

# Supporting On-demand Elasticity in Distributed Graph Processing

Mayank Pundir, Manoj Kumar, Luke M. Leslie, Indranil Gupta, Roy H. Campbell  
{pundir2, mkumar11, lmlesli2, indy, rhc}@illinois.edu  
University of Illinois at Urbana-Champaign

**Abstract**—While distributed graph processing engines have become popular for processing large graphs, these engines are typically configured with a static set of servers in the cluster. In other words, they lack the flexibility to scale-out or scale-in the number of servers, when requested to do so by the user. In this paper, we propose the first techniques to make distributed graph processing truly elastic. While supporting on-demand scale-out/in operations, we meet three goals: i) perform scale-out/in without interrupting the graph computation, ii) minimize the background network overhead involved in the scale-out/in, and iii) mitigate stragglers by maintaining load balance across servers. We present and analyze two techniques called **Contiguous Vertex Repartitioning (CVR)** and **Ring-based Vertex Repartitioning (RVR)** to address these goals. We implement our techniques in the LFGGraph distributed graph processing system, and incorporate several systems optimizations. Experiments performed with multiple graph benchmark applications on a real graph indicate that our techniques perform within 9% and 21% of the optimum for scale-out and scale-in operations, respectively.

## I. INTRODUCTION

Large graphs are increasingly common – examples include online social networks such as Twitter and Facebook, Web graphs, Internet graphs, biological networks, and many others. As a consequence, distributed graph processing has become an attractive solution for batch processing of large graphs. Google’s Pregel [21] and GraphLab [20] were two of the first such distributed graph processing engines. Subsequently, the research community has developed more efficient engines that adopt the vertex-centric approach for graph processing, such as LFGGraph [13], PowerGraph [9] and GPS [25].

Today’s graph processing frameworks operate on statically allocated resources; the user must decide resource requirements before an application starts. However, part-way through computation, the user may desire to *scale-out* by adding more machines (e.g., to speed up the computation), or *scale-in* by lowering the number of machines (e.g., to reduce hourly costs). The capability to scale-out/in when required by the user is called *on-demand elasticity*. Alternatively, an adaptive policy may request scale-out or in.<sup>1</sup> Such a concept has been explored for datacenters [7] [31], cloud systems [14] [22] [26], storage

This work was supported in part by the following grants: NSF CNS 1409416, NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, and generous gifts from Microsoft and Google.

<sup>1</sup>This paper does not deal with the details of adaptive policies or triggers. We focus instead on the on-demand mechanisms, which are an important building block for such policies.

systems [5] [6] [23] [28] [29] and data processing frameworks such as Hadoop [11] [12] and Storm [2]. However, on-demand elasticity remains relatively unexplored in batch distributed graph processing systems.

While partitioning techniques have been proposed to optimize computation and communication [9], these techniques partition the entire graph across servers and are thus applicable only at the start of the graph computation. On-demand elasticity requires an *incremental* approach to (re-)partitioning vertices on-demand. Solving the problem of on-demand elasticity is also the first step towards adaptive elasticity (e.g., satisfying an SLA in a graph computation), for which our techniques may be employed as black boxes.

A distributed graph processing system that supports on-demand scale-out/in must overcome three challenges:

- 1) *Perform scale-out/in without interrupting graph computation.* A scale-out/in operation requires a re-assignment of vertices among servers. During scale-out, new servers must obtain some vertices (and their values) from existing servers. Similarly, during scale-in, vertices from the departing servers must be re-assigned to the remaining servers. These transfers must be done while minimally affecting ongoing computation times.
- 2) *Minimize the background network overhead involved in the scale-out/in.* To reduce the effect of the vertex transfer on computation time, we wish to minimize the total amount of vertex data transferred during scale-out/in.
- 3) *Mitigate stragglers by maintaining load balance across servers.* Graph processing proceeds in iterations and stragglers will slow the entire iteration down. Thus, while re-assigning vertices at the scale-out/in point, we aim to achieve load balance in order to mitigate stragglers and keep computation time low.

Our approach to solving the problem of on-demand elasticity and overcoming the above challenges is motivated by two critical questions:

- 1) **How (and what) to migrate?** Which vertices from which servers should be migrated in order to reduce the network transfer volume and maintain load balance?
- 2) **When to migrate?** At what points during computation should migration begin and end?

To answer the first question, we present and analyze two new techniques. The first, called *Contiguous Vertex Repartitioning*

(CVR), achieves load balance across servers. However, it may result in high overhead during the scale-out/in operation. Thus, we propose a second technique, called *Ring-based Vertex Repartitioning (RVR)*, that relies on ring-based hashing to lower the overhead. To address the second question (when to migrate) concretely, we integrate our techniques into the LFGGraph graph processing system [13], and use our implementation to carefully decide when to begin and end background migration, and when to migrate static vs. dynamic data. We also use our implementation to explore systems optimizations that make migration more efficient.

We performed experiments with multiple graph benchmark applications on a real Twitter graph with 41.65 M vertices and 1.47 B edges. Our results indicate that our techniques are within 9% of an optimal mechanism for scale-out operations and within 21% for scale-in operations.

## II. HOW (AND WHAT) TO MIGRATE?

In this section, we address the question of which vertices to migrate when the user requests a scale-out/in operation.

Most existing distributed graph processing systems use the vertex-centric synchronous Gather-Apply-Scatter (GAS) decomposition [9] [13] [20] [21] [25], and our techniques are intended for such systems.<sup>2</sup> In vertex-centric graph processing, a vertex is the basic unit of processing. Graph processing frameworks partition vertices or edges across servers. Computation then proceeds in iterations (sometimes called super-steps). In the synchronous GAS decomposition, each iteration comprises of *gather*, *apply*, and *scatter* stages, wherein each vertex first gathers values from its neighbors, processes and applies the result, and then scatters the updated value to its neighbors. Synchronization across servers may be enforced either during intermediate stages (e.g., PowerGraph), or only at the end of the iteration (e.g., LFGGraph).

**Assumptions:** While some graph processing engines have proposed “intelligent” mechanisms to partition vertices across servers before computation starts (e.g., PowerGraph [9]), recent studies have shown that these systems may consume a large fraction of their total run time doing so [13]. For instance, [13] demonstrated that when using PowerGraph to run PageRank on the Twitter graph with 8 servers, 80% of the total runtime was constituted by the intelligent partitioning operation, with subsequent iterations only taking up 20%. As a result, [13] concludes that intelligent partitioning techniques are not worth the cost, and shows that hash-based partitioning is simpler and can result in a faster graph computation time.

As a result of these studies, we assume that our distributed graph processing engine relies on hash-based partitioning. Our contributions are applicable to all major graph processing engines, including LFGGraph [13], Giraph [1], PowerGraph [9] and GPS [25], all of which support hash-based partitioning.

