Dynamic Scaling for Parallel Graph Computations

Muyang Liu

Master of Science by Research

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

This thesis 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.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my principle supervisor, Professor Wenfei Fan, for his guidence, trust, encouragement and support. He introduced me to database theory, gave me advice on my career and trained me in every aspect of doing research. His talent, self-discipline, passion for research and hard work inspired me to keep working on challenging problems. Without his supervision and unfailing support, this thesis would not have been possible.

I am very grateful to my collaborators, Ruochun Jin, Yuanhao Li, Dr. Ping Lu, Dr. Chao Tian, Dr. Jingbo Xu, Ruiqi Xu, Dr. Qiang Yin and Dr. Wenyuan Yu, who supported me with their constructive advice and valuable experience. I feel lucky to work with these brilliant and passionate researchers.

I would like to thank the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics. This thesis is supported, in part, by its scholarship.

I would also like to express my heartfelt thanks to my fellow colleagues in the Database group and Pervasve Parallelism project, for the helpful seminars, constructive discussions and all the fun memories I have had through the last year.

Finally, I would like to thank my parents and Lai, for their unconditional support and love. My heartfelt gratitude also goes to my friends, whom I consider part of my family, for their continuous help: Mahesh Dananjaya, Shangmin Guo, Wenbin Hu, Yiyun Jin, Shuyao Li, Siting Lu, Zezhong Wang, Yan Yang and Hao Zheng.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Muyang Liu*)

Table of Contents

Chapter 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 [Chieu et al., 2009].

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.

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. 1.1 (a). It has two types of nodes: user nodes $u_1, \ldots, u_6$ and product nodes $p_1, \ldots, p_5$. In Fig. 1.1 (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

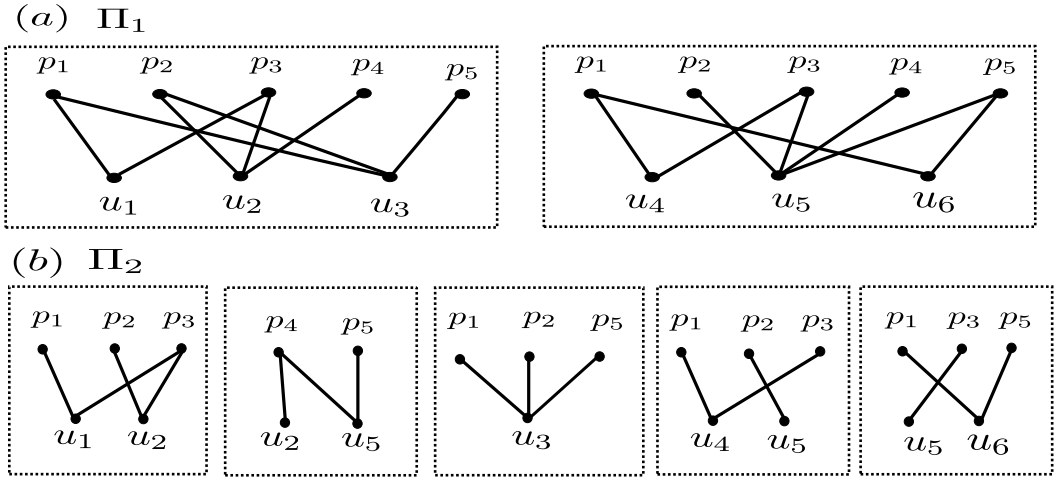


Figure 1.1: Partitions of a graph

too far from the average. Imbalanced partitions often lead to skewness and stragglers, which slow down computations. For $\varepsilon \geq 0$, a partition is ε-*balanced* if each fragment is bounded by $\lceil(1+\varepsilon)|E|/n\rceil$. For Π_1, ε = 0 since the size of each part is at most $\lceil(1+\varepsilon)|E|/2\rceil = 8$. (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 $\varepsilon = 0$, the size of each fragment is at most 4. Such a partition Π_2 is shown in Fig. 1.1 (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 interact with each other, *e.g.,* when $\varepsilon = 0$, (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 [Chieu et al., 2009, Yu and Cai, 2016, Wang et al., 2012, Nguyen et al., 2013, Pujol et al., 2010, Vaquero et al., 2014], 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 [Xu et al., 2014, Shang and Yu, 2013, Schloegel et al., 1997, Walshaw et al., 1997, Huang and Abadi, 2016, Nicoara et al., 2015, Zheng et al., 2016, Dai et al., 2017, Vaquero et al., 2014] 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 thesis (1) formalizes the dynamic scaling problem and establishes its complexity (Chapter 3); (2) provides an approximation solution with bounds on replication factor and migration cost (Chapter 4); and (3) proposes a generic scheme to scale existing vertex-cut partitioners with low migration cost and relative bounds on partition quality (Chapter 5).

(4) Experimental study (Chapter 6). 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 systematic study of dynamic scaling, from complexity and approximation to scaling of existing partitioners. All proofs of the results are in Appendix A.

Chapter 2

Related Work

We summarize the related work as follows.

Graph partitioning. Vertex-cut was proposed in [Gonzalez et al., 2012]. It was shown in [Bourse et al., 2014] 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 [Zhang et al., 2017]. 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 [Chen et al., 2015]. 2DHash [Xin et al., 2013] mitigates this problem by maintaining a $2\sqrt{n}-1$ bound on f, where n is the number of fragments. Degree-based hash partitioning [Xie et al., 2014] assigns edges based on vertex degrees and favors cutting vertices with relatively large degrees. HDRF [Petroni et al., 2015] also replicates (or cuts) high-degree vertices in streaming partition. Apart from these, several heuristics were developed, *e.g.,* [Chen et al., 2015, Margo and Seltzer, 2015, Zhang et al., 2017].

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 [Goldschmidt and Hochbaum, 1994]. This is not implied by the results of [Bourse et al., 2014, Zhang et al., 2017], 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 [Xie et al., 2014] by only a small factor, a small price for balancing, which is not guaranteed by [Xin et al., 2013, Xie et al., 2014, Petroni et al., 2015, Margo and Seltzer, 2015].

(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 [Xin et al., 2013, Gonzalez et al., 2012, Xie et al., 2014, Petroni et al., 2015, Bourse et al., 2014, Zhang et al., 2017].

Consistent hashing. The method was proposed in [Karger et al., 1997] to reduce the movement of hashed clients when the size of hash table changes (see Section 4.1). As shown in [Raab and Steger, 1998, Mirrokni et al., 2018], when there are far more clients than servers as in real-life dynamic scaling, simple consistent hashing [Karger et al., 1997] suffers from imbalanced load. In [Mirrokni et al., 2018], 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 [Ratnasamy et al., 2001] and Chord [Stoica et al., 2001]. 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 [Ratnasamy et al., 2001, Naor and Wieder, 2007, Malkhi et al., 2002, Karger and Ruhl, 2004, Kenthapadi and Manku, 2005] for adding or removing servers (analogous to fragments) in DHT, and [DeCandia et al., 2007, Li and Venugopal, 2013] for balancing the workload of servers in DHT. When adding a new server, CAN [Ratnasamy et al., 2001] 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 [Naor and Wieder, 2007, Malkhi et al., 2002] to handle server removal, and multiple-choice algorithms were used in [Naor and Wieder, 2007, Kenthapadi and Manku, 2005] to add servers. Servers are evenly distributed over a unit circle for load balancing [DeCandia et al., 2007]. Upper and lower bounds for workload are used to guide interval adjustments [DeCandia et al., 2007].

Our work differs from the prior work in the following.

(1) In contrast to [Karger et al., 1997, Mirrokni et al., 2018] 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 [Mirrokni et al., 2018].

(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 [Ratnasamy et al., 2001, Naor and Wieder, 2007]; (b) we define an order in which fragments are removed; and (c) we add or remove fragments, but do not move fragments as in [Naor and Wieder, 2007, Malkhi et al., 2002, Karger and Ruhl, 2004]. These help us guarantee provable bounds on load balance, replication factor and migration cost.

(3) We integrate a degree-based approach [Xie et al., 2014] 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 [Chieu et al., 2009, Yu and Cai, 2016, Wang et al., 2012, Nguyen et al., 2013], or how to reduce energy consumption when workload is low [Lang and Patel, 2010, Leverich and Kozyrakis, 2010]. For cloud computing, [Chieu et al., 2009] allocates VMs based on thresholds of virtual setting. AGILE [Nguyen et al., 2013], a resource scaling system, assigns new VMs based on a resource demand predictor. Adjustment of VMs was studied in [Nguyen et al., 2013] under constraints on communication costs. An algorithm for allocating VMs was given in [Wang et al., 2012] in a hierarchically structured cloud. In a tree-structured virtual network, [Yu and Cai, 2016] considered how to allocate VMs to meet bandwidth requirements. For energy management, a covering set strategy [Leverich and Kozyrakis, 2010] and an all-in strategy [Lang and Patel, 2010] were proposed to select power-down nodes when the utilization is low.

