# PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs

RONG CHEN, JIAXIN SHI, YANZHE CHEN, BINYU ZANG, HAIBING GUAN, and HAIBO CHEN, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China

Natural graphs with skewed distributions raise unique challenges to distributed graph computation and partitioning. Existing graph-parallel systems usually use a "one size fits all" design that uniformly processes all vertices, which either suffer from notable load imbalance and high contention for high-degree vertices (e.g., Pregel and GraphLab) or incur high communication cost and memory consumption even for low-degree vertices (e.g., PowerGraph and GraphX).

In this paper, we argue that skewed distributions in natural graphs also necessitate differentiated processing on high-degree and low-degree vertices. We then introduce PowerLyra, a new distributed graph processing system that embraces the best of both worlds of existing graph-parallel systems. Specifically, PowerLyra uses centralized computation for low-degree vertices to avoid frequent communications and distributes the computation for high-degree vertices to balance workloads. PowerLyra further provides an efficient hybrid graph partitioning algorithm (i.e., hybrid-cut) that combines edge-cut (for low-degree vertices) and vertex-cut (for high-degree vertices) with heuristics. To improve cache locality of inter-node graph accesses, PowerLyra further provides a locality-conscious data layout optimization. PowerLyra is implemented based on the latest GraphLab and can seamlessly support various graph algorithms running in both synchronous and asynchronous execution modes. A detailed evaluation on three clusters using various graph-analytics and MLDM (Machine Learning and Data Mining) applications shows that PowerLyra outperforms PowerGraph by up to 5.53X (from 1.24X) and 3.26X (from 1.49X) for real-world and synthetic graphs accordingly, and is much faster than other systems like GraphX and Giraph, yet with much less memory consumption. A porting of hybrid-cut to GraphX further confirms the efficiency and generality of PowerLyra.

CCS Concepts: • **Information systems** → **Computing platforms**; • **Computing methodologies** → *Distributed computing methodologies*; Distributed programming languages.

Additional Key Words and Phrases: graph computation, graph partitioning, power-law degree distribution, skewed graph, locality-conscious data layout

**ACM Reference Format:**
Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2018. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* 1, 1, Article 1 (November 2018), 40 pages. https://doi.org/0000001.0000001

## 1 INTRODUCTION

Graph-structured computation has become increasingly popular, which is evidenced by its adoption in a wide range of areas including social computation, web search, natural language processing, and

This article extends a prior conference version that received the "Best Paper Award" of the Tenth ACM SIGOPS European Conference on Computer Systems (EuroSys 2015) [12]. Authors' addresses: R. Chen, J. Shi, Y. Chen, and H. Chen, Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, 200240, China; Corresponding author: Haibo Chen (haibochen@sjtu.edu.cn).

1539-9087/2018/11-ART1 $15.00
https://doi.org/0000001.0000001

recommendation systems [5, 25, 54, 67, 88, 98]. The intense desire for efficient and expressive programming models for graph-structured computation has recently driven the development of numerous distributed graph-parallel systems such as Pregel [47], GraphLab [43], and PowerGraph [24]. They usually follow the "think like a vertex" philosophy [47] by coding graph computation as vertex-centric programs to process vertices in parallel and communicate across edges.

On the other hand, the distribution of real-world graphs tends to be diverse and constantly evolving [41]. For example, some real-world datasets exhibit a skewed power-law distribution [21, 52] where a small number of vertices have a significant number of neighboring vertices, while some other existing datasets (e.g., road networks) exhibit a more balanced distribution. The diverse properties inside and among graph datasets raise new challenges to efficiently partition and process such graphs [1, 24, 42].

Unfortunately, existing graph-parallel systems usually adopt a "one size fits all" design where different vertices are equally processed, leading to suboptimal performance and scalability. For example, Pregel [47] and GraphLab [43] centralize their designs in making resources locally accessible to hide latency. This is done by evenly distributing vertices to machines, which may result in imbalanced computation and communication for vertices with high degrees (i.e., the number of neighboring vertices). In contrast, PowerGraph [24] and GraphX [26] focus on evenly parallelizing the computation by partitioning edges among machines, which incurs high communication cost among partitioned vertices even with low degrees.

