 2DHash have ϵ as large as 3.46 and 1.16, respectively. Libra has a smaller ϵ, but it is not efficient as ParBVC$^+$ (Exp-1).

(3) The balance factor of CH is much worse than ParBVC$^+$, from 23.1 to 34.6 times, since it uses hash function to place fragments and its virtual-server strategy does not improve balance much when $m \gg n$, *i.e.,* when there are far more edges than fragments as found in our setting. This verifies the benefit of our fragment placement strategy.

(4) The results for scaling in are consistent (not shown). HDRF$^-$ achieves the best balance factor in most cases, while ParBVC$^-$ guarantees a user-defined balance factor.

We also evaluated the impact of user-imposed balance factor by setting $\epsilon = 0.1$ and 0.3 for ParBVC$^+$ and ParBVC$^-$

(not shown). (1) With larger ϵ, both get slightly better replication factors f. (2) Smaller ϵ incurs larger migration cost. When $n = 96$ and $k = 40$, the migration cost of ParBVC$^+$ over UKWeb increases from $0.26|E|$ to $0.34|E|$ when ϵ varies from 0.3 to 0.1. The results of ParBVC$^-$ are consistent.

Edge-cut partitions. We also compared with LEC [32], a scaling algorithm for edge-cut partitions. Following [47], we deduced a vertex-cut partition from an edge-cut partition, and computed its replication factor accordingly.

The results on UKWeb are shown in Figures 5(o) and 5(p). (1) When k or n increases, the replication factor of LEC also increases. When k varies from 20 to 100 (resp. 10 to 50), the replication factor of LEC varies from 6.4 to 7.7 (resp. 4.9 to 5.9). It is slight better than ParBVC$^+$ (resp. ParBVC$^-$), but is much worse than HDRF$^+$ and Greedy$^+$ (resp. HDRF$^-$ and Greedy$^-$). On average the replication factor of LEC is 2.3 (resp. 2.2) times larger than HDRF$^+$ (resp. HDRF$^-$). (2) Its scaling time is much larger than our algorithms. On average it is 2188.6, 87.6 and 93.8 times slower than ParBVC$^+$, HDRF$^+$ and Greedy$^+$, respectively. This is because LEC migrates vertexes and edges greedily, and is hard to parallelize. (3) Edge balancing of LEC is much worse than our algorithms, varying from 0.8 to 1.7, since LEC focuses on vertex balance only. Due to its imbalance, graph processing takes longer on partitions computed by LEC. On average, PageRank with LEC is 1.5, 3.7 and 2.9 times slower than with ParBVC$^+$, HDRF$^+$ and Greedy$^+$, respectively.

Exp-3: Scalability. Fixing n=320 and k=110, we varied the size $|G|$=$(|V|, |E|)$ of synthetic graphs from (88M,2.8B) to (440M,14B) to test the scalability of the algorithms.

As shown in Fig. 5(q)-5(r), (1) ParBVC$^+$ and ParBVC$^-$ scale well with $|G|$. When G varies from (88M, 2.8B) to (440M, 14B), ParBVC$^+$ (resp. ParBVC$^-$) takes 1.99s to 9.45s (resp. 2.15s to 11.37s), almost linear with $|G|$. On average, ParBVC$^+$ beats CH, HDRF$^+$ and Greedy$^+$ by 4.5, 46.1 and 43.3 times, respectively. ParBVC$^-$ beats CH, HDRF$^-$ and Greedy$^-$ by 2.9, 46.6 and 42.9 times, respectively. (2) CH scales almost as well as ParBVC$^+$ and ParBVC$^-$, since they all employ consistent hashing. (3) Although the efficiency of HDRF$^+$ and Greedy$^+$ is not as good as that of ParBVC$^+$, they scale well; their computation and communication costs are linear with $|G|$. When $|G|$ increases 5 times, running time of HDRF$^+$ (resp. Greedy$^+$) increases 4.9 (resp. 5.1) times.

