TopoX: Topology Refactorization for Efficient Graph Partitioning and Processing[*]

Dongsheng Li[†]
NUDT
Sanyi Rd.
Changsha, China
dsli@nudt.edu.cn

Yiming Zhang[†]
NUDT
Sanyi Rd.
Changsha, China
zhangyiming@nudt.edu.cn

Jinyan Wang
NUDT
Sanyi Rd.
Changsha, China
wjy0213@vip.qq.com

Kian-Lee Tan
NUS
13 Computing Drive
Singapore
tankl@comp.nus.edu.sg

ABSTRACT

Traditional graph partitioning methods attempt to both minimize communication cost and guarantee load balancing in computation. However, the skewed degree distribution of natural graphs makes it difficult to simultaneously achieve the two objectives. This paper proposes *topology refactorization* (TR), a topology-aware method allowing graph-parallel systems to separately handle the two objectives: refactorization is mainly focused on reducing communication cost, and partitioning is mainly targeted for balancing the load. TR transforms a skewed graph into a more communication-efficient topology through *fusion* and *fission*, where the fusion operation organizes a set of neighboring low-degree vertices into a super-vertex, and the fission operation splits a high-degree vertex into a set of sibling sub-vertices.

Based on TR, we design an efficient graph-parallel system (TopoX) which pipelines refactorization with partitioning to both reduce communication cost and balance computation load. Prototype evaluation shows that TopoX outperforms state-of-the-art PowerLyra by up to 78.5% (from 37.2%) on real-world graphs and is significantly faster than other graph-parallel systems, while only introducing small refactorization overhead and memory consumption.

[*]This work is supported by the National Key R&D Program of China (2018YFB0204300) and the National Natural Science Foundation of China (61772541 and 61872376). Yiming Zhang is the corresponding author. Kian-Lee Tan is partially supported by a MoE Tier 2 grant (MOE2017-T2-1-141) in Singapore.

[†]Co-first authors.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 12, No. 8
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3324301.3324306

PVLDB Reference Format:
Dongsheng Li, Yiming Zhang, Jinyan Wang, and Kian-Lee Tan. TopoX: Topology Refactorization for Efficient Graph Partitioning and Processing. *PVLDB*, 12(8): 891-905, 2019.
DOI: https://doi.org/10.14778/3324301.3324306

1. INTRODUCTION

Graph computation is central to machine-learning-data-mining (MLDM) applications such as social computing, recommendation, and language processing [16, 45, 62, 8, 42]. Graph processing on a single machine has been extensively studied [64, 52, 28, 41, 38, 44, 58, 44, 18] and has achieved remarkably high efficiency. However, the rapidly-growing graph size has exceeded the capacity of any individual machines. This has recently driven the study of graph-parallel systems [32, 51] including Pregel [34], Giraph [47], Blogel [55], GraphLab [31], PowerGraph [15], GraphX [17], GraphBuilder [21], PowerLyra [10], GraphA [29], HDRF [40], Cube [59] and Gemini [63].

While reducing per-machine computation load, parallel processing faces difficulty in graph *partitioning* [24, 6], which is crucial for both reducing the communication cost and balancing the load in graph-parallel systems. Further, real-world natural graphs usually have skewed distributions of vertex degrees where a small subset of vertices tend to have a large number of neighbors [17], making it even more challenging for efficient partitioning.

Early graph-parallel systems (like Pregel [34] and GraphLab [31]) partition a graph by cutting the edges to evenly distribute the vertices. These *edge-cut* systems suffer from imbalanced computation and communication for high-degree vertices. In contrast, *vertex-cut* systems (like PowerGraph [15] and GraphX [17]) cut the vertices and evenly distribute the edges among machines. Although vertex-cut alleviates the imbalance problem of high-degree vertices, it incurs high communication cost and excessive memory consumption for low-degree vertices. Recently, PowerLyra [10] proposes the *hybrid-cut* strategy to address this problem, which differentiates high-degree and low-degree vertices by (i) assigning the edges of a high-degree vertex to all machines to evenly distribute the computation load and (ii) assigning the edges of a low-degree vertex to the same machine to reduce unnecessary communication.

Essentially, the partitioning procedure divides a graph into n sub-graphs in an n-machine cluster, each being assigned to one worker machine. Traditional graph partitioning methods attempt to both minimize communication cost (i.e., minimize the number of edges spanning the n sub-graphs) and guarantee load balancing (i.e., evenly distribute the vertices/edges among the n sub-graphs) in computation. However, the skewed degree distribution of natural graphs makes it difficult to simultaneously achieve the two objectives: such ideal partitioning is NP-hard (incurring overwhelming coordination overhead in large-scale graph-parallel systems) and cannot be achieved for large graphs. Consequently, most (edge-cut, vertex-cut and hybrid-cut) partitioning methods cut *individual* vertices or edges (probably with per-machine locality heuristics) and thus cannot leverage the *topology* information of the graphs.

To address this problem, in this paper we propose *topology refactorization* (TR), a topology-aware method that allows graph-parallel systems to separately handle the two objectives: refactorization is mainly focused on reducing communication cost, and partitioning is mainly targeted for balancing the load. TR transforms a skewed graph into a more communication-efficient topology (with slight imbalance) before partitioning. At the core of TR is the *fusion* operation which merges a set of neighboring low-degree vertices into a super-vertex to reduce communication cost. TR also performs the *fission* operation which splits a high-degree vertex into a set of sibling sub-vertices (similar to hybrid-cut) to eliminate computation bottlenecks. The refactorized graph is partitioned by uniformly distributing the super- and sub-vertices (along with edges) to all workers.

Based on TR, we design an efficient graph-parallel system called TopoX, which pipelines refactorization with partitioning to both reduce communication cost and balance computation load. For abelian [3] algorithms, we propose the delta-based GAS (D-GAS) model by extending delta-caching [15] of PowerGraph, so as to further reduce the communication cost. Compared to delta-caching, D-GAS reduces the (maximum) communication cost per active mirror from 4 to 2 messages. We have implemented a prototype of TopoX on PowerGraph [15]. Extensive evaluation shows that TopoX outperforms state-of-the-art PowerLyra by up to 78.5% (from 37.2%) on real-world skewed graphs and is much faster than other systems like PowerGraph, GraphX and Blogel, while only introducing small refactorization and partitioning overhead and memory consumption.

This paper makes the following contributions:

- We present the TR method which transforms a skewed graph into a better topology, and design the corresponding partitioning method.
- We propose the delta-based GAS computation model for abelian algorithms, which is more efficient than delta-caching in graph-parallel systems.
- We implement a comprehensive and high-performance prototype (TopoX) integrating TR, pipelined partitioning, and standard/delta-based GAS.

The rest of this paper is organized as follows. Section 2 introduces the background of graph partitioning and computation. Section 3 presents an overview of topology refactorization. Section 4 discusses the design of parallel factorization, partitioning and computation. Section 5 introduces the PowerGraph-based implementation of TopoX as well as the D-GAS model. Section 6 presents the evaluation results. Section 7 discusses related work. And finally Section 8 concludes the paper.

2. BACKGROUND

For a graph $G = \{V, E\}$ where V is the vertex set and E is the edge set, graph computation is abstracted as a vertex-program $Q(u)$ that is executed on each $u \in V$ and interacts with $Q(v)$ where v is an in-/out-neighbor of u (denoted by $v \in \Gamma(u)$). This section will review current partitioning methods of edge-cut (Section 2.1), vertex-cut and hybrid-cut (Section 2.2), and analyze the processing in the standard GAS model [15] of graph-parallel systems (Section 2.3).

2.1 Edge-Cut

Edge-cut systems [31, 34, 46, 47, 48] evenly distribute vertices to multiple workers each of which maintains a consistent partial state of the graph. For example, Pregel [34] adopts a *bulk synchronous parallel* (BSP) message passing abstraction where all vertex programs run simultaneously in a sequence of super-steps, and GraphLab [31] adopts an asynchronous and distributed shared-memory abstraction where vertex programs have shared access to a distributed graph. Edge-cut systems perform well for *balanced* graphs with low-degree vertices [15].

However, real-world natural graphs (such as social networks and the web) usually have skewed power-law degree distributions, where most vertices have a small number of neighbors and a few vertices have many neighbors. For instance, 1% of the vertices in the Twitter network graph [27] are connected to almost 50% of all the edges. Specifically, under a power-law degree distribution the probability $P(d)$ that a vertex has degree d is given by $P(d) \propto d^{-\alpha}$, where the positive constant α controls the skewness of the distribution. Higher exponent α indicates lower density and less high-degree vertices. The computation load of a vertex is proportional to the vertex degree [15], so the skewness leads to significant computation load imbalance in edge-cut systems. The execution on workers having more high-degree vertices could be much slower than that on workers having more low-degree vertices.