Further, prior graph partitioning algorithms may result in suboptimal performance for both skewed and non-skewed (i.e., regular) graphs. For example, edge-cut [36, 61, 68, 72], which divides a graph by cutting cross-partition edges among sub-graphs with the goal of evenly distributing vertices, usually results in excessive replication of edges as well as imbalanced messages with high contention. In contrast, vertex-cut [8, 24, 31], which partitions vertices among sub-graphs with the goal of evenly distributing edges, incurs high communication overhead among partitioned vertices and excessive memory consumption.

In this paper, we make a comprehensive analysis of existing graph-parallel systems over skewed graphs and argue that the diverse properties of different graphs and the skewed vertex distributions demand differentiated computation and partitioning on low-degree and high-degree vertices. Based on our analysis, we introduce PowerLyra, a new distributed graph processing system that embraces the best of both worlds of existing systems. The key idea of PowerLyra is to process different vertices adaptively according to their degrees.

PowerLyra follows the GAS (i.e., Gather, Apply and Scatter) programming interface of PowerGraph [24] and can seamlessly support existing graph algorithms running in either synchronous and asynchronous execution mode. Internally, PowerLyra distinguishes the processing of low-degree and high-degree vertices: it uses centralized computation for low-degree vertices to avoid frequent communications and only distributes the computation for high-degree vertices to balance workloads.

To efficiently partition a skewed graph, PowerLyra is built with a balanced $p$-way hybrid-cut algorithm to partition different types of vertices for a skewed graph. The random (i.e., hash-based) hybrid-cut evenly distributes low-degree vertices along with their edges among machines (like edge-cut) and evenly distributes edges of high-degree vertices among machines (like vertex-cut). We further provide a greedy heuristic to improve partitioning of low-degree vertices in a skewed graph.

Finally, PowerLyra mitigates the poor locality and high interference among threads during the communication phase by a locality-conscious data layout optimization built atop hybrid-cut. It trades a small increase of pre-processing time for a notable speedup during graph computation.

We have implemented PowerLyra[1] based on GraphLab PowerGraph v2.2[2] (released in February 2015), which comprises about 3000 lines of C++ code. Since the first release of PowerLyra in 2015, there have been a number of graph-structured systems inspired by our hybrid approach [10, 56, 64, 73, 91, 95], employing hybrid partitioning algorithms [32, 84, 93, 94], or directly implemented based on PowerLyra [33, 40, 75, 77]. The hybrid-cut algorithm has also been used by WeChat, one of the world's largest social platforms with over 1 billion active users, for anomaly detection in its social platform [70].

Our evaluation on three different clusters using various graph-analytics and MLDM applications shows that PowerLyra outperforms PowerGraph by up to 5.53X (from 1.24X) and 3.26X (from 1.49X) for real-world and synthetic graphs accordingly, and consumes much less memory, due to significantly reduced replication factor, less communication cost, and better locality in computation and communication. A porting of the hybrid-cut to GraphX further confirms the efficiency and generality of PowerLyra.

This paper makes the following contributions:

- A comprehensive analysis that uncovers some performance issues of existing graph-parallel systems (§2).
- The PowerLyra model that supports differentiated computation on low-degree and high-degree vertices, as well as adaptive communication with minimal messages while not sacrificing generality (§3).
- A hybrid-cut algorithm with heuristics that provides more efficient partitioning and computation (§4), as well as a locality-conscious data layout optimization (§5).
- A comprehensive evaluation that demonstrates the performance benefits of PowerLyra (§6).

## 2 BACKGROUND AND MOTIVATION

Many graph-parallel systems, including PowerLyra, abstract computation as a vertex-centric program $P$, which is executed in parallel on each vertex $v \in V$ in a sparse graph $G = \{V, E\}$. The scope of computation and communication in each $P(v)$ is restricted to the neighboring vertices $n$ through edges where $(v, n) \in E$.

This section briefly introduces skewed graphs and illustrates why prior graph-parallel systems fall short using Pregel, GraphLab, PowerGraph, and GraphX as examples, as they are representatives of existing systems.