Exp-4: Impact on graph analysis tasks. To further evaluate the effectiveness of our scaling algorithms, we tested the execution time and communication cost of two standard graph analysis tasks, PageRank and SSSP (single source shortest path), over the partitions obtained by our scaling algorithms. Fixing $k/n = 1/3$ and varying n from 32 to 160, we report their performance on UKWeb; the results on Twitter and PLD are consistent (not shown).

(1) As shown in Figures 5(s)-5(t), (a) when n gets larger, PageRank and SSSP get faster on UKWeb with all partitioning algorithms. (b) Pagerank (resp. SSSP) with HDRF$^+$ is 1.3, 1.3, 2.5, 5.3, 2.6 and 16.9 (resp. 1.2, 1.3, 3.0, 6.4, 2.8 and 22.9) times faster than with Greedy$^+$, Greedy, ParBVC$^+$, 2DHash, Libra and CH on average, respectively. (c) ParBVC$^+$ and Libra have similar effectiveness since they have comparable replication and balance factors. On average, PageRank and SSSP with these two are 4.5 and 4.9 times faster than with the other hash-based partitioners, respectively.

(2) Pagerank (resp. SSSP) with HDRF$^+$ incurs less communication costs (not shown), and ships 71.9%, 73.4%, 28.5%, 20.4%, 28.1% and 11.3% (resp. 74.4%, 72.9%, 26.7%, 17.3%, 25.9% and 7.5%) of data shipped with Greedy$^+$, Greedy, ParBVC$^+$, 2DHash, Libra and CH on average, respectively.

Summary. We find the following. (1) Algorithms ParBVC$^+$ and ParBVC$^-$ perform the best in efficiency. ParBVC$^+$ outperforms CH, Libra, 2DHash, HDRF$^+$ and Greedy$^+$ by 2.7, 8.7, 10.8, 20.4 and 18.9 times on average. When n=96 and k=100, it is 2.6, 7.1, 8.4, 26.5 and 24.2 times faster. ParBVC$^-$ is 2.8, 10.3, 12.2, 18.5 and 17.9 times faster than CH, Libra, 2DHash, HDRF$^-$ and Greedy$^-$, respectively. Algorithms HDRF$^+$ and Greedy$^+$ (resp. HDRF$^-$ and Greedy$^-$) are 43.8 and 40.1 times (resp. 43.7 and 41.2) faster than HDRF and Greedy on average, respectively, up to 114.7 and 106.6 times (resp. 129.8 and 132.3). (2) Our algorithms achieve good partition quality. In the same setting as (1), ParBVC$^+$ (resp. ParBVC$^-$) does better than hash-based CH and 2DHash in replication factor by 3.37 and 1.45 (resp. 3.56 and 1.52) times on average, and 17.7 (resp. 24.6) times in balance factor on average. HDRF$^+$ and HDRF$^-$ (resp. Greedy$^+$ and Greedy$^-$) have replication and balance factors comparable to re-partitioning with HDRF (resp. Greedy). HDRF$^+$ (resp. HDRF$^-$) does even better than ParBVC$^+$ (resp. ParBVC$^-$) in partition quality, but not as fast. (4) Our algorithms have stable performance and scale well with large n, k and graphs. On graphs with 440 million vertices and 14 billion edges, ParBVC$^+$, HDRF$^+$ and Greedy$^+$ (resp. ParBVC$^-$, HDRF$^-$ and Greedy$^-$) take 9.45s, 427.2s and 413.5s (resp. 11.37s, 490.6s and 453.8s), when n=320 and k>$\frac{n}{3}$. (5) Graph analysis tasks work well with partitions generated by our scaling algorithms. PageRank (resp. SSSP) over HDRF$^+$ is on average 4.9 (resp. 6.3) times faster. Moreover, PageRank (resp. SSSP) with HDRF$^+$ ships 38.9% (resp. 37.5%) data shipped by the others on average.

6. CONCLUSION

To the best of our knowledge, this work is a first systematic study of dynamic scaling for parallel graph computations. We have provided (a) the complexity of the problem and its dominating factor, (b) parallel approximate algorithms with provable bounds on migration cost and partition quality, and (c) the first generic scheme for scaling existing vertex partitioners with (relative) bounds. Our empirical study has verified that the solutions are promising.