2.2 Vertex-Cut & Hybrid-Cut

To address the challenge of high-degree vertices in natural graphs, Gonzalez et al. [15] abstract graph computation into the GAS model (§2.3), a general model which could express a wide range of abstractions including Pregel's BSP message passing [34] and GraphLab's asynchronous shared-memory [31]. Following GAS, PowerGraph [15] partitions a graph by cutting vertices instead of edges and evenly assigning the vertices to workers. However, since PowerGraph indiscriminately cuts all vertices, it incurs unnecessarily high communication cost and excessive memory consumption for low-degree vertices. Other vertex-cut systems (like GraphX [17] and GraphBuilder [21]) have similar problems for low-degree vertices.

To address this problem, PowerLyra [10] proposes *hybrid-cut*, which distinguishes the processing of low-degree and high-degree vertices. Hybrid-cut evenly distributes the edges of a high-degree vertex among workers (following vertex-cut) to distribute the computation load, and assigns all the edges

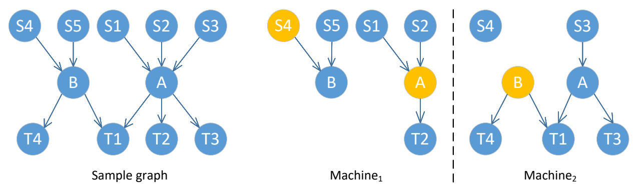


Figure 1: Hybrid-cut partitioning. Right: blue and yellow circles respectively represent masters and pmirrors.

of a low-degree vertex to the same worker (following edge-cut) to reduce communication.

In hybrid-cut, each vertex is assigned to a machine by hashing its ID. If an edge (e) is assigned together with one of its vertex (s) to machine M and e's other vertex (t) is assigned to machine $M' \neq M$, then vertex s will have a *master* replica on M, and vertex t will have a *master* replica on M' as well as a *mirror* replica on M. Without loss of generality, PowerLyra performs hybrid-cut by considering the in-degrees of vertices: if the *target* vertex of an edge (e) is low-in-degree then e will be assigned with its target; otherwise (if e's target is high-in-degree) e will be assigned with its source.

Consider the graph in Fig. 1 (*left*). Suppose that the degree threshold is 3, i.e., a vertex is considered to be high-degree if its in-degree ≥ 3. Consequently, in Fig. 1 (*left*) vertex A is high-degree and all others are low-degree. Suppose that vertices S_1, S_2, S_5, T_2 and B are hashed to Machine$_1$ and vertices S_3, S_4, T_1, T_3, T_4 and A to Machine$_2$. Then, the in-edges of vertex A, namely, $<S_1, A>$, $<S_2, A>$ and $<S_3, A>$ will be assigned with their source vertices (*source hashing*); and the in-edges of all other vertices, namely, $<A, T_1>$, $<A, T_2>$, $<A, T_3>$, $<S_4, B>$, $<S_5, B>$, $<B, T_1>$, $<B, T_4>$ will be assigned with their target vertices (*target hashing*). The result is shown in Fig. 1 (*right*), where the blue and yellow circles respectively represent the masters and mirrors.

2.3 Distributed GAS Model

The Gather-Apply-Scatter (GAS) model is widely adopted in distributed graph processing. It comprises three conceptual phases running on each vertex u. In the Gather phase, u collects its neighboring information via `Gather()` to get a *generalized sum* over the adjacent vertices and edges of vertex u.

The generalized sum operation must be commutative and associative, such as *numerical* sum, max, min, and *set* union of neighboring vertices/edges [15]. In the Apply phase, The sum is passed to `Apply()` to calculate the new value of u. In the Scatter phase, vertex u uses `Scatter()` to update the data on adjacent edges and activate further GAS phases of adjacent vertices in the next iteration.

In graph-parallel systems, the master/mirror replicas of vertices significantly complicate the GAS model. Consider the *high*-degree vertex A in Fig. 1 (*right*), which follows hybrid-cut to adopt source hashing for its in-edges and thus has a master on Machine$_2$ and a mirror on Machine$_1$. Data is gathered along in-edges and scattered along out-edges. The distributed GAS phases are as follows.

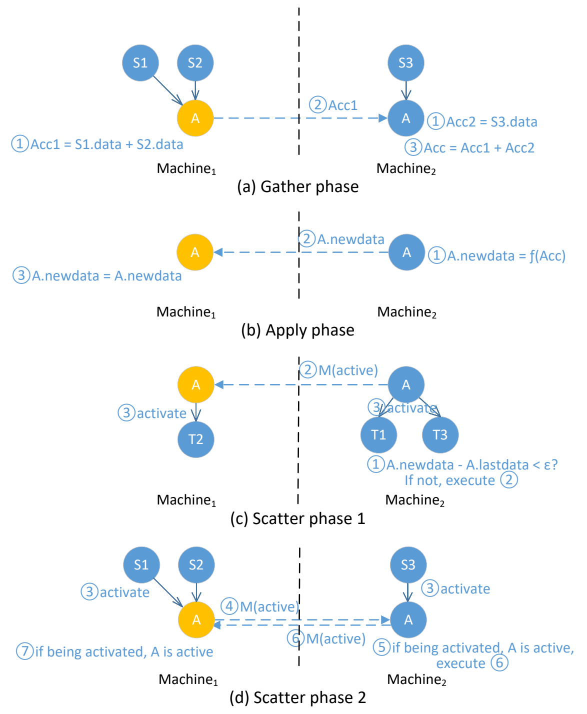


Figure 2: Example of the distributed GAS model. Vertex A is high-degree. Note that in the implementation of PowerGraph (v2.2), Fig. 2 (c) is called Scatter phase and Fig. 2 (d) is further decomposed into a Receive phase (R) and a Send phase (X).

Gather. As shown in Fig. 2 (a), in the Gather phase the data is collected along the in-edges of A. First, both A's master (on Machine$_2$) and mirror (on Machine$_1$) collect the partial sum (Acc$_2$ and Acc$_1$, respectively) from their in-neighbors. Second, the partial sum (Acc$_1$) of A's mirror is sent to A's master. And last, A's master computes the sum (Acc $=$ Acc$_1$ $+$ Acc$_2$).

Apply. As shown in Fig. 2 (b), in the Apply phase A's master (on Machine$_2$) applies f on the sum (Acc) to compute the new value, which is sent to A's mirror (on Machine$_1$). Then the mirror is updated with the new value.

Scatter. The Scatter phase is to figure out whether to activate the next iteration of GAS phases, and could be divided into two subphases. In Scatter phase 1 (Fig. 2 (c)), A's master (on Machine$_2$) first computes the delta between the new and old values. If the delta is greater than a predefined threshold (ϵ), the master will send an activation message to the mirror (on Machine$_1$), and both the master and the mirror will then activate their out-neighbors (T_1 and T_3 for the master, and T_2 for the mirror, respectively). The above processing is also performed on A's in-neighbors (S_1, S_2 and S_3), which may or may not activate A in their Scatter phase 1. As shown in Fig. 2 (d), if the mirror of A is activated then in Scatter phase 2 it will send an activation message to the master. And if the master is activated by its in-neighbors or receives activation messages from its mirrors, it will be active in the next iteration and activate all its mirrors, so that both the master and the mirrors will collect data in (the Gather phase of) the next iteration.

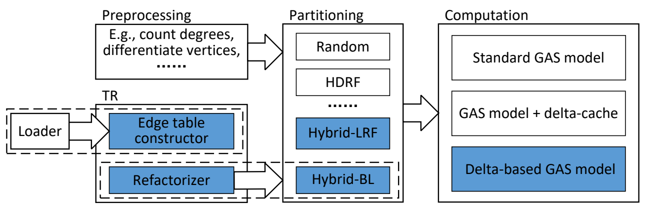


Figure 3: Overview of TopoX. Blue components are newly designed. Processing in dashed rectangles is pipelined.

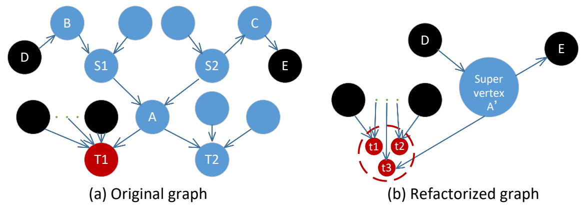


Figure 4: Refactorization. *Fusion* merges neighboring low-degree vertices to a super-vertex A'. *Fission* splits the high-degree vertex (T_1) into sub-vertices (t_1, t_2, t_3).

The GAS phases for low-degree vertices (which adopts target hashing) are simpler than that for high-degree vertices. Consider the low-degree vertex B in Fig. 1 (*right*). Since all the in-edges of B ($<S_4, B>$ and $<S_5, B>$) are assigned to Machine$_1$ by hashing B, only B's master (on Machine$_1$) has in-edges and B's mirror (on Machine$_2$) has no in-edges. Therefore, in Gather there is no communication for transferring partial sums from B's mirror to B's master; and in Scatter all activation messages are received only by B's master and B's mirror will never perform the collect operation in the next Gather phase, so the entire Scatter phase 2 (Fig. 2 (d)) could be omitted.