The scaling problems studied in the prior work differ from $\mathsf{DS}(\varepsilon, f, m)$ (Chapter 3) in that it does not consider graph partitioning, not to mention its three objectives (ε, f, m).

Closer to this work are [Pujol et al., 2010, Vaquero et al., 2014, Curino et al., 2010], which study graph partitioning in dynamic scaling; these focus on edge-cut partitioning. A greedy heuristic was developed in [Pujol et al., 2010] to migrate vertices when scaling; [Vaquero et al., 2014] randomly picks vertices based on a given probability, and moves the vertices to other fragments in response to changes to the graphs; and [Curino et al., 2010] adopts a lazy strategy: when a worker is added, necessary

vertices are moved to it only when the worker processes a query.

This work differs from [Pujol et al., 2010, Vaquero et al., 2014, Curino et al., 2010] in the following. (a) We study scaling with vertex-cut partition, which is not yet well studied, as opposed to edge-cut. (b) None of [Pujol et al., 2010, Vaquero et al., 2014, Curino et al., 2010] guarantees partition quality as we do. In particular, [Curino et al., 2010] accumulates vertices at new workers and is not load balanced. (c) We propose a generic scheme to scale existing vertex partitioners with (relative) bounds on migration cost and partition quality. No previous work has studied this.

Chapter 3

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 $\varepsilon \geq 0$, $\Pi(n)$ is called ε-*balanced* if

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

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, ..., 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 $\mathsf{DS}(\varepsilon, f, m)$ or simply DS.

To identify the impact of the three criteria on the complexity, we also study three variants of $\mathsf{DS}(\varepsilon, f, m)$, when one of the three criteria is dropped. Denote by $\mathsf{DS}(f, m)$, $\mathsf{DS}(\varepsilon, m)$ and $\mathsf{DS}(\varepsilon, f)$ 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 $\mathsf{DS}(\varepsilon, f, m)$ is NP-complete. We show that the intractability is quite robust: it remains NP-hard as long as the replication factor is one of the optimization goals, even when k is a constant, *i.e.,* the number of processors added or removed is predefined and fixed.

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

(2) $\mathsf{DS}(\varepsilon, m)$ *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 $\mathsf{DS}(\varepsilon, f)$.

We verify the NP-hardness of DS and $\mathsf{DS}(\varepsilon, f)$ by reduction from the 3-partition problem [Andreev and Racke, 2006], and $\mathsf{DS}(f, m)$ by reduction from the maximal clique problem (cf. [Garey and Johnson, 1979]). The reductions are constructed with constant k (see Appendix A.1 for proof).

(2) For $\mathsf{DS}(\varepsilon, m)$, 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 when both n and k are constants, we first show that there exists a partition such that its replication factor is minimal, and the number of cut nodes is bounded by a constant, where cut nodes are the ones that appear in more than one fragment. Based on this property, we give a PTIME algorithm for $\mathsf{DS}(f, m)$ with $m = \infty$: (a) enumerate all possible sets of cut nodes; (b) for each set S of cut nodes, compute the associated replication factor f_S, and check whether f_S is no larger than f.

□

Chapter 4

Approximation Algorithms

In light of the intractability of $\mathsf{DS}(\varepsilon, f, m)$, 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 4.2). 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 4.3), retaining the same bounds. We are not aware of other dynamic scaling solutions that offer such bounds.

Our solution extends consistent hashing [Karger et al., 1997, Mirrokni et al., 2018] and hash-based partitioning [Xie et al., 2014]. We remark the following (see Chapter 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 [Byers et al., 2003, Karger and Ruhl, 2004, Xin et al., 2013, Raab and Steger, 1998, Mirrokni et al., 2018], consistent hashing does no better than random hash partitioning and gives no guarantee on partition quality. (3) In particular, the algorithms of [Karger et al., 1997, Mirrokni et al., 2018] have no guarantee on replication factor f, and [Xie et al., 2014] gives no guarantee on balance factor ε.

4.1 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 [Karger et al., 1997] 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 [Xin et al., 2013], often leads to poor locality. To rectify this, we employ degree-based hashing proposed in [Xie et al., 2014], 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 $\sqrt{\frac{2M \log N}{N}}$ [Raab and Steger, 1998], where M and N are the number of balls and bins, respectively. One might want to add virtual workers to mitigate the unbalance [Karger et al., 1997], but it works only when $M = O(N \log N)$ [Raab and Steger, 1998]. 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 [Mirrokni et al., 2018]. In addition, we adopt degree-based hashing and extend consistent hashing to weighted consistent hashing, which was not studied in [Karger et al., 1997, Mirrokni et al., 2018].

(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 $\mathcal{C}$ [DeCandia et al., 2007]. 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.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 [Gonzalez et al., 2012]. The power-law constant helps us bound replication factor, but it has no impact on the bound on migration cost.

4.2 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 compute the hash value of edges to improve replication factor; and (2) we evenly distribute the fragments on the unit circle $\mathcal{C}$ to reduce migration cost. More specifically, the initial partition $\Pi(n) = (E_1, \ldots, E_n)$ 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

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

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. 1.1 (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 $\mathsf{next}(i) = (i+1) \bmod n$. 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*:

$$I_{\max} \leq 2I_{\min}, \tag{4.1}$$

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 [Mirrokni et al., 2018] (see below).

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

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

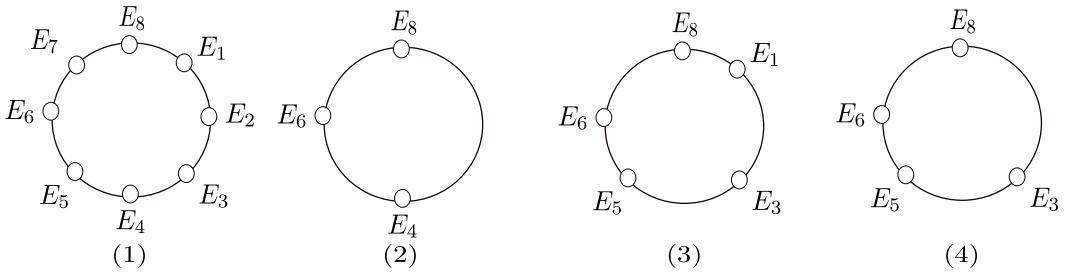


Figure 4.1: Scaling in

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 (see Appendix A.2 for proof). □

Example 3: Suppose that we initially have 8 fragments as shown in Fig. 4.1 (1). We show how to remove 5 fragments.

(1) Based on the strategy, the fragments in Fig. 4.1 (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. 4.1 (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. 4.1 (1), we end up with Fig. 4.1 (3), and can no longer remove more fragment without violating the invariant. Indeed, if we further remove E_1, we end up with Fig. 4.1 (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. 4.2.

Algorithm BVC^+ Given $\Pi(n)$, ε and $k > 0$, BVC^+ extends $\Pi(n)$ to $\Pi(n+k)$ in three

---

Algorithm BVC$^+$

Input: A partition $\Pi(n) = (E_1, \ldots, E_n)$ of G,
a number $k > 0$, and a balance factor ε.

Output: An ε-balanced new partition $\Pi(n+k) = (E_1, \ldots, E_{n+k})$.

/* Step (1): Adjust fragments on $\mathcal{C}$ */
1. identify k locations $L_{n+1}, \ldots, L_{n+k}$ for fragments to plug in;
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. $i^* = \text{next\_par}(e.\mathsf{hash}, \mathcal{C})$; /* find 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. $w \leftarrow \lceil (1+\varepsilon)\frac{|E|}{n+k} \rceil$;
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. $\mathsf{next} \leftarrow (i+1) \mod n$;
12. migrate ΔE_i to fragment E_{next};
13. $E_{\mathsf{next}} \leftarrow E_{\mathsf{next}} \cup \Delta E_i$;

Algorithm BVC$^-$

Input: A partition $\Pi(n) = (E_1, \ldots, E_n)$ of G,
a number $0 < k < n$, and a balance factor ε.

Output: A new partition $\Pi(n) = (E'_1, \ldots, E'_{n-k})$ of G.

1. identify and remove fragments $E_{j_1}, \ldots E_{j_k}$ from $\mathcal{C}$, using a stack;
2. **for** each edge $e \in \bigcup_{i=1}^{k} \{E_{j_i}\}$ **do**
3. $i = \text{next\_par}(e.\mathsf{hash}, \mathcal{C})$; /* find 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 in Algorithm BVC$^+$;

Figure 4.2: Algorithm for scaling out/in

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 [Karger et al., 1997, Mirrokni et al., 2018] and the degree-based approach [Xie et al., 2014].