One topic for future work is to adapt the methods to edge-cut and improve the bounds. Another topic is to study online scaling, to adjust partitions in response to load surges without interrupting ongoing computations.

Acknowledgments. The authors are supported in part by ERC 652976, Royal Society Wolfson Research Merit Award WRM/R1/180014, 973 2014CB340302, NSFC 61421003, EPSRC EP/M025268/1, Shenzhen Institute of Computing Sciences, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Lu is also supported in part by NSFC 61602023. Liu is also supported in part by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), ESPRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics. The authors thank Lihang Fan, Ziyan Han, Jingbo Xu and Wenyuan Yu for help with the experiments.

7. REFERENCES

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

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

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

[4] J. W. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *IPTPS*, pages 80–87, 2003.

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

[6] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal. Dynamic scaling of Web applications in a virtualized cloud computing environment. In *ICEBE*, pages 281–286, 2009.

[7] C. Curino, E. Jones, Y. Zhang, E. Wu, and S. Madden. Relational cloud: The case for a database service. *New England Database Summit*, pages 1–6, 2010.

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

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[10] D. P. Dubhashi and A. Panconesi. *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press, 2009.

[11] W. Fan, Y. Wu, J. Xu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing Sequential Graph Computations. In *SIGMOD*, pages 495–510, 2017.

[12] K. Fernandes, R. Melhem, and M. Hammoud. Dynamic elasticity for distributed graph analytics. In *CloudCom*, pages 145–148. IEEE, 2018.

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

[14] O. Goldschmidt and D. S. Hochbaum. A polynomial algorithm for the k-cut problem for fixed k. *Math. Oper. Res.*, 19(1):24–37, 1994.

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

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

[17] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, pages 654–663, 1997.

[18] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA*, 2004.

[19] K. Kenthapadi and G. S. Manku. Decentralized algorithms using both local and random probes for P2P load balancing. In *SPAA*, 2005.

[20] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[21] W. Lang and J. M. Patel. Energy management for MapReduce clusters. *PVLDB*, 3(1):129–139, 2010.

[22] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of Hadoop clusters. *Operating Systems Review*, 44(1):61–65, 2010.

[23] H. Li and S. Venugopal. Efficient node bootstrapping for decentralised shared-nothing key-value stores. In *Middleware*, pages 348–367, 2013.

[24] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *PODC*, pages 183–192, 2002.

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

[26] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. Graph structure in the Web — revisited: A trick of the heavy tail. In *WWW*, 2014.

[27] V. Mirrokni, M. Thorup, and M. Zadimoghaddam. Consistent hashing with bounded loads. In *SODA*, pages 587–604, 2018.

[28] M. Naor and U. Wieder. Novel architectures for P2P applications: The continuous-discrete approach. *ACM Trans. Algorithms*, 3(3):34, 2007.

[29] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, 2013.

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

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

[32] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *SIGCOMM*, pages 375–386, 2010.

[33] M. Raab and A. Steger. "Balls into bins" - A simple and tight analysis. In *RANDOM'98*, pages 159–170, 1998.

[34] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[35] H. P. Sajjad, A. H. Payberah, F. Rahimian, V. Vlassov, and S. Haridi. Boosting vertex-cut partitioning for streaming graphs. In *BigData Congress*, 2016.

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

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

[38] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

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

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

[41] C. Walshaw, M. Cross, and M. G. Everett. Parallel

dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, 1997.

[42] W. Wang, H. Chen, and X. Chen. An availability-aware virtual machine placement approach for dynamic scaling of cloud applications. In *UIC/ATC*, pages 509–516, 2012.

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

[44] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on Spark. In *GRADES*, page 2, 2013.

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

[46] L. Yu and Z. Cai. Dynamic scaling of virtual clusters with bandwidth guarantee in cloud datacenters. In *INFOCOM*, pages 1–9, 2016.

[47] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li. Graph edge partitioning via neighborhood heuristic. In *KDD*, pages 605–614, 2017.

[48] A. Zheng, A. Labrinidis, and P. K. Chrysanthis. Planar: Parallel lightweight architecture-aware adaptive graph repartitioning. In *ICDE*, pages 121–132, 2016.