Although target hashing is simpler than source hashing, hybrid-cut adopts it only for low-degree vertices. This is because for target hashing (i) all the computation along the in-edges is performed on the target which causes imbalance; and (ii) the source vertex (e.g., S_4) of *each* in-edge ($<S_4, B>$ in Fig. 1 (*right*)) of the target (B) has a mirror (S_4 on Machine$_1$) if the target and source are hashed to different machines.

3. ARCHITECTURE OVERVIEW

As shown in Fig. 3, traditionally there are three processing stages after loading the graph, namely, (i) *preprocessing* such as counting degrees and differentiating vertices [10], (ii) *partitioning* where the vertices and edges are assigned to workers using various strategies such as Random, HDRF [40], Grid, and hybrid-cut, and (iii) *computation* where the graph engine runs graph applications following the GAS model or its variations. For *partitioning*, most strategies (including edge-cut, vertex-cut, and hybrid-cut) cut individual vertices/edges by hashing the vertices (probably with per-machine locality heuristics) to divide a graph, making them unable to leverage the topology information for reducing communication cost.

Topology refactorization (TR) could be viewed as a special preprocessing phase before partitioning. As shown in Fig. 3, TR consists of an *edge table constructor* and a *refactorizer*, which collaboratively perform *fusion* and *fission* on vertices/edges to obtain a more balanced topology. Fusion merges a set of neighboring low-degree vertices into a super-vertex, and fission splits a high-degree vertex into a set of sibling sub-vertices. The new topology is partitioned by evenly distributing the super- and sub-vertices, adopting the *hybrid-BL* partitioning method (an extension of hybrid-cut [10]) which could be *pipelined* with refactorization for both reducing communication cost and balancing the load. TR and hybrid-BL will be introduced in §4.

After partitioning, graph computation could be performed following the standard GAS model. As shown in Fig. 3, TopoX further introduces the *delta-based GAS model* (D-GAS) for abelian [3] algorithms, which transfers the delta (instead of the value) of vertices to reduce the communication cost in graph computation. The D-GAS model will be introduced in §5.

4. TOPOLOGY REFACTORIZATION

In GAS model, the *replication factor* (λ) is defined as the average number of replicas for all vertices, indicating the expected communication cost in graph computation. Traditional graph partitioning methods [15] simultaneously have two objectives: (i) minimize communication cost (i.e., reduce the replication factor), and (ii) guarantee load balancing (i.e., evenly distribute the edges among sub-graphs).

The optimal partitioning problem is NP-hard [15] and thus cannot be solved for large graphs. To alleviate this problem, in this paper we propose to separate the two objectives by introducing refactorization (including *fusion* and *fission*) before partitioning: refactorization is mainly focused on reducing the replication factor, and partitioning is mainly targeted for balancing the load.

4.1 Fusion & Fission

Fusion. TR performs fusion by (i) randomly choosing a (low-degree) center vertex, and (ii) attaching its h-hop ($h \leq \gamma$) low-degree neighbors where the *radius* γ is the maximum number of hops from any inner vertex to the center. During fusion, both in-neighbors and out-neighbors satisfying $h \leq \gamma$ are added into the super-vertex no matter along which direction the data is gathered/scattered, so as to eliminate communication within a super-vertex. Consider the original topology depicted in Fig. 4(a). Taking a low-degree vertex A as the center and assuming radius $\gamma = 2$, its fusion procedure is as follows.

First, TR constructs an empty super-vertex A' and add A into A'. Second, TR adds all the one-hop low-degree neighbors (S_1, S_2, T_2) of A, together with the in-edges and out-edges to the super-vertex A'. High-degree neighbors (T_1) are skipped and will be processed with *fission* (discussed later). Third, TR adds all the one-hop low-degree neighbors of S_1, S_2, T_2 (i.e., two-hop neighbors of A) with edges to A'. The resulting (logical) topology is depicted in Fig. 4(b), where the super-vertex A' consists of A and its one-hop and two-hop low-degree neighbors. After fusion the edges inside A' (e.g., $<S_1, A>$) become invisible in the new topology. The super-vertices' edges will be assigned with their target vertices during partitioning. For instance, $<D, A'>$ and $<A', E>$ are assigned together with A' and E, respectively. Note that different from Blogel [55], TR is transparent to programmers and allows the more flexible vertex-level communication without super-vertex-level message aggregation before communication in each phase.

Fission. TR performs fission on a high-degree vertex by splitting it into a set of sub-vertices each having a smaller degree. Intuitively, the degrees of sub-vertices should be similar to that of super-vertices so as to achieve load balancing by evenly distributing all super- and sub-vertices. However, the super-vertex degrees will not be known until all fusion operations complete, causing time-consuming fission-after-fusion dependencies. Inspired by the splitting process on high-degree vertices in previous studies [15, 10, 30, 39], we perform fission without waiting for fusion by (i) splitting a high-degree vertex (v_h) into n sub-vertices (where n is the number of workers) each being assigned to one worker, and (ii) assigning v_h's edges ($e = <u, v_h>$ or $<v_h, u>$) according to (the hash of) the opposite ends (vertices u).

Without loss of generality, suppose that the vertex (e.g., T_1 in Fig. 4(a)) is high-in-degree. Then the edges to the high-in-degree vertex T_1 (split into t_1, t_2, t_3 in Fig. 4(b)) will be assigned together with their source vertices during partitioning. Fission evenly distributes the computation load of *each* high-degree vertex to all workers, and thus introduces no imbalance in partitioning *all* high-degree vertices.

Refactorization vs. locality-constraint. Existing partitioning strategies usually adopt locality constrained policies, such as Oblivious/Coordinated [15], Grid [21], Ginger [10], and HDRF [40], to restrict local edge placement. They configure a constraint set $S(v)$ for each vertex v and place an edge $<u, v>$ in one of the workers $\in S(u) \cap S(v)$. In these strategies, workers independently follow the constraints when partitioning/placing the edges in parallel. The locality-constraint policies attempt to *simultaneously* achieve the two objectives of (i) minimizing the replication factor (λ) and (ii) balancing the load. Consequently, there is no guarantee for a set of neighboring vertices/edges to be placed on the same worker. For instance, PowerLyra Ginger [10] assigns each individual vertex to a worker by calculating the (temporarily) minimum λ (which might be inconsistent with the final λ) and thus cannot ensure the blue vertices (in Fig. 4(a)) to be placed together. In contrast, TR explicitly transforms the original graph, allowing graph-parallel systems to *separately* handle the two objectives (as discussed in §4.2).

Refactorization vs. coarsening. Some Pregel-like systems [55, 47, 14] coarsen a graph by grouping neighboring vertices and merging duplicated edges between groups. They follow Pregel's BSP model and expose group-aware API to users for aggregating inter-group communication in each superstep. or instance, Blogel [55] (a variation of Pregel [34]) divides a graph into *blocks*. A block has at most one edge from/to another block, and a vertex v belongs to a block only if v is closer to the block's center than to any other blocks' centers. Blogel performs better than Pregel since coarsening eliminates intra-block communication. The Block communication model is straightforward for traversal and aggregation applications such as CC and SSSP [40], but could significantly complicate a broad range of other graph applications. To implement PageRank over a coarsened graph, for example, in each superstep Blogel (i) computes the local PageRank of each $v \in V(B)$, denoted by $lpr(v)$, (ii) for each block B, constructs $\Gamma(B)$ from $\Gamma(v)$ of all $v \in V(B)$ following a specific algorithm [23], to assign a weight to each out-edge, (iii) computes the *BlockRank* of each block, denoted by $br(B)$, (iv) distributes the BlockRank to out-neighbors proportionally to the edge weights, and (v) computes the PageRank of each vertex by $pr(v) = lpr(v) \times br(block(v))$. Moreover, it is nontrivial to adapt the BSP-based systems to the GAS model, which prevents them from applying vertex-cut and thus causes poor performance when computing at high-degree vertices [15]. In contrast, TR does not coarsen the graph, and allows communication between vertices rather than super-vertices.

4.2 Parallel Refactorization

By introducing refactorization, we could divide the traditional graph partitioning procedure into two stages: (i) refactorizing the original low- and high-degree vertices to get super- and sub-vertices, and (ii) assigning the super- and sub-vertices to the n workers. Since the fission operation for a high-degree vertex simply generates n replicas each evenly having $\frac{1}{n}$ share of the edges (similar to hybrid-cut [10]) as discussed in §4.1, the main challenge lies in the fusion operation for low-degree vertices.

Our basic idea for fusion is to focus on the objective of minimizing λ. Slight imbalance between super-vertices is acceptable because the second assignment stage can handle it. Therefore, instead of generating strictly equal-sized super-vertices, in the fusion stage we limit the super-vertices' *radius* to obtain roughly equal sizes. We then resort to the second assignment/partitioning stage to satisfy the requirement of load balancing.

Parallel refactorization algorithm. As shown in Algorithm 1, each worker first reads a subgraph G' of the graph, and builds the local in-edge/out-edge tables (IET/OET) for indexing the edges (Line 3). For instance, in the IET table the corresponding entry for vertex A (in Fig. 1) is "target: A; source: S_1, S_2, S_3", and in the OET table the corresponding entry is "source: A; target: T_1, T_2, T_3". Each worker initializes an empty fusion-queue (Q) to allow fusion requests from others.