In addition, we assume the graph can fit in-memory after any scale-in operation. We also assume that no other jobs are

deployed alongside specialized distributed graph processing systems. Papers such as [4] reinforce the second assumption, alongside the reality that multi-tenancy for many of these specialized systems is not well supported or even existent.

In this remainder of this section, we present two techniques for on-demand elasticity in graph processing. Both techniques assume hash-based partitioning, which works by consistently hashing vertices to a vertex space using a function that assigns  $m$  bit values to vertices (e.g., using a hash function like SHA-1). The resulting hashed vertex space ranges from 0 to  $2^m - 1$ , where  $m$  may be set high to minimize collisions. Our two techniques deal with two different and popular ways in which hash-based partitioning is done in graph processing systems.

### A. Contiguous Vertex Repartitioning (CVR)

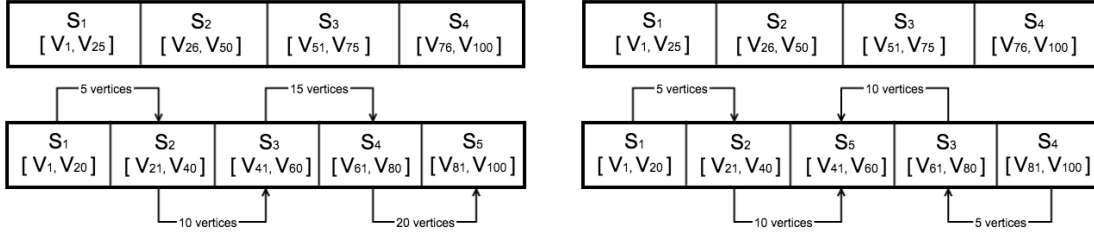
Our first technique assumes that the hashed vertex space is divided into as many partitions as there are servers, and each server is assigned one partition. Partitions are equi-sized in order to accomplish load-balancing. The top of Figure 1a illustrates an example graph containing 100 vertices, split across 4 servers. The vertex sequence (i.e.,  $V_i$ ) is random but consistent due to our use of consistent hashing, and is split into four equi-sized partitions which are then assigned to servers  $S_1 - S_4$  sequentially.

Upon a scale-out/in operation, the key problem we need to solve is: how do we assign the (new) equi-sized partitions to servers (one partition per server), such that network traffic volume is minimized? For instance, the bottom of Figure 1 shows the problem when scaling out from 4 to 5 servers. To solve this problem, we now: 1) show how to reduce it to a graph matching problem, and 2) propose an efficient heuristic.

When we scale-out/in, we repartition the vertex sequence into equi-sized partitions. Assigning these new partitions in an arbitrary fashion to servers may be sub-optimal and transfer large amounts of vertex data across the network. For instance, in the bottom of Figure 1a, we scale-out by adding one server, resulting in five new partitions. Merely adding the new server to the end of the server sequence and assigning partitions to servers in that order results in 50 total vertices being moved. On the other hand, Figure 1b shows the optimal solution for this example, wherein adding the new server in the middle of the partition sequence results in moving only 30 vertices.

To achieve the optimal solution, we consider the scale-out problem formally (the solution for scale-in is analogous and excluded for brevity). Let the cluster initially have  $N$  servers  $S_1, \dots, S_N$ . With a graph of  $V$  vertices, initially the size of each partition  $P_i^{old}$  is  $M^{old} = \frac{V}{N}$ , where  $1 \leq i \leq N$ . Each  $j$ -th vertex id is first hashed, and the resulting value  $V_j$  is initially used to assign it to partition  $P_i^{old}$  where  $i = \left\lceil \frac{V_j}{M^{old}} \right\rceil$ . If we add  $k$  servers to this cluster, the size of each new partition becomes  $M^{new} = \frac{V}{N+k}$ . We label these new partitions as  $P_i^{new}, 1 \leq i \leq N+k$ , and assign each  $j$ -th vertex, as usual, to a new partition by first hashing the vertex id and using the resulting hash  $V_j$  to partition  $P_i^{new}$  where  $i = \left\lceil \frac{V_j}{M^{new}} \right\rceil$ .

<sup>2</sup>We focus on specialized graph processing systems and exclude dataflow systems such as GraphX [10] and Pregelix [3].



(a) Sub-optimal partition assignment with total vertex transfer = 50 (b) Optimal partition assignment with total vertex transfer = 30

Figure 1: Scale-out from 4 (top) to 5 (bottom) servers using Contiguous Vertex Repartitioning (CVR). Fewer vertices are transferred in the optimal partition assignment (30 vs. 50).

Next, we create a bipartite graph  $B$ , which contains: i) a left set of vertices, with one vertex per new partition  $P_i^{new}$ , and ii) a right set of vertices, with one vertex per server  $S_j$ . Each of the left and right sets contains  $(N+k)$  vertices. The result is a complete bipartite graph, with the edge joining a partition  $P_i^{new}$  and server  $S_j$  associated with a cost. The cost is equal to the number of vertices that must be transferred over the network if partition  $P_i^{new}$  is assigned to server  $S_j$  after scale-out. In other words, the cost is equal to  $|P_j^{old} \cap P_i^{new}| = |P_j^{old}| + |P_i^{new}| - |P_j^{old} \cup P_i^{new}|$ .

The problem of minimizing network transfer volume now reduces to finding a minimum-cost perfect matching in  $B$ . This is a well-studied problem, and an optimal solution can be obtained by using the Hungarian algorithm [16]. However, the Hungarian algorithm has  $O(N^3)$  complexity [15], which may be prohibitive for large clusters.

As a result, we propose a greedy algorithm that iterates sequentially through  $S_1, \dots, S_N$ , in that order.<sup>3</sup> For each server  $S_j$ , the algorithm considers the new partitions with which it has a non-zero overlap – due to the contiguity of partitions, there are only  $O(1)$  such partitions. Among these partitions, that which has the largest number of overlapping vertices with  $P_j^{old}$  is assigned to server  $S_j$ . Due to the linear order of traversal, when  $S_j$  is considered,  $S_j$  is guaranteed to have at least one (overlapping) candidate position. This makes the greedy algorithm run efficiently in  $O(N)$ . For example, in Figure 1b, to determine the new partition for  $S_1$ , we need to consider only two partitions  $[V_1, V_{20}]$  and  $[V_{21}, V_{40}]$ . Next, we need to consider partitions  $[V_{21}, V_{40}]$  and  $[V_{41}, V_{60}]$  and so on. Section II-A provides a mathematical analysis of the optimal server ordering that minimizes the network transfer volume.

We next analyze CVR for scale-out operations, and then for scale-in operations. All proofs can be found in Appendix A.