(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 [Mirrokni et al., 2018] to balance the partition (lines 7-13). For each fragment E_i, if it is not balanced ($|E_i| > \lceil (1+\varepsilon)\frac{|E|}{n+k} \rceil$), then it forwards $|E_i| - \lceil (1+\varepsilon)\frac{|E|}{n+k} \rceil$ 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 (see Appendix A.3 for details). 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. 1.1 (b). □

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 $E_{j_1}, \ldots, E_{j_{|k|}}$ 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 $E_{j_1}, \ldots, E_{j_{|k|}}$ 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^+

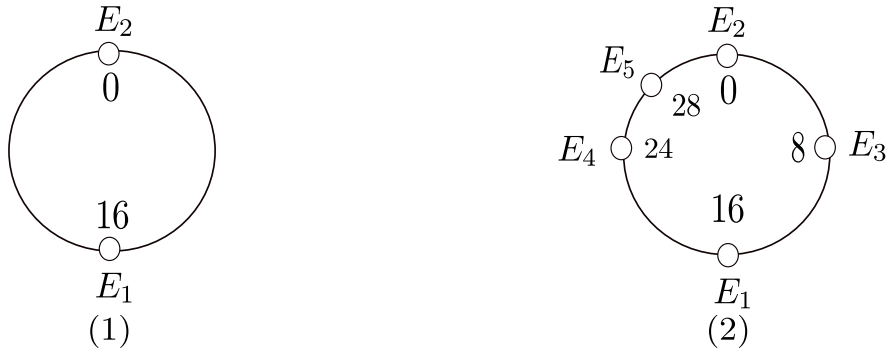


Figure 4.3: Scaling out

(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 [Xie et al., 2014].

Theorem 3: *If $k > -n$ and $\varepsilon > 1 + 2\beta_k$, 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 Chapter 6), $\beta_k \leq 0.009$ for n=64, where $|E|$ is approximately 1.5 billion.

(2) Edge selection in linear probing affects neither migration cost [Mirrokni et al., 2018] 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 [Xie et al., 2014]; 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 (see Appendix A.4 for details).

(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 $\lceil(1+\varepsilon)\frac{|E|}{n}\rceil$ 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 [Dubhashi and Panconesi, 2009]. 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 ($i \in [1,n]$) after BVC^- terminates. To bound the replication factor, by its definition, we only need to bound the expected value of $|V_i|$ for all $i \in [1,n]$. 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 [Xie et al., 2014], 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. □

4.3 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 [Valiant, 1990], 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 4.2), 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 this *in parallel*: for each existing fragment E_i $(0 < i < n)$, 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 (see Appendix A.5 and A.6).

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.

Chapter 5

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 5.1). As proof of concept, we scale two existing vertex-cut partitioners (Section 5.2).

5.1 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 $\varepsilon = 0$ 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 $\varepsilon = 0$. 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 (see Appendix A.7 and A.8 for proofs of 4 and 5).

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.* □

One can show that VP^+ and VP^- generate partitions as balanced as $\Pi(n)$, no matter whether $\Pi(n)$ is balanced.

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 $F + k \cdot \frac{k}{n+k} \frac{2|E|}{\min\{\tau_i\}_{i=1}^n \cdot |V|}$ with the edge selection strategy above. Here $\min\{\tau_i\}_{i=1}^n$ 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.

5.2 Scaling Stream Partitioners

As case studies, we next scale HDRF [Petroni et al., 2015] and Greedy (Powergraph [Gonzalez et al., 2012]), two well-known vertex-cut partitioners. Consider a current vertex-cut partition $\Pi(n) = (E_1, \ldots, E_n)$.

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:

$$S(u, v, E_i) = S_{\text{REP}}(u, v, E_i) + S_{\text{BAL}}(E_i), \tag{5.1}$$

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

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

Here $\deg(u)$ and $\deg(v)$ are the degrees of u and v, respectively. The balance score $S_{\text{BAL}}(E_i)$ is defined as

$$S_{\text{BAL}}(E_i) = \lambda \frac{\text{MAXSIZE} - |E_i|}{1 + \text{MAXSIZE} - \text{MINSIZE}},$$

where λ is a user-defined parameter that controls the impact of the balance score, and MAXSIZE and MINSIZE are the maximum and minimum sizes of all fragments when processing edge e. 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 $\frac{k}{n(n+k)}|E_i|$ 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 Chapter 6, 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 [Petroni et al., 2015].

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 (see Appendix A.9). 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 [Gonzalez et al., 2012]. It can be seen as a special case of HDRF. It also uses Eq. (5.1) 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 [Sajjad et al., 2016], 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.

Chapter 6

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.

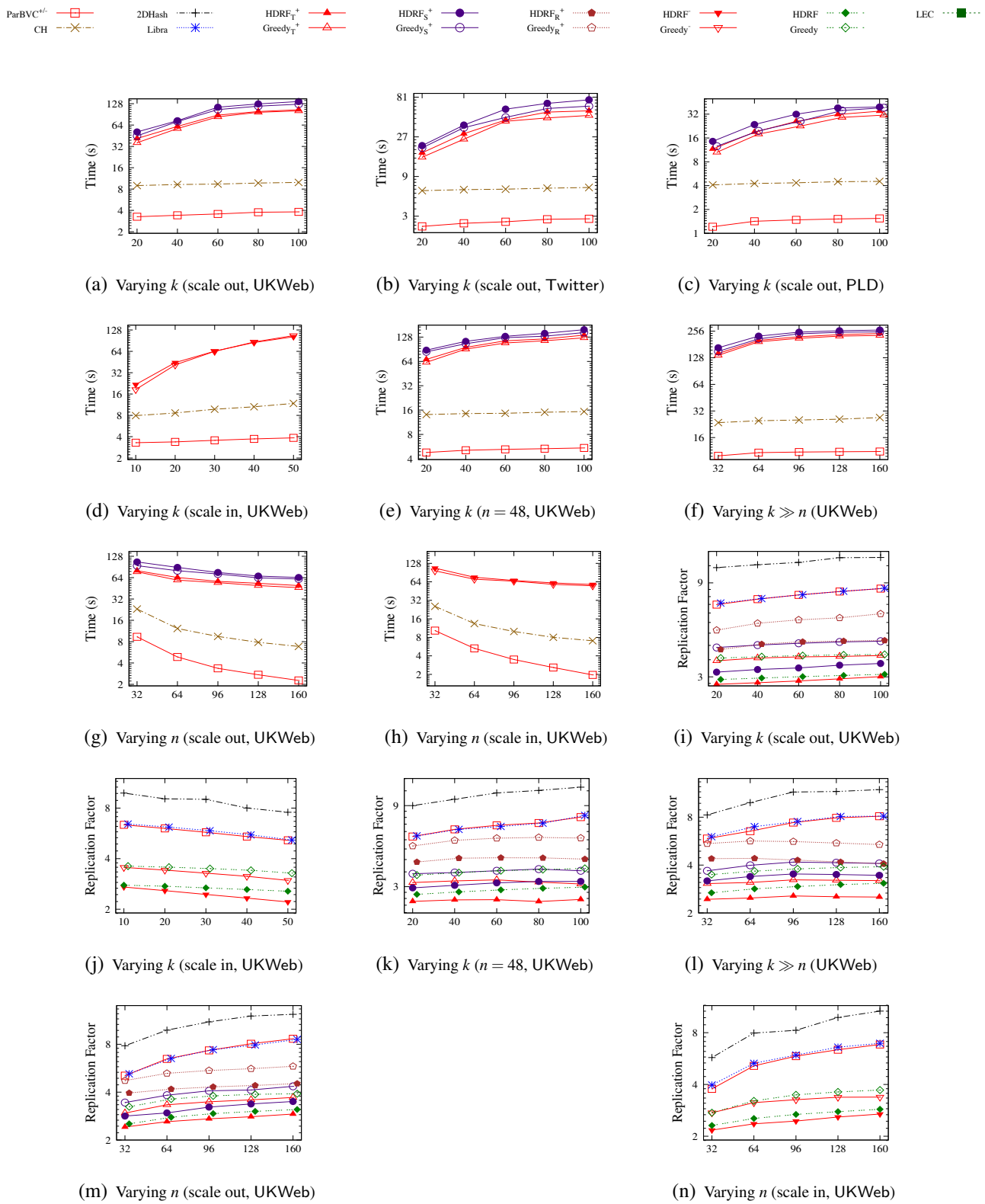
Experimental setting. We start with the setting.

Datasets. We used three real-life power-law graphs: (a) PLD [Meusel et al., 2014], 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 [Kwak et al., 2010], a social network with 42 million users and 1.5 billion links; and (c) UKWeb [ukw, 2006], 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$^+$ (Chapter 4), and parallel HDRF$^+$, HDRF$^-$, Greedy$^+$ and Greedy$^-$ (Chapter 5), all in C++, compared with the following: (1) CH [Karger et al., 1997], 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 [Xin et al., 2013], a widely used hash-based vertex partitioner; (3) Libra [Xie et al., 2014], a state-of-the-art degree-based hashing algorithm; and (4) vertex partitioners HDRF and Greedy (Chapter 5). 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



(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)

Figure 6.1: Performance Evaluation

by $\mathsf{HDRF}^+_{\mathrm{S}}$ and $\mathsf{HDRF}^+_{\mathrm{T}}$ the implementations of HDRF^+ with edge selection based on score and timestamp, respectively; similarly for $\mathsf{Greedy}^+_{\mathrm{S}}$ and $\mathsf{Greedy}^+_{\mathrm{T}}$. 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 $\mathsf{HDRF}^+_{\mathrm{R}}$ and $\mathsf{Greedy}^+_{\mathrm{R}}$, respectively. We parallelized the algorithms as described in Section 5.2. The mini-batch size is set to 256 by default.