There is a main loop (Line 5) for distributed refactorization. In the loop, if Q is empty then a vertex v is taken out (Line 7). If v is low-degree and has no super-vertex, then it initiates a super-vertex taking itself as the center (Line 9). Then v performs fusion (Line 25) where for each neighbor (w) it adds w and the in-/out-edge to the super-vertex (Lines 31, 32), and notifies the worker (where the subgraph of w resides) to move w to Q (Line 33).

If vertex v is high-degree, then the worker performs fission (Line 11), which (i) generates n sub-vertices $\psi_v^{(i)}$ ($i = 1, 2, \cdots, n$) of v, (ii) assigns each of the n machines $M^{(i)}$ with a sub-vertex $\psi_v^{(i)}$, and (iii) splits v's edges into n subsets by hashing v's opposite end (u). The i^{th} edge subset is attached to $\psi_v^{(i)}$.

If Q is not empty (Line 13) then a vertex u (which has been previously added to Q by another worker in Line 33) is taken out of Q. If u has not been processed by the worker and the radius of u's super-vertex Υ_u is lower than the threshold γ, then the worker further performs fusion on u (Line 16). Otherwise the current pass completes.

Algorithm 1 Parallel Refactorization

```
 1: procedure REFACTORIZATION(Subgraph $G'$)
 2:     $V' \leftarrow$ local vertex subset
 3:     build local in-/out-edge tables (IET/OET) for $G'$
 4:     initialize an empty fusion-queue $Q$
 5:     while $V'$ is NOT empty do
 6:         if $Q$ is empty then
 7:             take a vertex $v$ out of $V'$
 8:             if $v$ is low-degree and $v.super$ is NULL then
 9:                 INITFUSION($v$)
10:             else if $v$ is high-degree then
11:                 FISSION($v$)
12:             end if
13:         else
14:             take a vertex $u$ out of $Q$
15:             if $u$ has NOT been completely processed
                before and $u.super.radius <$ Threshold $\gamma$ then
16:                 FUSION($u$)
17:             end if
18:         end if
19:     end while
20: end procedure

21: procedure INITFUSION(Vertex $u$)
22:     construct an empty super-vertex $\Upsilon_u$ for $u$
23:     $u.super \leftarrow \Upsilon_u$
24:     if $u.super.radius <$ Threshold $\gamma$ then
25:         FUSION($u$)
26:     end if
27: end procedure

28: procedure FUSION(Vertex $u$)
29:     for each neighbor $w$ of $u$ do
30:         if $w.super$ is NULL then
31:             $w.super \leftarrow u.super$
32:             add $e$ between $u$ and $w$ to $u.super$'s edge set
33:             if $w \in V'$ then move $w$ from $V'$ to $Q$
34:         end if
35:     end for
36: end procedure

37: procedure FISSION(Vertex $v$)
38:     for each machine $M^{(i)}$ do
39:         generate a sub-vertex $\psi_v^{(i)}$ of $v$
40:     end for
41:     for each neighbor $u$ of $v$ do
42:         if $u$ is hashed onto machine $M^{(i)}$ then
43:             add edge $e$ between $u$ and $v$ to sub-vertex
                $\psi_v^{(i)}$'s edge set
44:         end if
45:     end for
46: end procedure
```

Center vertex vs. affiliated vertex. We refer to the center of a super-vertex as *center* and others as *affiliated vertices*. The center proactively initiates its super-vertex (e.g., to find the host machine) while the affiliated vertices passively accept the arrangement. The affiliated vertices from Q (Line 14) are processed with higher priority than the center from the local vertex subset V' (Line 7), to add as much as possible affiliated vertices into one super-vertex instead of initiating more super-vertices. The affiliated vertices could be further divided into inner vertices (whose radius $< \gamma$) and border vertices (whose radius $= \gamma$), where inner vertices recursively invoke FUSION(), and border vertices stop further fusion operations. A border vertex of one super-vertex might be a border or inner vertex of another super-vertex. During refactorization, we use locks to avoid possible conflicts of multi-threaded data access on a single machine, e.g., when moving a vertex out of the local vertex set V' (Lines 7 and 33).

Threshold for super-vertex size. The threshold of super-vertices is for controlling their sizes for load balancing. Originally, we had used the number of edges in a super-vertex as the threshold, which directly reflects the computation load. Different from the radius threshold which could be predefined and well-known by all machines, however, the edge number threshold required the workers to dynamically coordinate the overall numbers of assigned edges, which induced overwhelming synchronization overhead in large-scale graph computation.

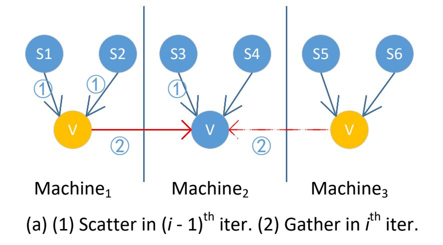
Therefore, we use the radius (Lines 15 and 24) as the threshold for the size of a super-vertex. Although super-vertices with the same radius may have different number of edges, if the radius is not too large the difference will be small enough (note that fusion only involves low-degree vertices) to be easily balanced in the second assignment stage. The radius threshold controls the tradeoff between computation balance and communication overhead. Higher radius results in larger super-vertices and consequently less inter-machine communication. However, it will also make some worker machines passively process more affiliated vertices (Line 16) while others proactively process more center vertices (Line 9). This exacerbates the imbalance due to passive assignments, which will be evaluated in more details in §6.6.

4.3 Partitioning after Refactorization

A graph could be partitioned by evenly assigning its vertices/edges to all workers. For a super-vertex, all its vertices and edges are assigned with its center, and its cross-border edges are assigned with their targets. For a sub-vertex, there are no inner edges and all the edges are assigned with the opposite ends of the high-degree sub-vertex.

We follow the recent analysis [51] of existing partitioning strategies and extend the state-of-the-art hybrid-cut strategy [10] to design *hybrid-BL*, a hybrid and refactorization-aware partitioning method for balancing the computation load. Hybrid-BL avoids global coordination by letting each worker assign its super- and sub-vertices according to its local knowledge about the load. It pipelines refactorization and partitioning to reduce the ingress time. Specifically, during the initiation of an empty super-vertex S on the current worker M (Line 22 in Algorithm 1), S is assigned to the least-loaded worker M' to which M has assigned the least edges; and in the fusion of S, each of the edges of S is assigned to M' (Line 32).

Originally, we had designed a replication-factor-optimized partitioning method called hybrid-LRF (least replication factor). Instead of pipelining, Hybrid-LRF performs partitioning after refactorization to calculate the best placement for each super-vertex. After refactorization is complete, consider the placement of the $(i+1)^{\text{th}}$ edge of a worker M given the i edges that have been assigned by M. Each super-vertex is assigned to its *optimal* worker M' that causes



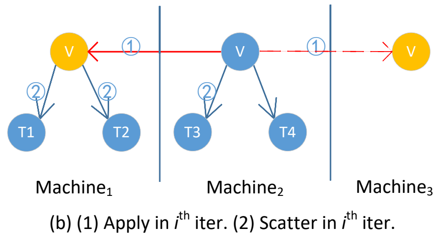


Figure 5: Phases in the D-GAS model of TopoX, assuming v is high-in-degree and its in-edges adopt source hashing. Compared to D-Cache, D-GAS avoids communication in Scatter phase.

the smallest increase of the replication factor for M. As shown in §6.6, although slightly reducing the replication factors, hybrid-LRF not only guarantees no balancing but also introduces much higher partitioning overhead compared to hybrid-BL, and thus is not recommended.

5. TOPOX GRAPH ENGINE

5.1 Standard GAS Model

TopoX follows the standard GAS model (Section 2.3) for computation on the refactorized graph, which provides a general abstraction supporting gather/scatter in both directions of edges. Without loss of generality, consider an algorithm (e.g., PageRank in Fig. 2) which performs both gather and scatter along the directions of edges. In the Gather phase, each vertex v has its master and mirrors (if exist) compute the partial sum from their in-neighbors and accumulate the results to the master. In the Apply phase, the master of v applies f on the accumulated sum and updates its mirrors. And in the Scatter phase, the master and the mirrors activate their out-neighbors if the deltas between the current and previous values are greater than the threshold. All the low-degree vertices in a super-vertex are assigned to the same worker, so TopoX reduces the inter-machine communication; and each high-degree vertex is divided and assigned to all workers, so TopoX balances the load.

5.2 Delta-Based GAS (D-GAS) Model

We propose the delta-based GAS (D-GAS) model to further improve the communication efficiency of TopoX for abelian algorithms [15] (like PageRank). D-GAS borrows the idea of *delta transfer* from delta-caching [15] and shared-memory systems [63], which allows the master/mirror vertices to transfer the deltas (instead of the partial sum and activation messages) in the Gather and Scatter phases.