1) *Scale-out*: Assume a computation is running on a graph with  $V$  vertices, and suppose the user performs a scale-out operation by adding one server. Consider a case where the order of the old servers is retained, but the new server is inserted at a position  $k$ , i.e., right after server  $S_k^{old}$  but before  $S_{k+1}^{old}$ ,  $0 \leq k \leq N$  (there are  $N+1$  potential locations for

adding the new server). We identify which insertion position minimizes network traffic volume (proof is in Appendix A).

**Theorem 1.** *Consider a cluster of  $N$  servers running a computation on a graph with  $V$  vertices. Initially, the system is load balanced (each server has  $\frac{V}{N}$  vertices). If we add a new server to the cluster using our CVR approach, without changing the order of the old servers, then the total network transfer volume is minimized if the new server is added immediately after old server  $S_{\frac{N}{2}}$ , i.e., halfway through the list of old servers.*

2) *Scale-in*: Assume a computation running on a graph with  $V$  vertices and suppose the user performs a scale-in operation by removing one server. We assume the user only specifies the number of servers to remove, and we have flexibility in selecting *which* server to remove – this is a reasonable assumption, for instance, in graph computations running on virtualized platforms. (If the user specifies the particular server to remove, the problem is not challenging.)

We consider a case where the order of the old servers is retained but the leaving server is removed from position  $k$ , i.e., right after server  $S_{k-1}^{old}$  but before  $S_{k+1}^{old}$  ( $1 \leq k \leq N$ ). This gives us  $N$  total choices for removing the leaving server. Similarly to Theorem 1, we now state which of these positions minimizes network traffic volume (proof is in Appendix A).

**Theorem 2.** *Consider a cluster of  $N$  servers running a computation on a graph with  $V$  vertices. Initially, the system is load balanced (each server has  $\frac{V}{N}$  vertices). If we remove one server from the cluster using our CVR approach, then the total network transfer volume is minimized if the leaving server is removed immediately after old server  $S_{\frac{N+1}{2}}$ , i.e., halfway through the list of old servers. The minimal number of vertices transferred in this case is  $C_{si}^* = \frac{V \times (N+1)}{4 \times N}$ .*

While the technique discussed in this section ensures balanced partitions, it may require a large number of partitions to be re-assigned. This affects a large number of servers because they all transfer vertices, which in turn reduces the resources they can utilize in the ongoing iteration of the graph computation, and prolongs completion time. This drawback motivates our next approach.

<sup>3</sup>A similar problem was studied in [8], but their approach is not load-balanced.

## B. Ring-based Vertex Repartitioning (RVR)

In this technique, we assume an underlying hash-based partitioning that leverages Chord-style consistent hashing [28]. To maintain load balance, servers are not hashed directly to the ring, but instead (as in Cassandra [18] and Riak [24]), we assume each server is assigned an equi-sized segment of the ring. Concretely, a server with ID  $n_i$  is responsible for vertices hashed in the interval  $(n_{i-1}, n_i]$ , where  $n_{i-1}$  is  $n_i$ 's predecessor.

Under this assumption, performing a scale-out/in operation itself is straightforward – a joining server splits a segment with its successor, while a leaving server gives up its segment to its successor. For instance, in a scale-out operation involving one server, the affected server receives its set of vertices from its successor in the ring, i.e., a server  $n_i$  takes the set of vertices  $(n_{i-1}, n_i]$  from its successor  $n_{i+1}$ . Scale-in operations occur symmetrically: a leaving server  $n_i$  migrates its vertex set  $(n_{i-1}, n_i]$  to its successor  $n_{i+1}$  which will now be responsible for the set of vertices in  $(n_{i-1}, n_{i+1}]$ .

More generally, we can state that: a scale-out/in operation, which involves  $k$  servers being added or removed at once affects at most  $k$  existing servers. If some of the joining or leaving servers have segments that are adjacent, the number of servers affected would be smaller than  $k$ .

While this technique is minimally invasive to existing servers and the ongoing graph computation, it may result in load imbalance. We can mitigate load imbalance for the scale-out case by intelligently deciding which point on the ring to add the new server(s). For the scale-in case, we can intelligently decide which server(s) to remove.

We next analyze our RVR technique for scale-out/in operations with one server, and then with multiple servers. All proofs appear in Appendix B.

1) *One Server*: We can state the following straightforward theorems for our RVR technique:

**Theorem 3.** *For a scale-out operation involving adding one server to a balanced cluster, the minimal number of vertices transferred by our RVR technique is  $\frac{V}{2N}$ .*

**Theorem 4.** *For a scale-in operation involving removing one server from a balanced cluster, the number of vertices transferred by our RVR technique is  $\frac{V}{N}$ .*

In the former theorem, this minima occurs when the new server is added half-way between two existing (old) servers.

2) *Multiple Servers*: As in Section II-A2, we assume that we are given the number of servers to remove but have flexibility in selecting which servers. When scaling-out/in with multiple servers, more than one partition may be affected. In both cases, our approach is to “spread out” the affected portions over the ring over disjoint segments and consequently preserving load balance. For instance, if we add servers to disjoint old partitions in the ring during scale-out, then as long as the cluster adds fewer than  $(N + 1)$  additional servers, the result of Theorem 3 holds. Similarly, as long as fewer than  $(\frac{N}{2} + 1)$  servers are removed during scale-in, alternating servers can be removed and the result of Theorem 4 holds.

However, RVR does not work when scaling out with more than  $N$  servers or scaling in with more than  $N/2$  servers. In this case, we can more generally state the following two theorems (proofs are in Appendix B).

**Theorem 5.** *Consider a cluster with  $N$  servers running a graph computation. Initially, each server has  $\frac{V}{N}$  vertices. Consider a scale-out operation which adds  $k > N$  servers to the cluster using RVR. Some subgroups of these new servers will be added to the same old partition (we assume they are equally distributed inside the old partition). Let  $m$  be the maximum size of such a group, i.e., the maximum number of new servers added to an old partition. Then, the approach of spreading out is minimax (i.e., minimizes the maximum loaded server) if  $(m - 1) \times N < k \leq m \times N$ .*

**Theorem 6.** *Consider a cluster with  $N$  servers running a graph computation; initially each server has  $\frac{V}{N}$  vertices. Consider a scale-in operation which removes  $k > N/2$  servers from the cluster using RVR. Some subgroups of these servers will be removed contiguously from the ring. Let  $m$  be the maximum size of such a group, i.e., the maximum number of servers removed from an old partition. Then, the approach of spreading in is minimax (i.e., minimizes the maximum loaded server) if  $\frac{(m-1) \times N}{m} < k \leq \frac{m \times N}{m+1}$ .*

## C. Similarities

While CVR and RVR are different ways of hash-based partitioning, there are some corner cases where our techniques output the same new assignment of vertices to servers. One of them is shown below (proof is in Appendix C).