### 2.1 Skewed Graphs

Natural graphs, such as social networks (e.g., follower, citation, and co-authorship), email and instant messaging graphs, or web graphs (hubs and authorities), usually exhibit a skewed distribution, such as the power-law degree distribution [21]. This implies that a large fraction of vertices have relatively few neighbors (i.e., low-degree vertex), while a small fraction of vertices has a significant number of neighbors (i.e., high-degree vertex). Given a positive power-law constant $\alpha$, the probability that the vertex has the degree $d$ under the power-law distribution is

$$P(d) \propto d^{-\alpha}$$

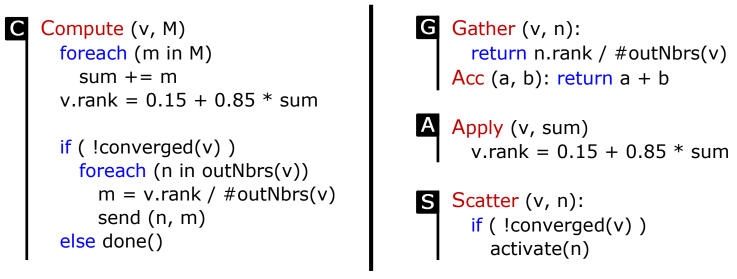
The lower exponent $\alpha$ implies that a graph has higher density and more high-degree vertices. For example, the *in* and *out* degree distribution of the Twitter follower graph is close to 1.7 and 2.0 accordingly [24]. Though there are also other models [41, 58, 59, 82] for skewed graphs, we restrict the discussion to the power-law distribution due to space constraints. However, PowerLyra is not

[1]The source code and a brief instruction of PowerLyra are available at http://ipads.se.sjtu.edu.cn/projects/powerlyra.html
[2]GraphLab prior to 2.1 runs the distributed GraphLab engine (i.e., GraphLab [43]).

Table 1. A summary of typical distributed graph-parallel systems. 'L' and 'H' represent low-degree and high-degree vertex respectively.

| | **Pregel-like** | **GraphLab** | **PowerGraph** | **GraphX** | **PowerLyra** |
|---|---|---|---|---|---|
| **Graph Placement** | edge-cuts | edge-cuts | vertex-cuts | vertex-cuts | hybrid-cuts |
| **Comp. Pattern** | local | local | distributed | distributed | L: local<br>H: distributed |
| **Comm. Cost** | $\leq \#edge\text{-}cuts$ | $\leq 2 \times \#mirrors$ | $\leq 5 \times \#mirrors$ | $\leq 4 \times \#mirrors$ | L: $\leq \#mirrors$<br>H: $\leq 4 \times \#mirrors$ |
| **Dynamic Comp.** | no | yes | yes | yes | yes |
| **Workload Balance** | no | no | yes | yes | yes |

bound to such a distribution and should benefit other models (e.g., bipartite graph [13, 14]) with skewed distributions as well (having high-degree and low-degree vertices).



```
C Compute (v, M)
    foreach (m in M)
      sum += m
    v.rank = 0.15 + 0.85 * sum

    if ( !converged(v) )
      foreach (n in outNbrs(v))
        m = v.rank / #outNbrs(v)
        send (n, m)
    else done()
```

(a) Pregel/GraphLab

```
G Gather (v, n):
    return n.rank / #outNbrs(v)
  Acc (a, b): return a + b

A Apply (v, sum)
    v.rank = 0.15 + 0.85 * sum

S Scatter (v, n):
    if ( !converged(v) )
      activate(n)
```

(b) PowrGraph/PowerLyra

Fig. 1. The sample code of PageRank on various systems.

## 2.2 Existing Graph-parallel Systems

Though a skewed graph has different types of vertices, existing graph systems usually use a "one size fits all" design and compute equally on all vertices, which may result in suboptimal performance. Table 1 provides a comparative study of typical distributed graph-parallel systems. Additional related work can be found in §7.