Fig. 5 shows an example of the different phases in our D-GAS model. In the $(i-1)^{\text{th}}$ iteration, suppose S_1, S_2 and S_3 are active. In the Scatter phase (in Fig. 5a (1)), they will pass their deltas to their out-neighbors, i.e., both the (yellow) mirrors and the (blue) master of V. The $(i-1)^{\text{th}}$ iteration completes when all vertices finish their Scatter phases. Then, in the Gather phase of the i^{th} iteration (in Fig. 5a (2)), each mirror sums the deltas and sends them to the (blue) master V. The Gather phase in this iteration completes after the mirrors send sums to their masters (shown by the red line in Fig. 5a). Note that there is no data transfer from Machine$_3$ to Machine$_2$ in Fig. 5a (2) (represented using the dashed line), because S_5 and S_6 are inactive in the $(i-1)^{\text{th}}$ iteration.

In the Apply phase of the i^{th} iteration (in Fig. 5b (1)), V's master receives the deltas from its mirrors, and accumulates the deltas and applies f to compute a new value. Then the master will compute the delta between the new/old values, and if the delta is greater than a predefined threshold it will send the delta to its mirrors (shown by the red line in Fig. 5b). Then, in the Scatter of the i^{th} iteration (in Fig. 5b (2)), each mirror vertex passes the delta of V to its out-neighbors ($T_1 \sim T_4$), the processing of which is the same as that (from S_i to V) in the $(i-1)^{\text{th}}$ iteration's Scatter phase.

Table 1: Communication cost of GAS, D-Cache and D-GAS. H: high-degree vertices. I: (low-degree) inner vertices. B: (low-degree) border vertices. m: # mirrors.

	Model	Comm. Cost
PowerGraph	GAS	$\leq 5m$
PowerGraph	D-Cache	$\leq 4m$
TopoX	GAS	I: 0; B: $\leq m$; H: $\leq 4m$
TopoX	D-GAS	I: 0; B: $\leq m$; H: $\leq 2m$

5.3 D-GAS vs. Delta-caching

PowerGraph provides the delta-caching (D-Cache) mechanism if the accumulator type forms an *abelian* group [3], i.e., supports commutative and associative sum (+) and inverse (−). For example, the *numerical sum* operation in PageRank is abelian and thus D-Cache could be applied to PageRank. In contrast, the *set union* operation in Graph Coloring (GC) [4] supports only commutative and associative sum but no inverse, and thus *set union* is not abelian and cannot apply D-Cache.

The main goal of D-Cache is for reducing *computation* (rather than communication) cost. Compared to the standard GAS, D-Cache (i) has the mirrors transfer the deltas to the master in the Scatter phase instead of the activation message in the 4th step in Fig. 2 (d), and (ii) simplifies the Gather phase by adding the deltas to the cached accumulators. Suppose that a vertex V has d neighbors of which d_a neighbors are active. In the Gather phase D-Cache only needs to sum $d_a + 1$ values (d_a deltas and the base value of V), while the standard GAS needs to sum d values from all the d neighbors of V even when many of them are inactive. D-Cache reduces one message per mirror in each iteration, eliminating the message of accumulated value from the mirror to the master (in the 2nd step in Fig. 2 (a)).

The D-GAS model extends D-Cache's *delta computation* and proposes to use *delta messages* to reduce not only the computation cost but also the communication cost. As depicted in Fig. 5, D-GAS avoids communication in the Scatter

Table 2: Real-world and synthetic datasets.

Datasets	#edges	#vertices	Type
UK-2007 [2]	3.74B	105.9M	Power-law
Twitter [27]	1.46B	41.6M	Heavy-tailed
UK-2002 [1]	298.1M	18.5M	Power-law
α2.0	102.8M	10M	Power-law
α2.1	57.1M	10M	Power-law
α2.2	35.0M	10M	Power-law

phase by combining the update/activation messages. Table 1 compares the communication cost of the standard GAS, D-Cache, and D-GAS models in one iteration.

First, the standard GAS model in PowerGraph transfers at most five messages for each mirror (Fig. 2 (a)$\sim$(d)). D-Cache combines Gather in the i^{th} iteration and Scatter in the $(i-1)^{\text{th}}$ iteration by transferring delta in the 4^{th} step in Fig. 2 (d), and thus transfers at most four messages for each mirror.

Second, when adopting standard GAS, TopoX splits a high-degree vertex into sub-vertices and transfers at most 4 messages per mirror. Compared to PowerGraph's standard GAS model, the one message reduction is because TopoX combines the two messages of the Apply phase (Fig. 2 (b)) and of the Scatter phase 1 (Fig. 2 (c)). For low-degree vertices in TopoX, the border vertices of a super-vertex transfer at most one message for each of the mirrors owing to target hashing (§2.3). The cost of inner vertices is always zero.

Third, when adopting D-GAS, TopoX transfers at most two delta messages for a mirror of a high-degree vertex, which are 2 messages fewer than D-Cache: the 1^{st} is because D-GAS eliminates the master-to-mirror activation (in the 6^{th} step in Fig. 2 (d)) since the mirror decides whether to start the next round of Gather based on the delta; and the 2^{nd} is because D-GAS combines the two master-to-mirror messages in Apply (Fig. 2 (b)) and in Scatter phase 1 (Fig. 2 (c)).

TopoX adopts D-GAS as its default computation model, but will resort to the standard GAS model if the abelian condition cannot be satisfied.

6. EVALUATION

6.1 Overview

Datasets & testbeds. We evaluate TopoX on skewed graphs shown in Table 2 (UK-2007, Twitter and UK-2002). Besides the real-world graphs, we also use PowerGraph tools to generate three synthetic power-law graphs ($\alpha = 2.0$, 2.1 and 2.2, smaller α producing denser graphs) with 10 million vertices. We randomly add edges by sampling the in-degrees from a Zipf distribution [60] and keep the out-degrees of the vertices nearly identical.

We build two testbeds. First, we build an EC2 cluster consisting of 48 instances to evaluate the two large datasets (UK-2007 and Twitter). Each instance has 8 Xeon E5-2676 v3 vCPUs, 16GB memory and 250GB SSDs. Second, we build a local testbed with 9 machines, each having a 6-core E5-2640 CPU, 32GB memory and a 1Gbps NIC. The local testbed is used to evaluate the relatively small datasets (UK-2002, α2.0, α2.1, and α2.2). Each evaluation result is an

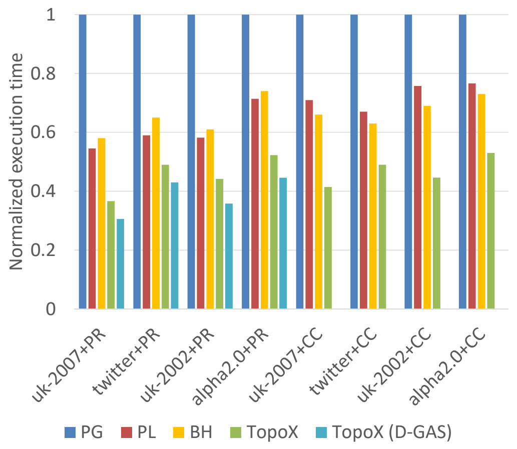


Figure 6: Normalized execution time.

average of 10 runs and the differences to the mean are less than 5%, which are omitted for clarity.

Graph-parallel systems. We compare TopoX to PowerGraph [15], PowerLyra [10], Blogel [55] and GraphX [17], running algorithms including PageRank (PR) [7] and Connected Components (CC) [11].

(1) TopoX adopts hybrid-BL (§4.3) for partitioning, and implements D-GAS (§5.2), D-Cache [15], and the standard GAS computation models. Although TopoX supports both synchronous and asynchronous execution directly inherited from PowerGraph, we focus on the default synchronous mode of PowerGraph (which is also the sole execution mode for most existing graph-parallel systems). The radius threshold (at Lines 15 and 24 in Algorithm 1 for performing fusion) is $\gamma = 3$ for UK-2007 and Twitter and $\gamma = 2$ for others. As shown in §6.6, higher γ might cause imbalance problems. The degree threshold is the same as PowerLyra (discussed below).

(2) PowerGraph supports several partitioning strategies including Grid, Random, Oblivious, Coordinated and HDRF [40]. For all tested graphs in Table 2, Oblivious and HDRF always have similar performance and outperform others, as revealed in [51]. Therefore, (if not specified) the results of PowerGraph are presented only for its best Oblivious strategy. PowerGraph adopts D-Cache for abelian applications and the standard GAS for others.

(3) PowerLyra is a variation of PowerGraph and adopts hybrid-cut with/without Ginger, which places the next low-degree vertex on the worker that minimizes the expected replication factor. Our evaluation shows that Ginger improves little performance while causing much higher memory and communication cost, which has also been suggested in [51]. Therefore, (if not specified) the results of PowerLyra are presented for its default hybrid-cut strategy without Ginger. PowerLyra adopts D-Cache whenever possible. The default degree threshold in PowerLyra is 100 [10].