**Theorem 7.** *Consider a cluster with  $N$  servers each having  $\frac{V}{N}$  vertices. If we add  $m \times N$  servers or remove  $\frac{m \times N}{m+1}$  servers (for  $m \geq 1$ ) using CVR, then the optimal position of servers to be added or removed is same as their position with RVR.*

## III. WHEN TO MIGRATE?

Given the knowledge of which vertices must be migrated and to where, we must now decide *when* to migrate them in such a way that minimizes interference with normal execution. Two types of data need to be migrated between servers: (i) static data, including sets of vertex IDs, neighboring vertex IDs and edge values to neighbors, and (ii) dynamic data, including the latest values of vertices and latest values of neighbors. Static data corresponds to graph partitions, while dynamic data represents computation state. Once this migration is completed, the cluster can switch to the new partition assignment.

### A. LFGraph Overview

We have incorporated our elasticity techniques into LFGraph [13], a vertex-centric graph processing framework. We have chosen LFGraph because it offers low computation and communication load, low pre-processing cost and low memory consumption. However, our proposed techniques are not limited to LFGraph and can be incorporated into any synchronous framework configured to use hash-based partitioning, such as Giraph [1], PowerGraph [9], and GPS [25].

LFGGraph uses hash-based partitioning of vertices across servers for efficient pre-processing. Each server’s partition is further divided into equi-sized vertex groups so that each group can be assigned to a separate worker thread. Gather and scatter phases across servers are independent of each other but are synchronized using a barrier server.

### B. Executing Migration

LFGGraph uses a publish-subscribe mechanism. Before the iterations start, each server subscribes to in-neighbors of the vertices hosted by the server. Based on these subscriptions, servers build a publish list for every other server in the cluster. After each iteration, servers send updated values of the vertices present in the publish lists to the respective servers. After a scale-out/in operation, we perform the publish-subscribe phase again to update the publish lists of servers.

1) *First-cut Migration*: A first-cut approach is to perform migration of both static and dynamic data during the next available barrier synchronization interval. However, when we implemented this approach, we found that it added significant overheads by prolonging that iteration. As a result, we introduce our next two optimizations.

2) *Static Data Migration*: This technique is based on the observation that static data can be migrated in the background while computation is going on. Recall that static data consists of vertex IDs, their neighboring vertex IDs and edge values to neighbors. Only dynamic data (vertex values and neighboring vertex values) needs to wait to be migrated during a barrier synchronization interval (i.e., after such data is last updated). This reduces the overhead on that iteration.

3) *Dynamic Data Migration*: LFGGraph has two barrier synchronization intervals. One interval is between the gather and scatter phases and the other is after the scatter phase. This gives us two options for the transfer of dynamic data. We perform dynamic data transfer and cluster re-configuration in the barrier synchronization interval *between* the gather and scatter phases. This enables us to leverage the default scatter phase to migrate neighboring vertex values. The scatter phase simply considers the new set of servers in the cluster while distributing updated vertex values. This optimization further reduces the overhead on that iteration.

A scale-out/in operation that starts in iteration  $i$  ends in iteration  $i+2$ . Background static data migration occurs in iterations  $i$  and  $i+1$  while vertex value migration occurs after the gather phase of iteration  $i+2$ . At this point, computation continues on the new set of servers. The performance impact due to background data migration is more in iteration  $i$  than in iteration  $i+1$ , i.e., iteration times are longer in iteration  $i$ . This is because a majority of the migration happens in iteration  $i$ . In iteration  $i+1$ , servers build their new subscription lists for the publish-subscribe phase.

To demonstrate further, we now describe the steps involved in a scale-out: (1) The joining server sends a *Join* message containing its IP address and port to the barrier server at the start of iteration  $i$ . (2) The barrier server responds with a *Cluster Info* message assigning the joining server an ID and

the contact information of the servers from which it should request its vertices. (3) Additionally, the barrier server sends an *Add Host* message to all servers informing them about the new server in the cluster. (4) The joining server requests its vertices with a *Vertex Request* message. (5) After receiving its vertices, it informs the barrier server that it can join the cluster with a *Ready* message. Concurrently, the servers start updating their subscription lists to reflect the modifications in the cluster servers (this is our Background subscription lists optimization discussed in Section IV). (6) The barrier server sends a *Re-configure* message to the servers in the synchronization interval after gather phase of iteration  $i+2$ . (7) Upon receiving the *Re-configure* message, joining servers request the vertex values with a *Vertex Value Request* message. Additionally, all servers update their vertex-to-server mapping to reflect newly added servers. (8) The scatter phase of iteration  $i+2$  executes with this new mapping. From this point on, computation proceeds on the new set of servers.

4) *Role of Barrier Server*: In our repartitioning techniques, the barrier server accepts join and leave requests and determines an optimal partition assignment. We adopted this approach instead of a fully decentralized re-assignment because of two reasons: i) fully decentralized re-assignment may lead to complex race conditions, and ii) the barrier server, being initialized, has the capability to obtain per-server iteration run times via the barrier synchronization messages and assigns new servers to alleviate the load on the busiest servers.

## IV. OPTIMIZING MIGRATION

In this section, we discuss specific performance optimizations available for migration in frameworks such as LFGGraph.

### A. Parallel Migration

LFGGraph uses consistent hashing to split the vertices assigned to a server (a partition) further into *vertex groups*, with each vertex group assigned to a separate thread [13]. If two servers are running the same number of threads, we can use a parallel migration optimization that works as follows. The (consistent hashing) ring inside each server is identical, as are the segments of the ring (each segment corresponding to a vertex group). As a result, there is a one-to-one correspondence between segments (and thus threads) across two servers. On a scale-out/in operation, data can be transferred directly and parallelly between these corresponding threads.

### B. Efficiency vs. Overhead Trade-off

Speeding up the background migration via parallelism slows the ongoing computation. On the other hand, if migration is performed serially, the time taken for transferring vertex data would increase. To achieve a balance between the two, we propose two further optimizations:

- *Thread-Count*: Based on experimental results, we use half as many threads as vertex groups. The result is a conservative use of the network resource, at the expense of slower background migration.

- *Background Subscription Lists*: We allow nodes to start receiving information in the background (from the barrier server) about joining or leaving servers. This allows them to start building subscription lists before the actual cluster re-configuration phase. This action can be performed in parallel for each vertex group. However, based on experimental results, we perform it serially to reduce the impact on ongoing computation. Thus, we utilize parallelism between foreground and background processes but not across the tasks done in background.

### C. Ring-based Optimizations

For scale-in operations using RVR, two further optimizations are possible:

- *Modified Scatter*: We perform a modified scatter phase after vertex transfer where servers transfer vertex values, subscribed by leaving servers, to their respective successors. Additionally, we leverage this scatter phase to have a leaving server transfer all vertex values to its successor while only subscribed values are sent to other servers. This optimization allows the scatter phase to be performed in parallel with cluster re-configuration after scale-out/in operation. Additionally, it avoids using an explicit vertex value migration step during the barrier interval as leaving servers migrate the vertex values as part of the modified scatter phase.
- *Subscription Lists*: For a scale-in, we quickly rebuild subscription lists at each remnant (non-leaving) server by merely appending the subscription list for a leaving server  $S$  to  $S$ 's next remnant successor – in RVR,  $S$ 's successor inherits leaving server  $S$ 's vertices. This avoids rebuilding subscription lists from scratch at remnant servers.