The experiments were conducted on GRAPE, a parallel graph processing engine [Fan et al., 2017], 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. In the experiments, each fragment was handled by one process that ran on an exclusive core. Each experiment was repeated 5 times; 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 $\varepsilon = 0.1$; 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 6.1(a)-6.1(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. 6.1(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 4.2).

(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 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. 6.1(e) on UKWeb, its scaling-out performance pattern is consistent with Fig. 6.1(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. 6.1(f), the results are consistent with Figures 6.1(a)–6.1(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. 6.1(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. 6.1(h), the results for scaling in are consistent with Fig. 6.1(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, (b) balance factor ε. We also evaluated (c) the effectiveness of the edge selection strategies (Section 5.2) for stream partitioners (see Appendix A.10). We used UKWeb; the results on Twitter and PLD are consistent (not shown).

Replication factor. In the same setting as Exp-1, Figures 6.1(i)-6.1(n) report replication factors of the algorithms.

(1) Varying k. As shown in Fig. 6.1(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. 6.1(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$^+_{\mathrm{R}}$) (see Appendix A.10).

(f) As in Exp-1, we also tested the case when $n = 48$. As shown in Fig. 6.1(k), the results are consistent with Fig. 6.1(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. 6.1(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 6.1(m) and 6.1(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.

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

Table 6.1: Balance factor

Balance factor. We next evaluated the balance factor. Table 6.1 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 5.2) 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.

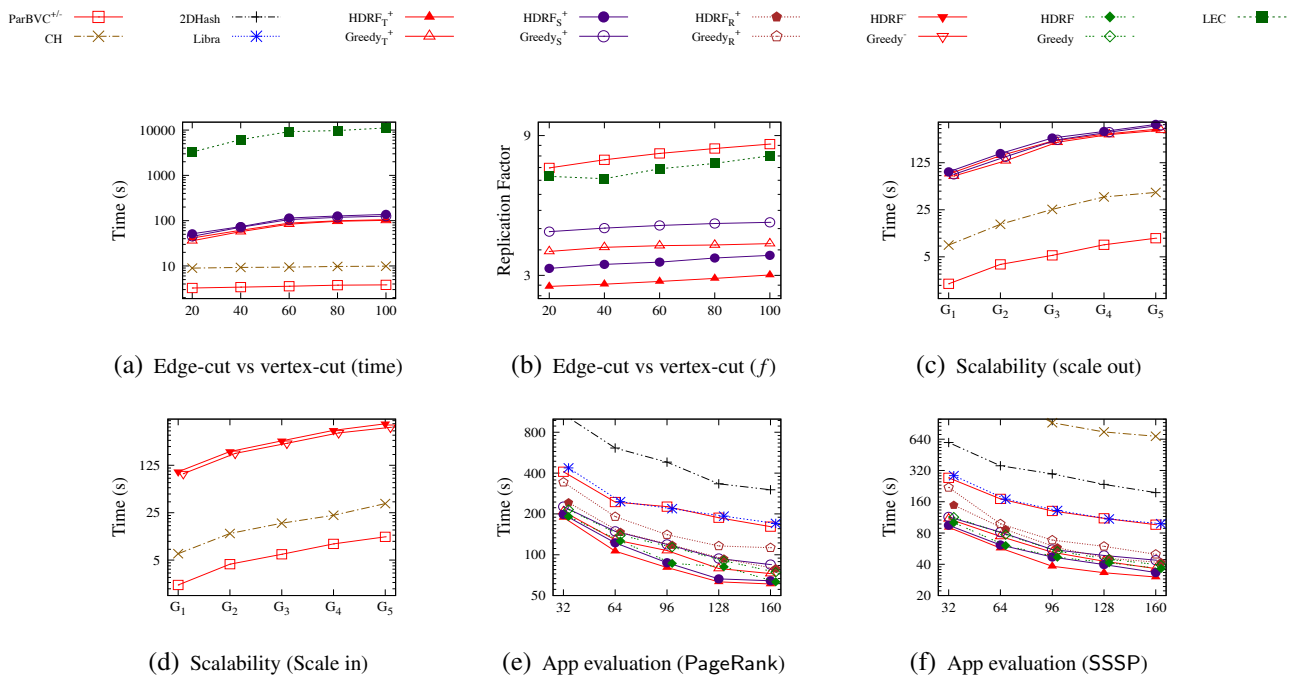
(2) ParBVC$^+$ enforces a user-defined balance factor $\varepsilon = 0.1$ by its rebalancing stage (Section 4.2). In contrast, CH and 2DHash have ε as large as 3.46, respectively, and 1.16. Libra has a smaller ε, but it is not efficient as ParBVC$^+$ (Exp-1).

(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 $\varepsilon = 0.1$ 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, since more data needs to be shipped for smaller ε. The results of ParBVC$^-$ are consistent.

Edge-cut partitions. We also compared with LEC [Pujol et al., 2010], a scaling algo-



(a) Edge-cut vs vertex-cut (time) (b) Edge-cut vs vertex-cut (f) (c) Scalability (scale out)

(d) Scalability (Scale in) (e) App evaluation (PageRank) (f) App evaluation (SSSP)

Figure 6.2: Performance Evaluation

rithm for edge-cut partitions. Following [Zhang et al., 2017], 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 6.2(a) and 6.2(b). (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. 6.2(c)-6.2(d), (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). We do not report the time that is longer than 1000 seconds.

(1) As shown in Figures 6.2(e)-6.2(f), (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.

Chapter 7

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.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Walshaw et al., 1997] Walshaw, C., Cross, M., and Everett, M. G. (1997). Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108.

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

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

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

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

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

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

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

Appendix A

Appendix: Proofs and Details

A.1 Proof of Theorem 1

To prove Theorem 1, we first study MIN-VC, a vertex-cut version of k-cut problem [Goldschmidt and Hochbaum, 1994]. Given a graph G, a number k and a replication factor $f \geq 1$, it asks whether there exists a vertex-cut k-partition $\Pi(k)$ of G with $\partial(\Pi(k)) \leq f$. Assume *w.l.o.g.* that no fragment of $\Pi(k)$ is empty.

Lemma 1: MIN-VC is NP-complete. □

Proof: Clearly MIN-VC is in NP. We show that MIN-VC is NP-hard by reduction from the maximum clique problem. Given a graph $G = (V,E)$ and a number k, the latter problem asks whether G has a k-clique, *i.e.,* a clique of size k. To simplify the discussion we assume *w.l.o.g.* that $|E| \geq \binom{k}{2}$.

Claim. G has a k-clique iff G has a vertex-cut k^*-partition $\Pi(k^*)$ with $\partial(\Pi(k^*)) \leq \frac{k+2(k^*-1)}{|V|}$, where $k^* = |E| + 1 - \binom{k}{2}$.

($\Rightarrow$) Assume that G has a k-clique. Then there exists a k^*-partition that puts the edges of the k-clique in a fragment, and the rest k^*-1 edges in the other fragments one by one.

($\Leftarrow$) Assume that there exists a partition $\Pi(k^*) = (E_1, \ldots, E_{k^*})$ of G such that $\partial(\Pi(k^*)) \leq \frac{k+2(k^*-1)}{|V|}$. We show that G has a k-clique. Denote by V_1, ..., V_{k^*} the vertex sets of E_1, ..., E_{k^*}, respectively. We pick one edge $\bar{e}_i$ for each E_i, $1 \leq i \leq k^*$, and let $\bar{e}_i = (u_i, v_i)$, $\overline{V_i} = V_i \setminus \{u_i, v_i\}$, and $\overline{E_i} = E_i \setminus \{\bar{e}_i\}$. By assumption we have that $\sum_{i=1}^{k^*} |\overline{V_i}| \leq k-2$. Let $\overline{E} = \bigcup_i^{k^*} \overline{E_i}$. Then $|\overline{E}| = \binom{k}{2} - 1$.