(4) The original Blogel is a variation of Pregel adopting edge-cut which performs poorly for high-degree vertices. Considering most graphs in Table 2 have high-degree vertices, for fairness we have implemented *Blogel-hybrid* by porting the block abstraction (a.k.a. Voronoi cells) to PowerLyra. Following [55], we construct blocks *only* for low-degree vertices by performing multi-round, multi-source breadth-

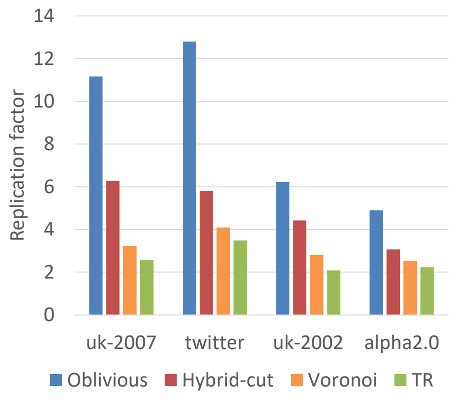


Figure 7: Replication factors.

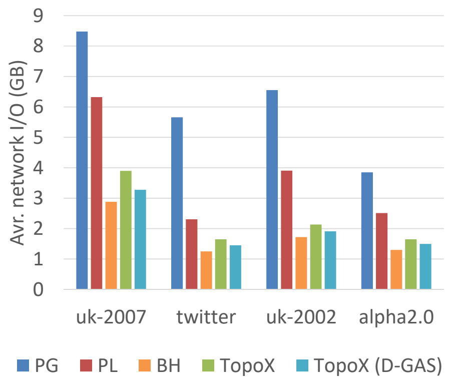


Figure 8: Avg. network I/O.

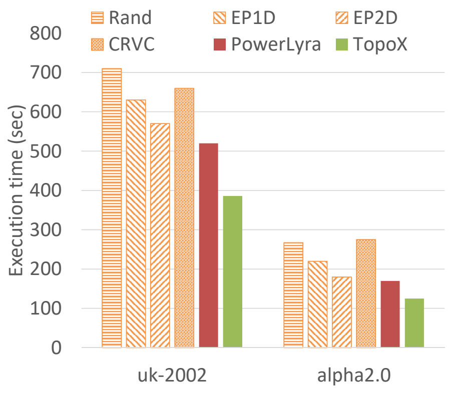


Figure 9: Execution time on Spark.

first search (BFS). A master is used to compute the block-to-worker assignment. Since communication between blocks is the same as that between vertices in PowerLyra, if the degree of a block (i.e., number of block neighbors) is larger than PowerLyra's degree threshold (100), it will be considered high-degree and assigned following hybrid-cut. Blogel-hybrid sets its initial sampling probability (for a vertex to be a block center) $p_{samp} = 0.1\%$, and increases p_{samp} by a factor of $f = 2$ after each round of BFS. The maximum value of p_{samp} (p_{max}) is 10%.

(5) GraphX is based on Spark [57] and supports both edge-cut and vertex-cut partitioning strategies, including Edge-Partition-1D (EP1D), Edge-Partition-2D (EP2D), Random, and Canonical-Random-Vertex-Cut (CRVC).

6.2 Computation Performance

Fig. 6 shows the normalized execution time of PR on UK-2007, Twitter, UK-2002 and α2.0 graphs, in PowerGraph (PG), PowerLyra (PL), Blogel-hybrid (BH), and TopoX (respectively with standard GAS and D-GAS).

PR keeps running until convergence in our evaluation. The baseline is the execution time of PowerGraph with Oblivious (376.6, 140.3, 152.7 and 40.1 seconds for the four graphs, respectively). TopoX with D-GAS achieves the best performance for all graphs, outperforming PowerLyra by up to 78.5% (on UK-2007). This is because refactorization leads to much lower replication factor than other strategies (Fig. 7), and D-GAS further reduces the communication cost (bringing an extra reduction of execution time from 12.2% to 18.9%). Blogel-hybrid performs even worse than PL because its block-level communication does not support PR well.

Fig. 6 also shows the results for CC (which does not support D-GAS). The baseline is the execution time of PowerGraph with Oblivious (170.1, 72.5, 62.7 and 16.4 seconds for the four graphs, respectively). TopoX (with standard GAS) is again the best for all graphs and outperforms PowerLyra by up to 71.3% (on UK-2007). Blogel-hybrid is slightly better than PowerLyra because its block-level communication adapts well to CC. However, the interference between construction and separation of high-degree blocks makes Blogel-hybrid perform worse than TopoX.

Replication factor & network I/O. We measure the replication factors of PG's Oblivious, PL's hybrid-cut, BH's Voronoi Cell and TopoX's TR. The results are shown in Fig. 7, where TopoX always outperforms others because TR effectively exploits topology structures. The interference between block construction and separation makes Voronoi Cell have higher replication factors than TR.

We run PR and evaluate the average network I/O. The results are shown in Fig. 8. TopoX outperforms others except Blogel-hybrid. But Blogel-hybrid suffers from not only much longer execution time (Fig. 6) but also several times longer ingress time (Fig. 10), because it adapts poorly to the skewness of the graphs. We will evaluate more partitioning metrics (ingress time and memory usage) in §6.3.

Performance on Spark. We port TopoX to Spark [57] to illustrate its generality. Fig. 9 compares the execution time of PR in TopoX (with standard GAS) on Spark, GraphX and PowerLyra (which also has a Spark-based implementation). The result shows that TopoX has lower execution time than all partitioning strategies of GraphX and Powerlyra. Note that the performance on Spark is lower than that on the PowerGraph platform, mainly because Spark's RDD structure complicates the processing of the GAS phases. Experiments on Spark for other graphs have similar results.

6.3 Refactorization & Partitioning Cost

We evaluate the ingress time of TopoX and compare it with that of PowerLyra, PowerGraph and Blogel-hybrid, respectively on UK-2007, Twitter, UK-2002 and α2.0 graphs. Fig. 10 shows the normalized ingress time. The baseline is the ingress time of PowerGraph with Oblivious (460.3, 258.8, 95.4 and 51.9 seconds for the four graphs).

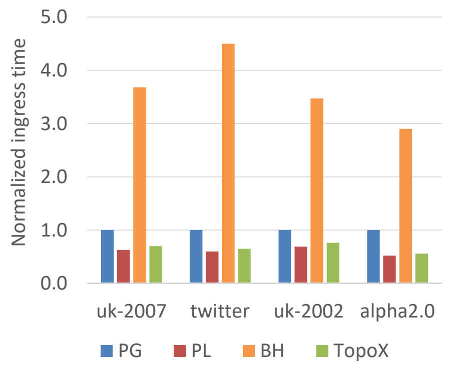


Figure 10: Normalized ingress time.

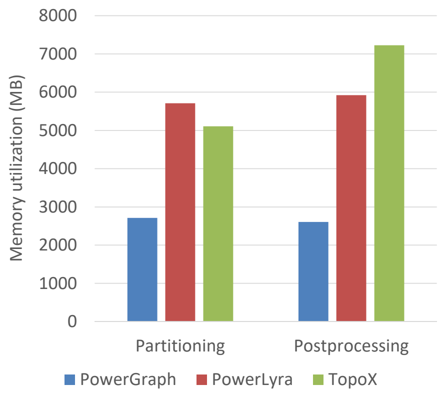


Figure 11: Avg. memory utilization.

First, the ingress time of TopoX is close to that of PowerLyra, because hybrid-BL pipelines not only graph loading and IET/OET construction (Line 3 in Algorithm 1) but also refactorization and partitioning. The small extra overhead is because the workers participate both in IET/OET construction (which is slightly more complicated than the preprocessing of others) and in its unique refactorization. Note that the partitioning results are *reusable* and thus it is worth paying for the extra refactorization/partitioning overhead in most cases.

Second, Blogel-hybrid has much higher ingress time than others, because the construction of Voronoi cells requires to sequentially perform the vertex-to-block and block-to-worker assignments, each causing a movement (and dump) of the entire graph [55]. In contrast, TopoX pipelines refactorization and partitioning and thus introduces little cost. Further, the Blogel master has to (i) record the changes of each block, (ii) dynamically compute the block-to-worker assignments, and (iii) broadcast the assignments to all workers, making it a severe bottleneck for partitioning.

Memory usage. We also evaluate the average memory consumption of TopoX, PowerLyra and PowerGraph when partitioning UK-2002. As shown in Fig. 3, TopoX's refactorization could be viewed as a special preprocessing phase, and thus we measure the memory consumption respectively for preprocessing and partitioning.

The result is shown in Fig. 11. *Partitioning* includes graph loading, preprocessing (like counting degrees in PowerLyra and refactorization in TopoX), and vertices/edges placement; and *postprocessing* includes finalizing the mirrors in the partitioned subgraphs. Compared to PowerLyra, TopoX requires less memory in partitioning owing to its smaller replication factors, but consumes slightly more in postprocessing because it cannot release the hash tables for locating neighbors when finalizing the mirrors.