## V. EVALUATION

In this section, we experimentally evaluate the efficiency and overhead of our elasticity techniques.

### A. Experimental Setup

We perform our experiments with both our CVR and RVR techniques on virtual instances each having 16 GB RAM and 8 VCPUs. We use a Twitter graph [17] containing 41.65 M vertices and 1.47 B edges (with larger graphs, we expect similar performance improvements). We evaluate our techniques using five graph benchmarks: PageRank, single-source shortest paths (SSSP), connected components, k-means clustering and multiple-source shortest paths (MSSP).

### B. Scale-out and Scale-in

Our first set of experiments measures the overhead experienced by the computation due to a scale-out operation. Figure 2 shows two experiments that perform a scale-out from  $X$  servers to  $2X$  servers (for  $X \in \{5, 10, 15\}$ ), with the scale-out starting at iteration  $i = 1$  and ending at iteration 3. The vertical axis plots the *per-iteration* run time. For comparison, we plot the per-iteration times for a run with  $X$  servers throughout, and a run with  $2X$  servers throughout.

From Figures 2a, 2b and 2c, we observe that: i) both CVR and RVR appear to perform similarly, and ii) after the scale-out operation is completed, the performance of the scaled-out system converges to that of a cluster with  $2X$  servers, demonstrating that our approaches converge to the desired throughput after scale-out.

Similarly, Figure 3 shows the plots for scale-in from  $2X$  servers to  $X$  servers (for  $X \in \{5, 10, 15\}$ ). Once again, the cluster converges to the performance of  $X$  servers. However, RVR has a lower overhead than CVR – this is due to the Ring-based optimizations discussed for scale-in in Section IV-C.

### C. Repartitioning Overhead

Figures 2 and 3 show the per-iteration time under different scenarios, and demonstrate that iterations take longer during scale-out/in. If the vertex migration had infinite bandwidth available for it, then the scale-out/in could be done *instantly* at the end of an iteration. We call this the *Infinite Bandwidth (IB)* technique. In a sense, IB is optimal in that the network overhead for transfer is not incurred at all. Comparing with IB thus gives us an indication of how much worse our approach is compared to the fastest possible network.

Figure 4 shows the overhead of our technique compared to IB. The overhead captures the extra time spent by our technique in running a graph computation during one scale-out/in operation occurring from iteration  $i = 1$  to  $i = 3$ . The overhead is computed as follows. First, we assume IB instantly scales-out/in after iteration  $i = 3$  – thus we plot its timeline as the timeline for  $X$  servers up to the third iteration and as the timeline for  $2X$  servers beyond the third iteration. We calculate the overhead as the area of the plot between our technique's line and the IB line.

Figure 4 shows that for PageRank our techniques result in less than 5% total overhead for scale-out compared to IB. For scale-in operations, RVR incurs less than 8% total overhead while CVR incurs less than 11% total overhead. Further, with increasing cluster size, the percentage overhead falls for both scale-out and scale-in – this is because less data must be migrated per server as the cluster size increases.

For short computations with one scale-out/in operation, affecting fewer servers, as achieved by RVR, is more important than achieving an optimal assignment of vertices to servers. However, the caveat is that if the graph computation were a very long-running one, and involved multiple scale-out/in operations during its lifetime, then RVR would create stragglers.

**Scale-out vs. Scale-in Overhead** We observe that the overhead numbers in Figure 4 for scale-in are higher than that of scale-out. This is purely an artifact of the way we constructed the IB technique. Comparing Figures 2a and 3a for instance, we notice that during scaling (iterations  $i = 1$  to 3), the absolute overhead for scale-out and scale-in are comparable (CVR line and RVR line both peak at about 26 s iteration time). However, for scale-in, IB's curve is much lower than it is for scale-out, because the former used a large number of servers prior to the scaling and thus the per-iteration time was lower. This results in the overhead (area between our

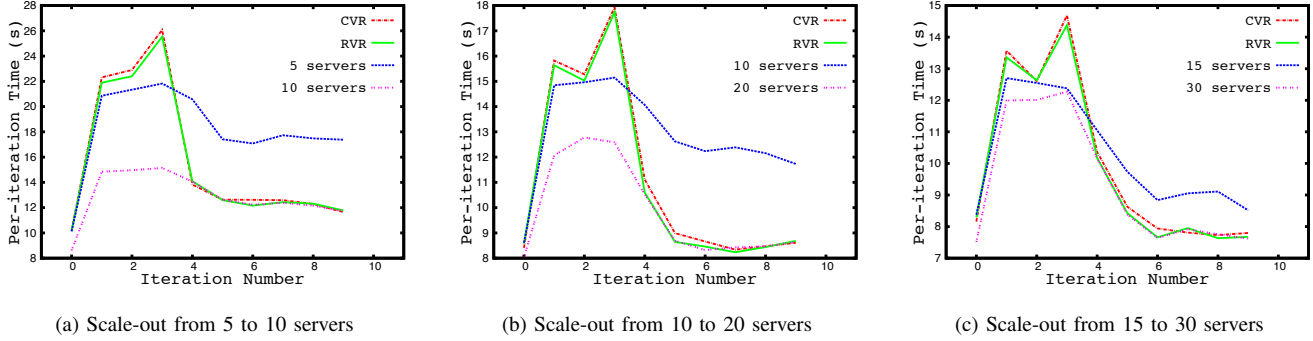


Figure 2: Per-iteration execution time with scale-out at iterations  $i = 1$  to  $i = 3$ , for different repartitioning strategies and cluster sizes.

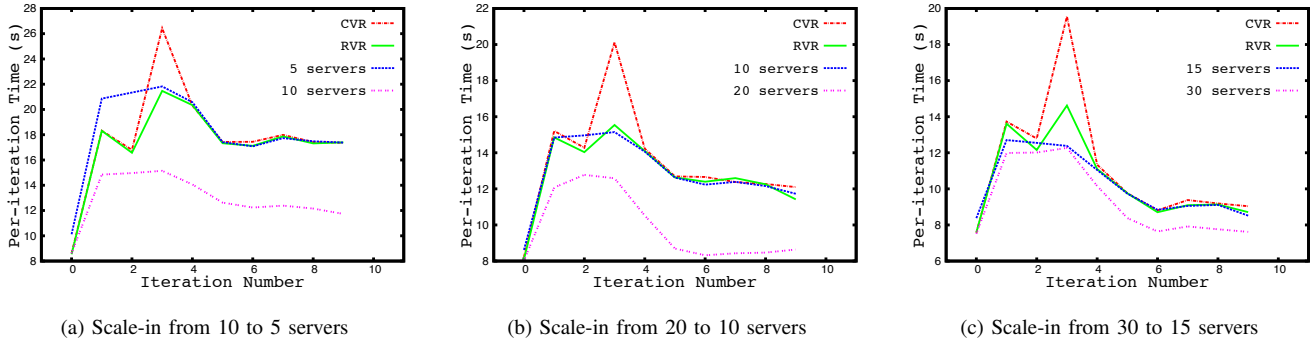


Figure 3: Per-iteration execution time with scale-in at iterations  $i = 1$  to  $i = 3$ , for different repartitioning strategies and cluster sizes.

technique’s line and IB’s line during iterations  $i = 1$  to 3) being measured as higher for scale-in than for scale-out.