In the following we show that there exists some $t \in [1,k^*]$ such that $\overline{E} \cup \{\bar{e}_t\}$ forms a k-clique in G. To see this, one can construct an auxiliary graph $G^* = (V^*, E^*)$ from G by treating the nodes in $\overline{V_i}$'s as distinct ones and merging all u_i's and v_i's into two mega vertices u^* and v^*, respectively. Observe that $|V^*| = \sum_{i=1}^{k^*} |\overline{V}_i| + 2 \leq k$ and $|E^*| = |\bar{E}| + 1 = \binom{k}{2}$. Hence $|V^*| = k$ and G^* is a k-clique. It follows that (a) $\overline{V}_i \cap \overline{V}_j = \emptyset$ for $i \neq j$; and (b) all edges in $\overline{E}$ are in the same fragment, say E_t.Then E_t, *i.e.,* $\overline{E} \cup \{\bar{e}_t\}$, forms a k-clique. □

Proof of Theorem 1. Given a n-partition $\Pi(n)$ of G and a new $(n+k)$-partition $\Pi(n+k)$, one can compute the balance factor ε, replication factor f of $\Pi(n+k)$ and migration cost m from $\Pi(n)$ to $\Pi(n+k)$ in polynomial time. This implies that DS, $\mathsf{DS}(f,m)$ and $\mathsf{DS}(\varepsilon,f)$ are all in NP.

(1) For the lower bounds, it suffices to show that $\mathsf{DS}(f,m)$ and $\mathsf{DS}(\varepsilon,f)$ are NP-hard, since $\mathsf{DS}(f,m)$ and $\mathsf{DS}(\varepsilon,f)$ are special cases of DS. We assume *w.l.o.g.* that $k=1$. The reductions can be revised accordingly for any fixed k.

(i) We show that $\mathsf{DS}(f,m)$ is NP-hard by reduction from MIN-VC. Given a graph G'=(V',E'), a number k', and a replication factor f', we construct an instance of $\mathsf{DS}(f,m)$ as follows: G=G', n=$k'-1$, k=1, m=$|E'|$, f=f' and the initial n-partition $\Pi(n)$ is constructed by assigning one edge to each of the first $n-1$ partitions and the rest of the edges to the n-th partition. Clearly G' has a k'-partition with replication factor smaller than f' if and only if there exists a new partition $\Pi(n+k)$ of G such that $\partial(\Pi(n+k)) \leq f$ and the migration cost from $\Pi(n)$ to $\Pi(n+k)$ is bounded by m.

(ii) We show that $\mathsf{DS}(\varepsilon,f)$ is NP-hard by reduction from the 3-Partition Problem, which is NP-complete [Garey and Johnson, 1979]. Given $3t$ integers $a_1, \ldots, a_{3t}$ and a threshold S such that $S/4 < a_i < S/2$ and $\sum_{i=1}^{m} a_i = tS$, it asks whether the numbers can be partitioned into t triples such that each triple adds up to S. It remains NP-complete when $a_1, \ldots, a_{3t}$ are unary.

Given an instance of the 3-Partition Problem, we construct an instance of $\mathsf{DS}(\varepsilon,f)$, which consists of a graph G, an initial partition $\Pi(n)$, a number k, a balanced factor ε and a replication factor f defined as follows:

- G is composed of m disjoint stars; more specifically, for each a_i, we add a vertex v_i and a_i additional vertices $u_{i,1} \ldots, u_{i,a_i}$, and connect v_i to each $u_{i,j}$;

- $n = t-1$ and $\Pi(n)$ is constructed similarly to (i);
- $k = 1$, *i.e.,* we increase the partition number by 1; and
- $\varepsilon = 0$ and $f = 1$, *i.e.,* we require that $\Pi(n+k)$ is perfectly balanced and no vertex is cut.

One can see that there exists a desired new partition $\Pi(n+k)$ for $\mathsf{DS}(\varepsilon, f)$ iff there exists a 3-partition of $a_1, \ldots, a_{3t}$.

(2) A PTIME algorithm for $\mathsf{DS}(\varepsilon, m)$ is as follows. Each time it moves one edge from the maximum partition to the minimal one until either (i) the current balance factor is no larger than ε; or (ii) the migration cost exceeds bound m. If (i) happens then it returns "yes"; otherwise it returns "no".

We now give a PTIME algorithm for $\mathsf{DS}(f, m)$, when both k and n are fixed and m is ∞ (unrestricted). Note that in a partition $\Pi(n+k)$ with the minimum replication factor, the number of cut vertices, *i.e.,* vertices that occur in more than one fragment, is bounded by $2(n+k)$. Given graph $G = (V, E)$, the algorithm computes the minimum replication factor by enumerating all possible partitions. Suppose that G has m_1 connected components. It works as follows.

(a) If $m_1 \geq n+k$, then return "yes"; otherwise continue.

(b) Check whether removing one vertex from G can result in at least $n+k$ connected components; if not, continue; otherwise check whether $1 + \frac{n+k-m_1}{|V|} \leq f$; if so, return "yes"; otherwise, return "no".

(c) Enumerate all subsets V_1 of V such that $|V_1| \leq 2(n+k)$. For each such subset, cut these nodes from G, and check whether from the cut graph we can deduce a partition such that the replication factor is no larger than f. If so, return "yes"; otherwise, if the replication factor is larger than the threshold f for any subset V, then return "no".

Since $n+k$ is a constant, the algorithm is in PTIME. Indeed, the number of possible partitions in (c) is bounded by a constant $2^{4(n+k)^2}$. The correctness is ensured by the following: (i) the minimum replication factor of (a) and (b) is 1 and $1 + \frac{n+k-m_1}{|V|}$, respectively; and (ii) the minimum replication factor for $(n+k)$-partitions is bounded by $1 + \frac{2(n+k)}{|V|}$. □

A.2 Proof of Lemma 2

Let $L_1, \ldots, L_{n_1}$ be the locations on the stack from bottom to top. We refer to the following property as *stack invariant*: for any $j \in [1, n_1]$, the interval invariant holds when placing j fragments at locations $L_1, \ldots, L_j$, respectively. The stack invariant implies the interval invariant since the latter only concerns the placement of n_1 fragments at $L_1, \ldots, L_{n_1}$.

To prove Lemma 2, we show that (a) the initial stack satisfies the stack invariant; and (b) adding or removing fragments during scaling does not violate the stack invariant.

(a) Let $L_1, \ldots, L_n$ be locations from bottom to top on the initial stack, which correspond to fragment removal sequence in the stack construction. Let $I_1, \ldots, I_j$ be intervals in the clockwise order induced by $L_1, \ldots, L_j$, and I_1 and I_2 be intervals proceeding and succeeding L_j, respectively. Removing L_j will merge I_1 and I_2. We show the following:

$$I_1 \leq I_2 \leq \cdots \leq I_j, \text{ and } I_j \leq 2I_1. \qquad (\star)$$

The stack invariant follows from $(\star)$ directly.

We prove property $(\star)$ by induction. Initially $j = n$ and $I_1 = \cdots = I_n$, and thus $(\star)$ holds. For the inductive step, suppose that $(\star)$ holds for j. We show that $(\star)$ holds for $j-1$. Since we remove fragments in an alternative manner, after removing L_j, L_{j-1} is on the top of the stack, and is located between I_3 and I_4. To show $(\star)$, we prove that $I_j \leq I_1 + I_2$ and $I_1 + I_2 \leq 2I_3$, because after removing L_j, the intervals I_1 and I_2 are merged to form the largest interval by $(\star)$. For $I_j \leq I_1 + I_2$ and $I_1 + I_2 \leq 2I_3$, note that $I_j \leq 2I_1 \leq I_1 + I_2$ and $I_1 + I_2 \leq 2I_2 \leq 2I_3$ by the inductive hypothesis.

(b) Suppose that before removing or adding a fragment, the stack invariant holds. Clearly removing a fragment does not violate the stack invariant since we simply pop a location from the stack. We show that adding a fragment does not violate the invariant either. Let $I'_{\max}$ and $I'_{\min}$ (resp. $I_{\max}$ and $I_{\min}$) be the lengths of the maximum and minimum intervals before (resp. after) adding a fragment, respectively. Since new fragments are added at the middle of the largest interval, and $I'_{\max} \leq 2I'_{\min}$ by the hypothesis, we have that

$$I_{\max} \leq I'_{\max} \leq 2 \times \frac{1}{2} I'_{\max} \leq 2 \times \min\{\frac{1}{2} I'_{\max}, I'_{\min}\} = 2I_{\min}. \square$$

A.3 More details of Example 4

We show how BVC^+ extends the partition $\Pi(2)$ of Example 2 to a new partition $\Pi(5) = (E_1, \ldots, E_5)$.

(1) Algorithm BVC^+ first identifies 3 locations on the circle $\mathcal{C}$ to place the new fragments E_3, E_4 and E_5. At the beginning, there are only two intervals with the same length induced by E_1 and E_2, located at 16 and 0, respectively, as shown in Fig. 4.3(1) (note that we have 2^5 locations, labeled 0–31). BVC^+ selects one of them and places E_3 at location 8. The largest interval then becomes the one between E_1 and E_2. BVC^+ then places E_4 at location 24, in the middle of E_1 and E_2. Now all intervals have length 8. BVC^+ simply bisects one of them to place E_5, *e.g.,* at location 28.