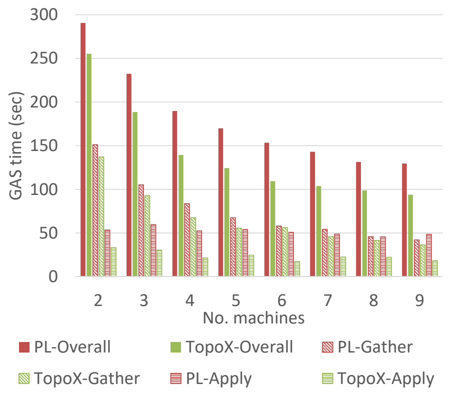


Figure 12: Comparison of overall, Gather, Apply times between TopoX and PowerLyra (PL).

6.4 Why TopoX Outperforms PowerLyra

To understand the advantage of TopoX over PowerLyra, we evaluate their execution time of the Gather, Apply and Scatter phases running PR on UK-2002 in TopoX and PowerLyra, for different numbers of workers. TopoX uses the standard GAS (instead of D-GAS) model to highlight the effect of refactorization. We evaluate not only the overall execution times but also the time of each GAS phase.

Fig. 12 shows the accumulated execution time, Gather time and Apply time, using a *single* core on each machine to simplify the accumulation. The Scatter time is omitted since it is negligible compared to the Gather and Apply times.

First, as the number of workers increases, both the overall execution times and the Gather times of TopoX and PowerLyra notably decrease, while the Apply times keep relatively stable. This is because the Gather phase has much more computation than the Apply phase, which could be naturally distributed among the machines. Second, the Apply times of TopoX are almost always more than 50% lower than that of PowerLyra, while the Gather times of TopoX are only slightly lower for all cluster sizes, indicating the Apply times contribute the most to the advantage of TopoX over PowerLyra in Fig. 6. Considering the fact that the Gather and Apply phases have almost the same amount of communication (as shown in Fig. 2 (a) and (b)), this difference is mainly because Gather has more computation than Apply making communication be able to be overlapped with computation, which weakens the benefits of communication reduction in TopoX. Clearly, TopoX would benefit more from refactorization for larger clusters where the ratio of Gather time to Apply time decreases.

6.5 D-GAS Vs. Delta-Caching

We evaluate the execution times of TopoX on UK-2002, as a function of the number of iterations (each containing three phases of Gather, Apply, and Scatter), respectively following the delta-caching (D-Cache) and D-GAS models.

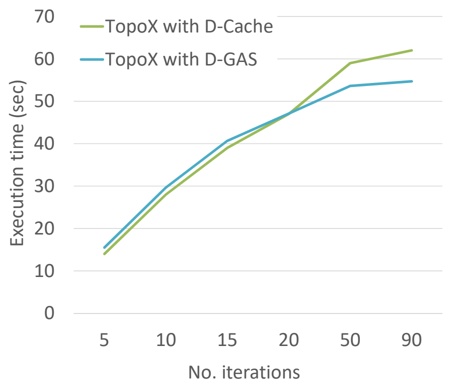


Figure 13: D-GAS vs. D-Cache.

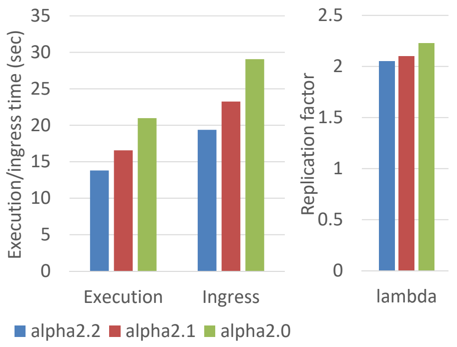


Figure 14: TopoX Performance on various α (2.0, 2.1 and 2.2).

The results are shown in Fig. 13, where the execution time is affected mainly by two factors, namely, the convergence speed (i.e., how many vertices become inactive in each iteration) and the per-vertex computation/communication cost. In the first few iterations, D-GAS has slightly higher execution time, because most vertices are active in which case D-GAS has slightly more computation for each vertex (as introduced in §5.2). After 20 iterations D-GAS achieves lower execution time because many vertices already get converged in which case D-GAS performs better. TopoX prefers D-GAS (if applicable) to the standard GAS model when it needs to run many iterations for more accurate results.

6.6 Self-Evaluation

We evaluate the execution time, ingress time and replication factors (λ) of TopoX, running PR on α2.0, α2.1 and α2.2 graphs, respectively. The result (Fig. 14) shows that all the execution time, ingress time and λ of TopoX decrease as α increases, because higher α represents sparser edges.

We compare the ingress time and replication factors of the two partitioning methods (hybrid-BL and hybrid-LRF). The result (Fig. 15) shows that LRF's λ is only slightly lower than that of BL. Clearly, Hybrid-BL should be the default partitioning strategy for TopoX, because its overhead is much lower than that of hybrid-LRF.

There are mainly two pre-defined parameters for TopoX, namely, the degree threshold for differentiating low-degree and high-degree vertices (Lines 8 and 10 in Algorithm 1), and the radius threshold for controlling the sizes of super-vertices (Lines 15 and 24 in Algorithm 1). We first evaluate the impact of degree thresholds on replication factors. The result (Fig. 16) shows that λ is insensitive to a large range of degree thresholds (from 64 to 512), which also conforms to the result of PowerLyra [10].

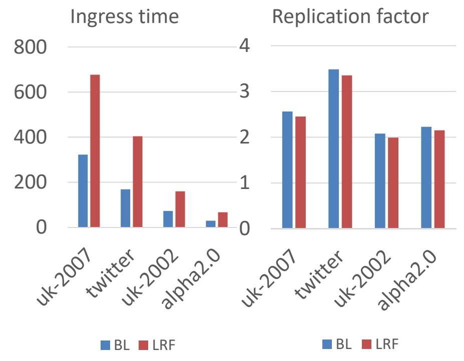


Figure 15: BL vs. LRF.

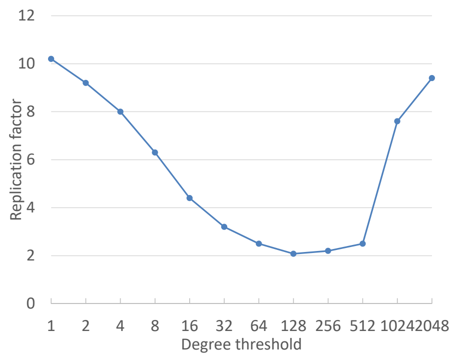


Figure 16: λ under different degree thresholds.

Next we study the radius threshold of TR. Since the graph size decides the maximum radius threshold (i.e., when a super-vertex becomes *too* large compared to the entire graph) which will not induce significant imbalance that cannot be handled in the assignment stage, we focus on the relation between graph size and radius threshold. We evaluate the impact of the radius threshold (γ) on UK-2002. We first evaluate the replication factors (λ) as γ increases from 1 to 10. The result is shown in Fig. 17, where λ (and consequently, the communication cost) decreases as γ increases but the decreasing rate becomes low after $\gamma \geq 3$.

We run PR for TopoX (with standard GAS) and measure the execution time of *each* of the 9 workers for different γ. The result is also shown in Fig. 17, where 9 colors respectively represent their execution time. When $\gamma > 3$, TopoX tends to have some worker whose execution time might be notably higher than others, because higher γ leads to larger super-vertices, which not only increase the possibility of conflict in parallel refactorization but also intensify the difficulty of load balancing in parallel partitioning. We also test various γ ($= 1 \sim 10$) on the large UK-2007 graph. Similar to UK-2002, The result for UK-2007 (omitted here) also shows that higher γ leads to higher performance but meanwhile is more likely to incur imbalance and dramatically degrade the overall performance when $\gamma > 4$, which is similar to the results for the middle-sized UK-2002 graph.

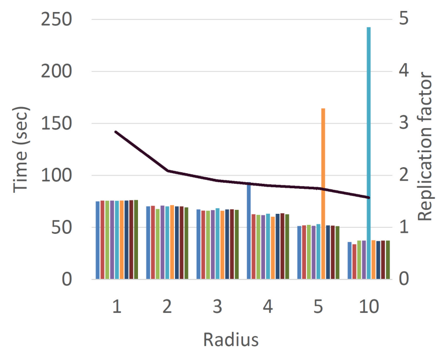


Figure 17: Radius threshold, execution time and λ.

7. RELATED WORK

Graph Partitioning. TopoX is inspired by prior edge-cut [31, 34, 47, 46, 48], vertex-cut [15, 21, 17, 29, 36, 5], and hybrid-cut [10] graph-parallel systems. For instance, Pregel [34] adopts hash-based edge-cut partitioning strategy to evenly assign vertices to machines, and provides message passing abstraction for vertices' interaction along edges. It follows the BSP (Bulk Synchronous Parallel) model [49] where all vertex-programs run simultaneously in multiple super-steps. GraphX [17] partitions a graph on top of Spark [57] by cutting the vertices and evenly assigning the edges to machines. It performs graph computation by following the GAS model (Section 2.3).

TopoX differs from them in adopting a novel refactorization scheme to achieve a balanced graph before partitioning by leveraging the topology information. Recently proposed vertex-splitting methods [30, 39] split a high-degree vertex into small ones to avoid bottlenecks, but they cannot place a large set of neighboring vertices in one machine. TopoX outperforms these systems in computation while only introducing small overhead.