#### D. Scale-out and Scale-in at Various Points in Computation

We evaluate the effect of starting the scale-out/in operation at different iterations in computation. Figure 5 shows this overhead with 5 to 10 servers scale-out and 10 to 5 servers scale-in performed at iterations  $i \in \{1, 2, 3, 4, 5, 6\}$  with the PageRank algorithm. The plot shows that our techniques incur less than 6% overhead during scale-out and less than 11% overhead during scale-in, compared to IB. The absolute overhead does not depend on the iteration number. However, due to shorter later iterations in PageRank, the total execution time of IB is larger if the scale-out is performed in later iterations and smaller if the scale-in is performed in later iterations. As a result, the percentage overhead increases if the scale-in is performed in later iterations (Figure 5b).

#### E. Scale-out and Scale-in for Various Algorithms

Figure 6 shows the overhead incurred with the four other graph benchmarks: SSSP, k-means clustering, connected components and MSSP. For k-means clustering, we chose 100 as the number of clusters. Even though k-means clustering spans multiple sets of iterations with different sets of clusters, we calculate overhead only in the set of iterations where the scale-out/in operation is performed. For MSSP, we computed the shortest paths to 10 landmark vertices in the graph.

The plots show that across algorithms, we incur less than 9% and 21% overhead for scale-out and scale-in respectively,

all compared to IB. The overheads for SSSP and k-means clustering algorithms are higher than others because of a smaller total execution time. The total execution time for SSSP and k-means clustering algorithms is within 80-90 s, while it is within 150-160 s for PageRank, within 120-130 s for connected components and within 190-200 s for MSSP. Although total execution time for MSSP is more than PageRank, MSSP incurs a higher overhead due to larger stored vertex values (shortest distance from 10 randomly chosen landmarks to the vertex).

Scale-out/in operations took 2 iterations to complete for connected components, but took 3 iterations for SSSP, k-means clustering and MSSP due to shorter initial iterations.

## VI. RELATED WORK

To the best of our knowledge, we are the first to explore elasticity for distributed graph computations. However, elasticity has been explored in many other areas.

### A. Data Centers

AutoScale [7] enables elastic capacity management for data centers. Its goal is to reduce power wastage by maintaining just enough server capacity for the current workload. [31] proposes solving the same problem using Model Predictive Control framework.

### B. Cloud and Storage Systems

CloudScale [26] and [14] propose mechanisms to scale-up VM resources based on predicted application needs in an Infrastructure as a Service (IaaS) environment. AGILE [22]



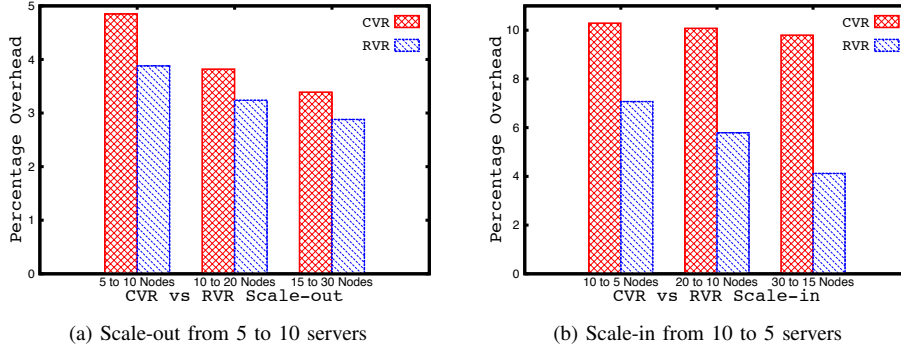


Figure 4: PageRank overhead comparison with the Infinite Bandwidth (IB) Technique.

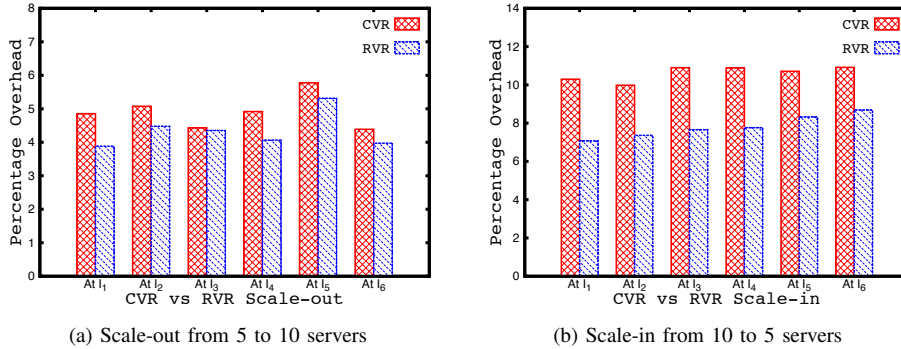


Figure 5: PageRank overhead comparison with the Infinite Bandwidth (IB) Technique with scale-out/in at different iteration times ( $I_i$  represents iteration  $i$ ).

proposes mechanisms to scale-out VM resources based on predicted application needs.

The authors of [23] propose a social partitioning and replication middleware to enable efficient storage of social network graphs. The technique enables the storage system to scale-out/in based on need. Albatross [5] enables scaling of multi-tenant databases using live migration. TIRAMOLA [29] allows NoSQL storage clusters to scale-out or scale-in by applying Markov Decision Process on the workload using user-provided policies. Transactional Auto Scaler [6] proposes another storage resource scaling mechanism that predicts workload behavior using analytical modeling and machine learning.

### C. Data Processing Frameworks

Starfish [12] performs tuning of Hadoop parameters at job, workflow and workload levels. The authors in [11] propose Elastisizer which has the capability to predict cluster size for Hadoop workloads. [2] proposes workload based scheduling of Storm topologies.

### D. Partitioning in Graph Processing

PowerGraph [9] performs vertex assignment using balanced partitioning of edges. The aim is to limit number of servers spanned by the vertices. Distributed GraphLab [19] proposes a two-phase assignment of vertices to servers. The first phase creates more partitions than the number of servers. In the second stage, servers load their respective partitions based on

a balanced partitioning. This allows the graph to be loaded in a distributed manner and the number of servers to change without affecting the first phase. The authors in [27] discuss partitioning strategies for streaming graph data. However, elasticity is not explored. Using partitioning during a scale-out/in operation is prohibitive as it partitions the entire graph – in comparison, we do incremental repartitioning.