(2) BVC^+ then finds edges that belong to the new fragments, and moves them to the right place. By consistent hashing, $e_{4,1}$, $e_{4,3}$, and $e_{5,2}$ are closer to fragment E_4, and thus are moved from E_2 to E_4. It also moves $e_{3,1}$, $e_{3,2}$ and $e_{3,5}$ from E_1 to E_3, and $e_{5,3}$, $e_{6,1}$ and $e_{6,5}$ from E_2 to E_5. 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 partition $\Pi(5)$ shown in Fig. 1.1 (b).

A.4 Proof of Theorem 3

We first remark the following about Theorem 3.

(1) The lower bound β_k for balance factor (a) is not very restrictive, (b) requires ε to be larger than 1, and (c) incorporates k. These aim to bound the migration cost.

(a) Taking Twitter as an example (see Chapter 6), $\beta_k \leq 0.009$ for $n = 64$, where $|E|$ is approximately 1.5 billion. Indeed, in the real world it is common to find that $|E| \gg n$.

(b) When $\varepsilon > 1$, the cost in linear probing is quite small. Note that the expected number of edges hashed to a fragment is proportional to the length of the interval preceding the fragment, since its vertices are hashed independently. If the fragments on $\mathcal{C}$ are not evenly distributed, it may incur heavy migration cost in the linear probing process to restore balance. By the interval invariant, *i.e.,* the largest interval is at most twice as large as the smallest one, the maximum expected number of edges in a fragment is at

most twice the average, and thus we can bound the migration cost.

(c) When adding k fragments and when k is large, the "capacity" of the fragments (the maximum size allowed by an ε-balanced partition) decreases, and the chance that a partition gets overfull increases. This increases the expected value of the migration cost. To cope with this, we incorporate k into the lower bound β_k for balance factor ε to ensure that only a small number of edges need to be migrated.

(2) The interval invariant allows us to bound not only migration cost (see (1) above), but also replication factor f. (a) Given the invariant, we can bound the probability of edges hashed to fragments, and deduce a bound on f. (b) As remarked in (1), when $\varepsilon > 1 + 2\beta_k$, the migration cost during linear probing is quite small; that is, we can avoid further degradation of partition quality when restoring balance.

Our bound on the replication factor f differs from the one of [Xie et al., 2014] by a small factor $\frac{2\eta}{|V|}$. The factor comes from the effort to balance fragments, which is not ensured by [Xie et al., 2014].

(3) Edge selection in linear probing affects neither migration cost [Mirrokni et al., 2018] nor the upper bound for replication factor.

(4) The bound for migration cost holds on general graphs, but not the expected replication factor f_e. On a power-law graph G, f_e of degree-bashed hashing would decrease when G gets more skewed [Xie et al., 2014]; this does hold on general graphs.

Proof. We next show the bounds on migration cost and the replication factor when removing or adding $|k|$ fragments.

Migration cost. We first bound the total migration cost.

(A) Removing fragments. We start with the migration cost of BVC^-. More specifically, we show that to remove $|k|$ fragments, we need to move at most $O(|k|\frac{|E|}{n})$ edges.

The migration cost for removing fragments consists of (1) the cost for moving edges from removed fragments to other fragments; and (2) the cost for rebalancing the fragments.

For (1), since each fragment has at most $\lceil(1+\varepsilon)\frac{|E|}{n}\rceil$ edges, and $|k|$ fragments are to be removed, $O(|k|\frac{|E|}{n})$ edges are moved from removed fragments to other fragments. Thus the migration cost for (1) is bounded by $O(|k|\frac{|E|}{n})$.

For (2), it suffices to show that the expected number of edges in E_i to be forwarded is bounded by $O(\frac{1}{(n+k)^2})$. For if it holds, then the expected migration cost is bounded by $O(\frac{1}{n+k})$ since each edge can be moved at most $n+k$ steps. Since there exist $n+k$ fragments, the total migration cost is bounded by $O(1)$, *i.e.,* bounded by a small number.

To show the bound $O(\frac{1}{(n+k)^2})$, let X_j be the number of edges hashed to E_i when hashing edges $e_j \in E$. Then the number of edges hashed to E_i is $X_1 + \ldots + X_{|E|}$, denoted by X. Let $e_{k_l^1}, \ldots, e_{k_l^{t_l}}$ be all the edges that are hashed by v_l to the same fragment, since we adopt degree-based hashing. Let $Y_l = X_{k_l^1} + \ldots + X_{k_l^{t_l}}$. Since each edge can be hashed by only one vertex, X can be rephrased as $Y_1 + \ldots + Y_{|V|}$. Moreover, since vertices are hashed independently, $Y_1, \ldots,$ and $Y_{|V|}$ are independent. Denote by h_{v_l} the number of edges hashed by v_l and let $h_{\max} = \max\{h_v\}_{v=1}^{|V|}$.

Now we show the bound on the expected number of moved edges. Denote $\lceil (1+\varepsilon)\frac{|E|}{n+k} \rceil$ by B. Since E_i contains at most B edges, we know that $E[X]$ can be bounded by

$$\Sigma_{l=0}^{|E|-B} l \times \Pr[X = B+l] \leq \Sigma_{l=0}^{|E|} \Pr[X > B+l] \leq \Sigma_{l=0}^{|E|} \Pr[X > B].$$

We next bound $\sum_{l=0}^{|E|} \Pr[X > B]$. To this end, we use the Bernstein's inequality [Dubhashi and Panconesi, 2009], which states: if $Y_1, \ldots, Y_{|V|}$ are independent from each other, and if $Y_j - E[Y_j] \leq b$ for a constant b with $j \in [1, |V|]$, then for any $t > 0$,

$$\Pr[X > E[X] + t] \leq \exp(-t^2/(2\sigma^2 + \frac{2bt}{3})),$$

where $\sigma^2 = \sum_1^{|V|} \sigma_j^2$ is the variance of X.

To use this inequality, we show the following: (a) $E[X]$ satisfies that $I_{\min}\frac{|E|}{2^c-1} \leq E[X] \leq I_{\max}\frac{|E|}{2^c-1}$, by the interval invariant; here $I_{\min}$ (resp. $I_{\max}$) is the size of the minimum (resp. maximum) interval, and $2^c - 1$ is the size of the circle; and (b) the variance $\sigma^2 \leq \sum_{j=1}^{|V|} \frac{I_{\max}}{2^c-1} h_j^2 \leq \frac{I_{\max}|E|h_{\max}}{2^c-1}$ and $Y_j - E[Y_j] \leq h_{\max}$. By $\varepsilon > 1 + 2\beta_k$, $\beta_k = \sqrt{\beta_k^1}(\sqrt{\beta_k^1} + \sqrt{2})$ and $\beta_k^1 = \frac{8(n+k)h_{\max}\log((n+k)\sqrt{|E|+1})}{|E|}$, $(\frac{\varepsilon-1}{2})^2 > \beta_k^1(2 + \frac{\varepsilon-1}{2})$. By Bernstein's inequality [Dubhashi and Panconesi, 2009], we can deduce the following:

$$\begin{aligned}
\Pr[X > B] &\leq \Pr[X > (1 + (\varepsilon-1)/2)E[X]] \\
&\leq \exp(-\frac{(\frac{\varepsilon-1}{2}E[X])^2}{2\sigma^2 + \frac{2h_{\max}(\frac{\varepsilon-1}{2}E[X])}{3}}) \\
&\leq 1/((n+k)^2 \times (|E|+1)).
\end{aligned}$$

Thus $\sum_{l=0}^{|E|} \Pr[X > B] \leq (|E|+1)\frac{1}{(n+k)^2(|E|+1)} \leq O(\frac{1}{(n+k)^2})$. Hence, $\sum_{l=0}^{|E|-B} l \times \Pr[X = B+l] \leq O(\frac{1}{(n+k)^2})$.

Putting these together, we know that the expected moving cost for removing $|k|$ fragments is bounded by $O(|k|\frac{|E|}{n})$.

(B) Adding fragments. Next, we show that to add k fragments, we need to move at most $O(k\frac{|E|}{n+k})$ edges.

Similar to BVC^-, the migration cost of algorithm BVC^+ consists of (1) the cost to move edges from old fragments to new fragments; and (2) the cost to balance fragments. The proof of (2) is similar to BVC^-; we omit its details here.

For (1), the analysis is almost the same as that of BVC^-, except that when $k \gg n$, it is possible that many new fragments are added between two old fragments. In this case, we send the edges directly to the fragments, rather than forward them by linear probing. Since the interval variant holds, the expected number of edges in each new fragment is in $O(\frac{|E|}{n+k})$. Hence the expected number of edges moved from old fragments to the new ones is in $O(k\frac{|E|}{n+k}) = O(k\frac{|E|}{n+k})$.

Putting these together, we know that the expected moving cost of algorithms BVC^+ and BVC^- is bounded by $O(k\frac{|E|}{n})$.

Replicated factor. Now we show the bound on the replication factor f after scaling as stated in Theorem 3.