Coarsening partitioning. Blogel [55] is a variation of the (edge-cut) Pregel [34] system, which performs multiple rounds of multi-source BFS to obtain and evenly distribute Voronoi [13] cells (i.e., blocks) to workers. A vertex v belongs to a block only if the block's center is closer to v than any other blocks' centers. Blogel differentiates intra- and inter-block communication to reduce communication cost through aggregation. Similar to Blogel, Giraph++ [47] coarsens a graph by grouping its vertices into subgraphs and opens up the subgraph structure to programmers to realize several algorithm-specific optimizations for traversal and aggregation applications. However, Giraph++ performs even worse than Blogel both in partitioning and in computation, because Giraph++ extends ParMetis [25] for coarsening, which is quite expensive [55] since it contains multiple rounds of matching phases (each having 2 supersteps) followed by a collapsing phase. Grape [14], G-Miner [9] and TurboGraph++ [26] also adopt coarsening and have similar problems with Blogel and Giraph++.

TopoX differs from them mainly in the following aspects. First, Blogel's edge-merging coarsens the original topology and requires aggregation of all vertices in a block before communication at each superstep, which cannot support the GAS model. In contrast, TopoX assigns vertices in a super-vertex to a worker while still supporting the more flexible vertex-level communication. Second, the construction of Voronoi cells requires Blogel to perform the vertex-to-block and block-to-worker assignments in sequence, which is time-consuming. In contrast, TopoX pipelines refactorization and partitioning and thus introduces little extra cost.

Dynamic and clustering partitioning. Some studies [20, 37, 56, 50, 54] are focused on dynamic and clustering partitioning for large-scale graphs. They incrementally partition the graph as vertices/edges are added and removed while still providing desirable computation and communication efficiency. For instance, LEOPARD [20] integrates a replication algorithm with the partitioning algorithm to reduce the number of replicated edges. Sedge [56] introduces a two-level partition structure, primary and secondary partitions, to handle queries with dynamic graphs. Vaquero et al. [50] design an dynamic partitioning algorithm that uses a lightweight heuristic relying only on local vertex information to provide a tradeoff between edge-cut efficiency and partitioning balance. Following their schemes, TopoX could easily realize dynamic topology maintenance.

Streaming partitioning. Partitioning may incur high overhead on large graphs, sometimes even higher than computation. Recent studies design mechanisms for streaming partitioning [48, 46, 61, 19]. For example, Fennel [48] designs a lightweight mechanism for streaming partitioning [46] which perform partitioning at the same time of graph loading. gSketch [61] combines traditional streaming technique and sketch partitioning to improve query estimation in graph streams. Inspired by these studies, TopoX pipelines not only graph loading and edge table construction but also refactorization and partitioning.

HPC-enabled graph computation. Some studies [63, 59, 43, 53, 22, 12, 35, 33, 44, 18] apply traditional HPC and database techniques (like shared memory, RDMA, versioning, and transactions) to enhance graph computation. For example, Ligra [44] is a single-machine graph processing framework for shared-memory multicore systems, which designs simple routines for mapping over edges/vertices to accelerate graph traversal algorithms that operate on subgraphs. Grazelle [18] is a pull-based shared-memory graph processing framework, which parallelizes/vectorizes graph computation loops by designing (i) the scheduler-aware interface to reduce write traffic and synchronization and (ii) the Vector-Sparse edge-representation format to enable loop vectorization. These studies are focused on powerful shared-memory machines with tens of cores and TB of memory, and thus are not for COTS (commercial off-the-shelf) clusters in the cloud. Since refactorization is general and orthogonal to these techniques, our proposal could be integrated to them for larger graphs and higher performance in distributed environments.

8. CONCLUSION

This paper introduces TopoX, a new graph-parallel system that refactorizes the topology of a skewed graph into a more communication-efficient one. Extensive evaluation results show that TopoX outperforms state-of-the-art graph-parallel systems while only introducing small overhead. In the future, we will study the adaptive mechanism for dynamic maintenance by adopting on-demand edge migration, and apply HPC techniques to TopoX for higher communication and computation performance.

9. REFERENCES

[1] http://law.di.unimi.it/webdata/uk-2002/.

[2] http://law.di.unimi.it/webdata/uk-2007-05/.

[3] https://en.wikipedia.org/wiki/Abelian_group.

[4] https://en.wikipedia.org/wiki/Graph_coloring.

[5] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1456–1465. ACM, 2014.

[6] S. Brandt and R. Wattenhofer. Approximating small balanced vertex separators in almost linear time. In *Workshop on Algorithms and Data Structures*, pages 229–240. Springer, 2017.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[8] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 95–106. ACM, 2008.

[9] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, page 32. ACM, 2018.

[10] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *European Conference on Computer Systems*, pages 1–15, 2015.

[11] F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics*, 6(2):125–145, 2002.

[12] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: a high-performance, transactional graph database based on refinable timestamps. *PVLDB*, 9(11):852–863, 2016.

[13] M. Erwig. The graph voronoi diagram with applications. *Networks: An International Journal*, 36(3):156–163, 2000.

[14] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 495–510. ACM, 2017.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 17–30, 2012.

[16] J. E. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 203–212. AUAI Press, 2009.

[17] 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 *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 599–613, 2014.

[18] S. Grossman, H. Litz, and C. Kozyrakis. Making pull-based graph processing performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 246–260. ACM, 2018.

[19] S. Guha, A. McGregor, and D. Tench. Vertex and hyperedge connectivity in dynamic graph streams. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 241–247. ACM, 2015.

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

[21] N. Jain, G. Liao, and T. L. Willke. Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*, page 4. ACM, 2013.

[22] X. Ju, D. Williams, H. Jamjoom, and K. G. Shin. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[23] S. Kamvar, T. Haveliwala, C. Manning, and G. Golub. Exploiting the block structure of the web for computing pagerank. Technical report, Stanford, 2003.

[24] G. Karypis and V. Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[25] G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *PPSC*, 1997.

[26] S. Ko and W.-S. Han. Turbograph++: A scalable and fast graph analytics system. In *Proceedings of the 2018 International Conference on Management of Data*, pages 395–410. ACM, 2018.

[27] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? *Www '10 Proceedings of International Conference on World Wide Web*, pages 591–600, 2010.

[28] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

[29] D. Li, C. Zhang, J. Wang, H. Xu, Z. Zhang, and Y. Zhang. Grapha: Adaptive partitioning for natural graphs. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017.

[30] L. Li, R. Geda, A. B. Hayes, Y. Chen, P. Chaudhari, E. Z. Zhang, and M. Szegedy. A simple yet effective balanced edge partition model for parallel computing. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):14, 2017.

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

[32] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB*, 8(3):281–292, 2014.

[33] L. Ma, H. Chen, J. Xue, and Y. Dai. Garaph: Efficient

gpu-accelerated graph processing on a single machine with balanced replication. In *USENIX Annual Technical Conference (ATC 17)*. USENIX Association.

[34] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. *Sigmod*, pages 135–146, 2009.

[35] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel. Graph: Heterogeneity-aware graph computation with adaptive partitioning. In *Proc. of IEEE ICDCS*, 2016.

[36] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[37] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 145–156. ACM, 2012.

[38] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[39] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 622–636. ACM, 2018.

[40] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 243–252. ACM, 2015.

[41] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[42] V. Satuluri, S. Parthasarathy, and Y. Ruan. Local graph sparsification for scalable clustering. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 721–732. ACM, 2011.

[43] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[44] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[45] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1):703–710, 2010.

[46] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[47] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *PVLDB*, 7(3):193–204, 2013.

[48] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342. ACM, 2014.

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

[50] L. M. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. Adaptive partitioning for large-scale dynamic graphs. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 144–153. IEEE, 2014.

[51] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *PVLDB*, 10(5):493–504, 2017.

[52] K. Wang, G. Xu, Z. Su, and Y. D. Liu. Graphq: Graph query processing with abstraction refinementscalable and programmable analytics over very large graphs on a single pc. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 387–401, 2015.

[53] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou. Tux2: Distributed graph computation for machine learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 669–682. USENIX Association, 2017.

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

[55] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[56] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 517–528. ACM, 2012.

[57] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[58] K. Zhang, R. Chen, and H. Chen. Numa-aware graph-structured analytics. In *ACM SIGPLAN Notices*, volume 50, pages 183–193. ACM, 2015.

[59] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng. Exploring the hidden dimension in graph processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[60] Y. Zhang and L. Liu. Distance-aware bloom filters: Enabling collaborative search for efficient resource discovery. *Future Generation Computer Systems*, 29(6):1621–1630, 2013.

[61] P. Zhao, C. C. Aggarwal, and M. Wang. gsketch: on query estimation in graph streams. *PVLDB*, 5(3):193–204, 2011.

[62] X. Zhao, A. Chang, A. D. Sarma, H. Zheng, and B. Y. Zhao. On the embeddability of random walk distances. *PVLDB*, 6(14):1690–1701, 2013.

[63] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[64] X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, 2015.