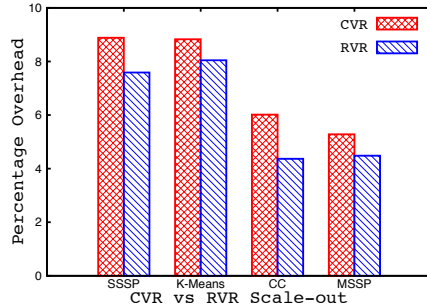
### E. Dynamic Repartitioning in Graph Processing

[30] proposes a vertex migration heuristic between partitions, to maintain balance and reduce communication cost. GPS [25] also proposes dynamic repartitioning of vertices by co-locating vertices which exchange more messages with each other. While these works do not explore on-demand elasticity explicitly, our techniques are orthogonal and can be applied to such systems. Concretely, our vertex re-partitioning technique (Section II-A) can be used in such systems by treating each partition as the set of vertices currently assigned to that server (instead of the hashed set of vertices, as we do).

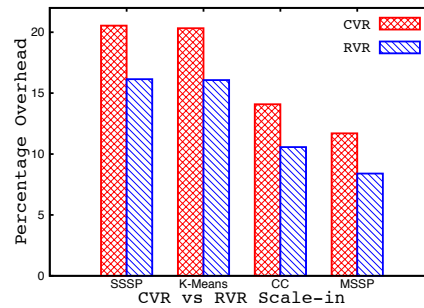
## VII. CONCLUSION

In this paper, we have proposed techniques to enable on-demand elasticity operations in distributed graph processing engines. Concretely, we proposed two techniques for deciding what vertices to migrate: Contiguous Vertex Repartitioning (CVR) and Ring-based Vertex Repartitioning (RVR). We also integrated these techniques into the LFGGraph graph processing





(a) Algorithms overhead comparison in scale-out



(b) Algorithms overhead comparison in scale-in

Figure 6: Overhead comparison with the Ideal Technique with 5 to 10 servers scale-out and 10 to 5 servers scale-in in single-source shortest paths (SSSP), k-means clustering, connected components (CC), and multiple-source shortest paths (MSSP).

engine, and incorporated several optimizations. Our experiments with multiple graph benchmarks and a real Twitter graph show that: i) compared to the optimal approach, our techniques incurred less than 9% overhead for scale-out and less than 21% overhead for scale-in operations, and ii) our techniques scale well.

## REFERENCES

- [1] Apache Giraph. <http://giraph.apache.org/>.
- [2] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive Online Scheduling in Storm," in *Proc. Int'l Conf. Distributed Event-Based Systems (DEBS)*. ACM, 2013.
- [3] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregel: Big(ger) Graph Analytics on a Dataflow Engine," *Proc. VLDB Endowment*, vol. 8, no. 2, pp. 161–172, 2014.
- [4] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: graph processing at Facebook-scale," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.
- [5] S. Das, S. Nishimura, D. Agrawal, and A. E. Abbadi, "Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration," in *Proc. VLDB Endowment*, 2011.
- [6] D. Didona, P. Romano, S. Peluso, and F. Quaglia, "Transactional Auto Scaler: Elastic Scaling of In-memory Transactional Data Grids," in *Proc. Int'l Conf. Autonomic Computing (ICAC)*. ACM, 2012.
- [7] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, Robust Capacity Management for Multi-tier Data Centers," in *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4. ACM, 2012, p. 14.
- [8] M. Ghosh, W. Wang, G. Holla V., and I. Gupta, "Morphus: Supporting Online Reconfigurations in Sharded NoSQL Systems," in *Proc. Int'l Conf. Autonomic Computing (ICAC)*. IEEE, 2015.
- [9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed Graph-Parallel Computation on Natural Graphs," in *Proc. Symp. Operating Systems Design and Implementation (OSDI)*. USENIX, 2012.
- [10] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *Proc. Symp. Operating Systems Design and Implementation (OSDI)*. USENIX, 2014.
- [11] H. Herodotou, F. Dong, and S. Babu, "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics," in *Proc. Symp. Cloud Computing (SoCC)*. ACM, 2011.
- [12] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A Self-tuning System for Big Data Analytics," in *Proc. Conf. Innovative Data Systems Research (CIDR)*. ACM, 2011.
- [13] I. Hoque and I. Gupta, "LFGraph: Simple and Fast Distributed Graph Analytics," in *Proc. Conf. Timely Results In Operating Systems (TRIOS)*. ACM, 2013.
- [14] J. Jiang, G. Zhang, and G. Long, "Optimal Cloud Resource Auto-scaling for Web Applications," in *Proc. Int'l Symp. Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2013.
- [15] R. Jonker and T. Volgenant, "Improving the Hungarian Assignment Algorithm," *Operations Research Letters*, vol. 5, no. 4, pp. 171–175, 1986.
- [16] H. W. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.
- [17] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" in *Proc. Int'l Conf. World Wide Web (WWW)*. ACM, 2010.
- [18] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," in *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2. ACM, 2010, pp. 35–40.
- [19] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proc. VLDB Endowment*, 2012.
- [20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A New Parallel Framework for Machine Learning," in *Proc. Conf. Uncertainty in Artificial Intelligence (UAI)*, 2010.
- [21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proc. Int'l Conf. Management of Data (SIGMOD)*. ACM, 2010.
- [22] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service," in *Proc. Int'l Conf. Autonomic Computing (ICAC)*. USENIX, 2013.
- [23] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The Little Engine (s) That Could: Scaling Online Social Networks," in *Proc. SIGCOMM*. ACM, 2010.
- [24] Riak Distributed Database. <http://basho.com/riak/>.
- [25] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," in *Proc. Int'l Conf. Scientific and Statistical Database Management*. ACM, 2013.
- [26] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic Resource Scaling for Multi-tenant Cloud Systems," in *Proc. Symp. Cloud Computing (SoCC)*. ACM, 2011.
- [27] I. Stanton and G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs," in *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 2012.
- [28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proc. SIGCOMM*. ACM, 2001.
- [29] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, Elastic Resource Provisioning for NoSQL Clusters using TIRAMOLA," in *Proc. Int'l Symp. Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2013.
- [30] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "xDGP: A Dynamic Graph Processing System with Adaptive Partitioning," *arXiv:1309.1049*, 2013.
- [31] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein, "Dynamic Energy-aware Capacity Provisioning for Cloud Computing Environments," in *Proc. Int'l Conf. Autonomic Computing (ICAC)*. ACM, 2012.