We first review a result established in [Xie et al., 2014]. For a graph, suppose that the minimum degree of its vertices is $d_{\min}$, and the degrees of the graph follow a power-law distribution $Pr(d_v = d) = (\alpha-1)d_{\min}^{\alpha-1}d^{-\alpha}$ for any vertex v in the graph, where α is a positive constant and $d \geq d_{\min}$, and each vertex is hashed by itself once. Then the expected number of vertices hashed to a fragment E_i is bounded by $|V|(1-(1-p_i)^{\theta})$, where $\theta = d_{\min} \times \frac{(\alpha-1)^2}{(\alpha-2)(2\alpha-3)} + \frac{1}{2}$, and p_i is the probability that a vertex is hashed to E_i.

Using this result, we show the bound on the expected replication factor of Algorithms BVC^- and BVC^+. To this end, we only need to bound the expected number of vertices in a fragment E_i, denoted by x^i, after BVC^- and BVC^+ are executed. Let x_h^i and x_f^i be the number of vertices hashed to E_i and the number of vertices forwarded to E_i, respectively. Then $x^i \leq x_h^i + x_f^i$ and $E[x^i] \leq E[x_h^i] + E[x_f^i]$.

We now bound $E[x_h^i]$ and $E[x_f^i]$. For $E[x_h^i]$, since we evenly place the fragments, the probability that a vertex is hashed to a fragment E_i^i is bounded by $2\frac{1}{n}$, which is

---

Algorithm ParBVC$^+$

Input: $\Pi(n)$ and ε as in ParBVC$^-$, and a number $k > 0$.

Output: A new partition $E_1, \ldots, E_{n+k}$ of G.

1. identify k locations $L_1, \ldots, L_k$ for fragments to plug in;
2. add k new fragments $E_{n+1}, \ldots, E_{n+k}$ at location $L_1, \ldots L_k$;
3. for $i \in [1,n]$, each Pi works on E_i *in parallel* **do**
4. **for each** $e \in E_i$ **do** /* superstep */
5. $i^* = \text{next\_par}(e.\text{hash}, \mathcal{C})$; /* find the next fragment on $\mathcal{C}$ */
6. **if** $i^* \in \{n+1, \ldots, n+k\}$ **then**
7. move e to fragment E_{i^*};
8. for $i \in [1,n]$, each P_i works on E_i *in parallel* **do** /* superstep */
9. balance E_i by linear probing;

Figure A.1: Algorithms ParBVC$^+$

ensured by the interval invariant. By the results of [Xie et al., 2014] above, $E[x_h^i]$ is at most $|V|(1-(1-2\frac{1}{n})^\theta)$. For $E[x_f^i]$, after one step of movement, only one fragment increases its number of vertices by 2 at most. By the bound on the migration cost given above, there exists a constant η such that we need at most η steps of balancing. Thus $\sum_{i=1}^{n} E[x_f^i]$ is bounded by 2η.

Taken together, the expected replication factor is at most $n \times \frac{|V|(1-(1-2\frac{1}{n})^\theta)}{|V|} + \frac{2\times\eta}{|V|} = n(1-(1-2\frac{1}{n})^\theta) + \frac{2\eta}{|V|}$. □

A.5 Pseudo Code of ParBVC$^+$

Figure A.1 shows the details of algorithm ParBVC$^+$ (Section 4.3). Given a hash-based partition $\Pi(n)$ of a graph G, a balance factor ε and a number $k > 0$, ParBVC$^+$ scales out $\Pi(n)$ to a new ε-balanced partition $\Pi(n+k)$ as follows. It first identifies k locations and plugs in k fragments just like BVC$^+$ (lines 1-2). It then identifies edges that belong to the fragments, and moves them to the right place, *in parallel* (lines 3-7). Finally, it balances the fragments via linear probing *in parallel* as in BVC$^+$ (lines 8-9).

Algorithm ParBVC$^-$

Input: $\Pi(n) = (E_1, \ldots, E_n)$ of G, $-n < k < 0$ and ε as in BVC$^-$.

Output: A new partition $(E'_1, \ldots, E'_{n+k})$ of G.

1. identify and remove k fragments $E_{j_1}, \ldots E_{j_k}$ from unit circle $\mathcal{C}$;
2. for $i \in [1,k]$, each Pj_i works on E_{j_i} *in parallel* **do**
3. **for each** e in E_{j_i} **do** /* superstep */
4. migrate e as in ParBVC$^+$;
5. $\{E'_1, \ldots, E'_{n+k}\} \leftarrow \{E_1, \ldots, E_n\} \setminus \{E_{j_1}, \ldots, E_{j_k}\}$;
6. balance $E'_1, \ldots, E'_{n+k}$ in parallel as in Algorithm ParBVC$^+$;

Figure A.2: Algorithms ParBVC$^-$

A.6 Pseudo Code of ParBVC$^-$

Algorithm ParBVC$^-$ for scaling in is shown in Fig. A.2. Given a partition $\Pi(n)$ of graph G placed on a unit circle $\mathcal{C}$, a balance factor ε and a number $-n < k < 0$, ParBVC$^-$ scales in $\Pi(n)$ to an ε-balanced partition $\Pi(n+k)$. Like BVC$^-$, it first identifies k fragments and removes them from the circle $\mathcal{C}$, also using a stack (line 1). It then migrates the edges from the removed fragments. As opposed to BVC$^-$, ParBVC$^-$ conducts this step *in parallel*: for each removed fragment E_{j_i}, its worker P_{j_i} migrates edges in E_{j_i} (line 2-4). Finally BVC$^-$ balances the resulting partition, *in parallel* via linear probing as in ParBVC$^+$ (lines 5-6).

A.7 Proof of Proposition 4

For VP$^+$, since at most $\frac{k|E_i|}{n+k}$ edges are moved to new fragments from each fragment E_i, one can conclude that the migration cost of VP$^+$ is bounded by $\frac{k|E|}{n+k}$.

For VP$^-$, since we remove all edges from the selected $|k|$ fragment, and the number of edges in each fragment is in $O(\frac{|E|}{n})$, the migration cost of VP$^-$ is at most $O(\frac{|k||E|}{n})$. □

A.8 Proof of Proposition 5

By the definition, the replication factor is the sum of the replication factors of the original fragments and that of the new fragments. Since some edges are removed from the original fragments, the replication factor of these fragments is still bounded by F after dynamic scaling. It remains to bound the replication factor of the new fragments.

To this end, since this part of replication factor is the result of dividing the number of vertices in the new fragments by the total number $|V|$ of vertices, we only need to bound the number of vertices in the new fragments. (a) Since these vertices are selected from the original fragments, we first analyze how many vertices are selected. We bound this number by $\frac{k}{n+k}\frac{2|E|}{\tau_i}$. Denote by E_i' the selected edges from E_i; then $v(E_i')$ is the set of selected vertices in E_i'. By the edge selection strategy, we select $\frac{k}{n+k}|E_i|$ edges from E_i such that the number of vertices in the selected edges is minimum. Hence $|v(E_i')|$ must be no larger than the average number of vertices in any $\frac{k}{n+k}|E_i|$ edges in E_i, which is $\frac{k}{n+k}\frac{2|E_i|}{\tau_i}$. Therefore, the number of vertices in E_i' is no larger than $\frac{k}{n+k}\frac{2|E_i|}{\tau_i}$. Then the total number of vertices in the selected edges over all original fragments is bounded by $\frac{k}{n+k}\frac{2|E|}{\min\{\tau_i\}_{i=1}^n}$. (b) Moreover, because each selected vertex can be assigned to at most k new fragments, the total number of vertices in k new fragments is bounded by $k\frac{k}{n+k}\frac{2|E|}{\min\{\tau_i\}_{i=1}^n}$. Putting these together, we have that the replication factor after VP^+ is bounded by

$$F + k \cdot \frac{k}{n+k}\frac{2|E|}{\min\{\tau_i\}_{i=1}^n \cdot |V|}. \qquad \square$$

A.9 Proof of Proposition 6

By its definition, 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 selected edges $E_1', \ldots, E_n'$. It is easy to verify that the replication factor of $\Pi(n)'$ is at worst f''. It remains to bound the replication factor of $\Pi(k)$.

Below we show the result for the score-based strategy first, followed by the timestamp-based strategy.

Score-based strategy. To bound the replication factor of the partition $\Pi(k)$ of k new fragments, we bound the number of vertices in $\Pi(k)$. Similar to the proof of Proposition 5, observe that (a) the number of vertices in E_i' is no larger than $\frac{2k}{n+k}|E_i|$, and

then the total number of vertices in the selected edges over n fragments is bounded by $\frac{2k}{n+k}|E|$; and moreover, (b) because each selected vertex can be assigned to at most k new fragments, the total number of vertices in $\Pi(k)$ is bounded by $k\frac{2k}{n+k}|E|$. Putting these together, we have that the replication factor after VP^+ is bounded by

$$f'' + \frac{2k^2}{n+k}|E|.$$

Timestamp-based strategy. Similar to the proof above, we need to bound the replication factor of $\Pi(k)$. Below we first review the assignment rules of HDRF, and identify cases we need to analyze. Based on these cases, we then prove that the replication factor is bounded by $f' + \frac{k}{n+k}\frac{|E|}{|V|} - \frac{|V_1|}{2\cdot|V|}$, where V_1 is the number of vertices in the selected edges.