APPENDIX A

CONTIGUOUS VERTEX REPARTITIONING (CVR)

*Proof. (Theorem 1)* For a given value of  $k$ , let  $t_{so}(i)$  be the number of vertices migrated from the server at old position  $i$  ( $1 \leq i \leq N$ ). The servers at positions to the left of the new server (i.e.,  $i < k$ ) will transfer some vertices to their immediate right neighbors, while the servers to the right of this position (i.e.,  $i \geq k$ ) transfer vertices to their left neighbors (see Figure 1 for an illustration). Initially, each server holds  $\frac{V}{N}$  vertices; after scale-out, each server would hold  $\frac{V}{N+1}$  vertices. Hence, each server contributes  $(\frac{V}{N} - \frac{V}{N+1})$  vertices to its neighbor, and these are accumulated across all old partitions to populate the new server's partition with  $(\frac{V}{N} - \frac{V}{N+1}) \times N = \frac{V}{N+1}$  vertices. Servers which receive vertices from a neighbor pass on as many vertices to their next neighbor, giving us:

$$\begin{aligned} t_{so}(i=1) &= \frac{V}{N} - \frac{V}{N+1} && \text{if } 0 < k \leq N \\ t_{so}(i=N) &= \frac{V}{N} - \frac{V}{N+1} && \text{if } 0 \leq k < N \\ t_{so}(i) &= t_{so}(i-1) + \frac{V}{N} - \frac{V}{N+1} && \text{for } 1 < i \leq k \\ t_{so}(i) &= t_{so}(i+1) + \frac{V}{N} - \frac{V}{N+1} && \text{for } k < i < N \end{aligned} \quad (1)$$

This can be solved to give:

$$\begin{aligned} t_{so}(i) &= i \times \left( \frac{V}{N \times (N+1)} \right) && \text{for } 1 \leq i \leq k \\ t_{so}(i) &= (N-i+1) \times \left( \frac{V}{N \times (N+1)} \right) && \text{for } k < i \leq N \end{aligned} \quad (2)$$

The total number of vertices migrated by all servers is:

$$\begin{aligned} C_{so}(k) &= \sum_{i=1}^N t_{so}(i) = \frac{V}{N \times (N+1)} \times \left( \sum_{i=1}^{i \leq k} i + \sum_{i=k+1}^{i \leq N} (N-i+1) \right) \\ &= \frac{V}{N \times (N+1)} \times \left( \frac{2 \times k^2 + N^2 + N - 2 \times k \times N}{2} \right) \end{aligned} \quad (3)$$

By solving for  $\frac{d}{dk} C_{so}(k) = 0$  (and checking that the second derivative is positive), we find that  $C_{so}(k)$  is minimized when  $k = k^* = \frac{N}{2}$ . The minimal number of vertices transferred is therefore  $C_{so}^* = \frac{V \times (N+2)}{4 \times (N+1)}$ .  $\square$

*Proof. (Theorem 2)* For a given value of  $k$ , let  $t_{si}(i)$  be the number of vertices migrated from the server at old position  $i$  ( $1 \leq i \leq N$ ). The leaving server transfers  $\frac{V(k-1)}{N(N-1)}$  vertices to its left neighbor while  $\frac{V(N-k)}{N(N-1)}$  vertices to its right neighbor. This is because it proportionally transfers its vertex set to the servers on its left and right side. The servers at positions to the left of the leaving server (i.e.,  $i < k$ ) will transfer some vertices to their immediate left neighbors, while the servers to the right of this position (i.e.,  $i > k$ ) transfer vertices to their right neighbors. Initially, each server holds  $\frac{V}{N}$  vertices. After scale-in, each server would hold  $\frac{V}{N-1}$  vertices. Thus, we have:

$$\begin{aligned} t_{si}(i=1) &= 0 && \text{if } 1 < k \leq N \\ t_{si}(i=N) &= 0 && \text{if } 1 \leq k < N \\ t_{si}(i) &= t_{si}(i+1) + \frac{V}{N} - \frac{V}{N-1} && \text{for } 1 < i < k \\ t_{si}(i) &= t_{si}(i-1) + \frac{V}{N} - \frac{V}{N-1} && \text{for } k < i < N \end{aligned} \quad (4)$$

This can be solved to give:

$$\begin{aligned} t_{si}(i) &= (k-i) \left( \frac{-V}{N \times (N-1)} \right) + \frac{V \times (k-1)}{N \times (N-1)} && \text{for } 1 \leq i \leq k \\ t_{si}(i) &= (i-k) \left( \frac{-V}{N \times (N-1)} \right) + \frac{V \times (N-k)}{N \times (N-1)} && \text{for } k \leq i \leq N \end{aligned} \quad (5)$$

The total number of vertices migrated by all servers is:

$$\begin{aligned} C_{si}(k) &= \sum_{i=1}^N t_{si}(i) = \frac{-V}{N \times (N-1)} \left( \sum_{i=1}^{i \leq k} (k-i) + \sum_{i=k}^{i \leq N} (i-k) \right) \\ &\quad + (k-1) \frac{V \times (k-1)}{N \times (N-1)} + (N-k) \frac{V \times (N-k)}{N \times (N-1)} + \frac{V}{N} \\ &= \frac{V}{N \times (N-1)} \left( \frac{N^2 + 2k^2 - 2Nk - 2k - N + 2}{2} \right) + \frac{V}{N} \end{aligned} \quad (6)$$

By solving for  $\frac{d}{dk} C_{si}(k) = 0$  (and checking the second derivative is positive) we find that  $C_{si}(k)$  is minimized when  $k = k^* = \frac{N+1}{2}$ . The minimal number of vertices transferred is  $C_{si}^* = \frac{V \times (N+1)}{4 \times N}$ .  $\square$

APPENDIX B

RING-BASED VERTEX REPARTITIONING (RVR)

*Proof. (Theorem 5)* The proof follows from our strategy of assigning the first  $N$  new servers each to a disjoint old partition, then, in the second round, iterating through the next  $N$  servers and assigning them again to the  $N$  old partitions, and so on for  $\lceil \frac{k}{N} \rceil$  rounds. Since new servers are evenly distributed inside an old partition, this achieves the minimax.  $\square$

APPENDIX C

SIMILARITIES

*Proof. (Theorem 7)* A scale-out technique achieves the minimum total network transfer volume (number of vertices migrated) if the extra vertices at existing servers due to scale-out are directly migrated to the new servers. Similarly, a scale-in technique achieves the minimum total network transfer volume if the vertices from leaving servers are directly migrated to the server that will finally be responsible for them. If we add  $m \times N$  servers or remove  $\frac{m \times N}{m+1}$  servers, we can achieve this by adding  $m$  servers to each of the  $N$  existing partitions or removing  $m$  servers from each of the  $\frac{N}{m+1}$  remnant partitions. That is, the optimal position of servers to be added or removed with RVR is same as their position with CVR.  $\square$