HDRF cases. To identify the cases, we first review the rules of assigning edges in HDRF [Petroni et al., 2015]. When $\lambda = 1$, given an edge (u,v), HDRF applies the following four rules.

(Rule 1). If none of u and v appears in the new fragments, then (u,v) is assigned to the fragment with the smallest size.

(Rule 2). If only u (resp. only v) appears in the new fragments, then the edge (u,v) is assigned to the fragment that contains u (resp. v) and has the smallest size.

(Rule 3). If there exist fragments containing both u and v, then the edge (u,v) is assigned to such a fragment that contains both u and v, and has the smallest size.

(Rule 4). If both u and v appear in the new fragments, and there does not exist a fragment containing both u and v, then the edge (u,v) is assigned as follows: (i) when the degree of u is smaller than that of v, the edge (u,v) is assigned to such a fragment that contains u and has the smallest size. (ii) otherwise, the edge (u,v) is assigned to such a fragment that contains v and has the smallest size.

When there exist multiple fragments satisfying the requirements, we randomly select one and assign the edge to it.

Then based on the changes of the edge assignment during dynamic scaling, we have the following cases.

Consider case F in Table A.1. It means that before dynamic scaling, edge (u,v) is assigned by rule 3 above, but during the dynamic scaling, this edge is assigned by

Table A.1: Changes of the edge assignment rules

	rule 1	rule 2	rule 3	rule 4
rule 1	A	×	×	×
rule 2	×	B	×	×
rule 3	×	×	D	F
rule 4	×	C	E	G

using rule 4. The other cases can be interpreted similarly.

Some cases do not happen (marked ×). For example, an edge (u,v) that is initially assigned by rule 1 cannot be reassigned by rule 3. Indeed, if (u,v) is initially assigned by rule 1, it is the first time that vertices u and v appear in the stream. Since we select edges based on timestamps of edges, u and v cannot appear in the new fragments before the assignment of (u,v). Hence we cannot use rule 3 to assign this edge. Similarly we interpret other cases that do not happen.

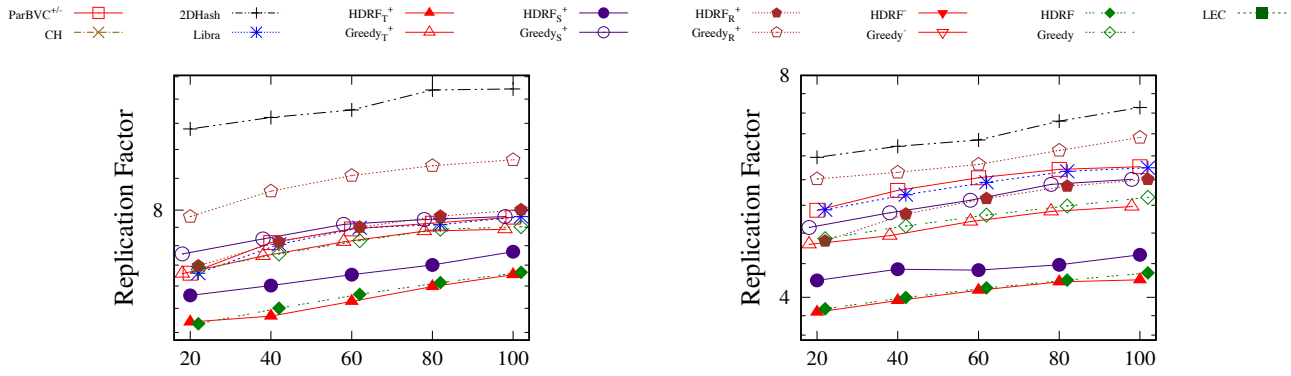
Analysis of the upper bound. Using Table A.1, we can show that when reassigning the edges, the replication factor cannot increase more than $\frac{k}{n+k}|E| - \frac{|V_1|}{2}$. That is, the replication factor of the partition $\Pi(k)$ is no larger than $f' + \frac{k}{n+k}\frac{|E|}{|V|} - \frac{|V_1|}{2\cdot|V|}$. More specifically, we show that the change of replication factor in all the cases in Table A.1 is either no larger than f' or no larger than $\frac{k}{n+k}\frac{|E|}{|V|} - \frac{|V_1|}{2\cdot|V|}$.

We group and analyze the cases as follows.

(1) For cases A, B, D and G, since the assignment rules for these edges are the same, the changes of the replication factor remain the same before and during the dynamic scaling. Then this part of replication factor is bounded by f'.

(2) For case C, it happens when (i) both u and v have been assigned before handling (u,v), and are in different fragments before dynamic scaling; and (ii) only one of them has been assigned before handling the edge (u,v) during dynamic scaling. Since both rule 2 and rule 4 increase the replication factor by $\frac{1}{|V|}$, we know that the replication factor after dynamic scaling for this case is bounded by f'.

(3) For case E, the replication factor is increased by one during the original partitioning by using rule 4, but it will remain the same during dynamic scaling by using rule 3. This the replication factor for this case is bounded in f'.



(a) Varying k (scale out, Twitter)

(b) Varying k (scale out, PLD)

Figure A.3: Replication factor

(4) For case F, it happens when both u and v have been reassigned before assigning the edge (u,v). Moreover, they are in the same fragment before dynamic scaling, but in different fragments during the scaling. Since (i) rule 4 only adds one new vertex, (ii) this case happens when v_1 and v_2 have been assigned, and (iii) reassigning $|V_1|$ vertices needs at least $\frac{|V_1|}{2}$ edges, we know that the total number of new copies of the vertices in case F is at most $\frac{k}{n+k}|E| - \frac{|V_1|}{2}$.

Putting these together, we know that the replication factor after HDRF$^+$ is bounded by $f' + f'' + \frac{k}{n+k}\frac{|E|}{|V|} - \frac{|V_1|}{2\cdot|V|}$. □

A.10 More Experimental Results

We next report more experiment results.

Replication factor. Fixing $n = 96$, we varied k from 20 to 100 (resp. 10 to 50) for scaling out (resp. in) on Twitter and PLD. The results are shown in Figures A.3(a) and A.3(b).

(1) HDRF$^+$ still delivers the best replication factor. On average, it outperforms Greedy$^+$, ParBVC$^+$ and CH by 1.2, 1.3 and 3.7 times, respectively, up to 1.3, 1.4 and 4.2 times.

(2) HDRF$^+$ also does better than re-partitioning approaches Libra, 2DHash and Greedy on average by 1.3, 1.7 and 1.2 times, respectively, up to 1.4, 1.9 and 1.3 times.

(3) Algorithms ParBVC$^+$ and Libra outperform the other hash-based algorithms CH

and 2DHash by 2.9 and 1.4 times, respectively, up to 3.5 and 1.6 times. ParBVC$^+$ and Libra have comparable replication factors since they both adopt the degree-based approach to improve locality.

(4) The results of scaling in are consistent (not shown).

Edge selection strategy. To evaluate the effectiveness of edge selection strategies for HDRF$^+$ and Greedy$^+$, we also implemented a strategy that randomly chooses edges for scaling out, denoted by HDRF$^+_{\text{R}}$ and Greedy$^+_{\text{R}}$, respectively.

(1) As shown in Figures 6.1(i)-6.1(n), HDRF$^+_{\text{T}}$ achieves the best replication factor among the scaling out algorithms derived from HDRF. On average it outperforms HDRF$^+_{\text{S}}$ and HDRF$^+_{\text{R}}$ by 1.2 and 1.6 times, respectively, up to 1.4 and 1.8 times. As remarked earlier, HDRF$^+_{\text{T}}$ even has a slight better replication factor than HDRF in most cases. This verifies the effectiveness of the timestamp-based strategy.

(2) The replication factor of HDRF$^+_{\text{S}}$ is also comparable to that of HDRF. On average it is 11.9% larger than that of HDRF, up to 18.1%. Moreover, HDRF$^+_{\text{S}}$ has better efficiency and migration cost than HDRF as we have seen in Exp-1.

(3) The balance factors of HDRF$^+_{\text{T}}$ and HDRF$^+_{\text{S}}$ are as good as that of HDRF since they use the same balancing mechanism.

(4) The results of the timestamp-based and score-based Greedy$^+$ are consistent with HDRF$^+$. On average, Greedy$^+_{\text{T}}$ is slightly better than Greedy$^+_{\text{S}}$ and Greedy in replication factor, and is 1.6 times better than Greedy$^+_{\text{R}}$, up to 1.8 times.

These demonstrate the score-based and timestamp-based strategies perform well for stream partitioners.