**Pregel** and its open-source relatives (e.g., Giraph[3], Hama[4], and GPS [60]) use the BSP (*Bulk Synchronous Parallel*) model [74] with explicit messages to fetch all resources for the vertex computation. Figure 1(a) illustrates an example implementation of PageRank [5] in Pregel. The `Compute` function sums up the ranks of neighboring vertices through the received messages $M$ and sets it as the new rank of the current vertex. The new rank will also be sent to its neighboring vertices by messages until reaching a global convergence estimated by a distributed aggregator. As shown in Figure 2, Pregel adopts traditional edge-cut [61, 68, 72] to evenly assign vertices among machines

[3]Apache Giraph: http://giraph.apache.org/
[4]Apache Hama: http://hama.apache.org/

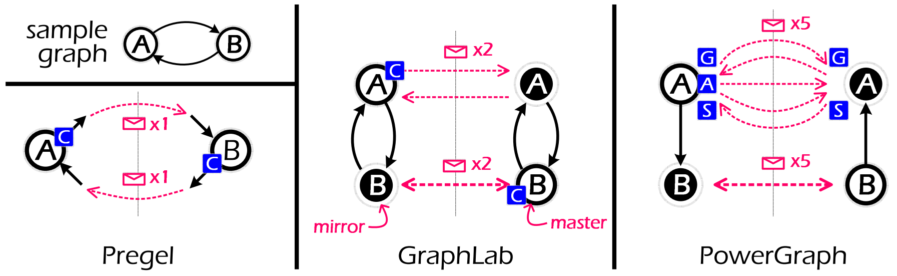


Fig. 2. A comparison of distributed graph-parallel models.

and provides interaction between vertices by message passing along edges. Since the communication is restricted to push-mode algorithms (e.g., vertex A cannot actively pull data from vertex B), Pregel does not support dynamic computation[5] [34, 43].

**GraphLab** replicates vertices for all edges spanning machines and leverages an additional vertex activation message to support dynamic computation [43]. In Figure 2, GraphLab also uses edge-cut as Pregel, but creates replicas (i.e., mirrors) and duplicates edges in both machines (e.g., for the edge from vertex A to B, there are one edge and one replica in both machines). The communication between replicas of a vertex is bidirectional, i.e., sending updates from a master to its mirrors and activation messages from mirrors to the master. PageRank implemented in GraphLab is similar to that of Pregel, except that it uses replicas to exchange messages from neighboring vertices.

**PowerGraph** abstracts computation into the GAS (Gather, Apply and Scatter) model and uses vertex-cut [24, 31] to assign edges evenly among machines with replicated vertices. A single vertex can be split into multiple replicas in different machines to parallelize the computation on it. Figure 1(b) uses the `Gather` and the `Acc` functions to accumulate the rank of neighboring vertices along in-edges, the `Apply` function to calculate and update a new rank to vertex, and the `Scatter` function to send messages and activate neighboring vertices along out-edges. Five messages for each replica are used to parallelize vertex computation to multiple machines in each iteration (i.e., 2 for Gather, 1 for Apply and 2 for Scatter), three of them are used to support dynamic computation. As shown in Figure 2, the edges of a single vertex are assigned to multiple machines to distribute workloads evenly, and the replicas of the vertex are placed in machines with its edges.

**GraphX** [26] builds on top of Apache Spark [90], a general dataflow framework by recasting graph-specific operations into analytics pipelines formed by basic dataflow operators such as Join, Map, and Group-by. GraphX also adopts vertex replication, incremental view maintenance, and vertex-cut partitioning to support dynamic computation and balance the workloads for skewed graphs.

There also have many single-machine graph-parallel systems [39, 46, 53, 57, 65, 66, 92, 101] designed for the case where a graph can fit within a single machine. This largely mitigates the cost of distributed graph computation and partitioning. However, they are unlikely to completely displace distributed graph-parallel systems. For example, industrial-scale graphs like those from Google [47] and Facebook [6] are unlikely to be fit within a single machine, making distributed graph computation indispensable to process them in a timely manner. It can be evidenced by the fact that many well-known companies like Google [47], Facebook [19], and Alibaba [30] deployed distributed graph-parallel systems in their production systems. Hence, in this paper, we mainly focus on distributed graph-parallel systems.

[5]Dynamic computation allows only some of the vertices to be active in each iteration.

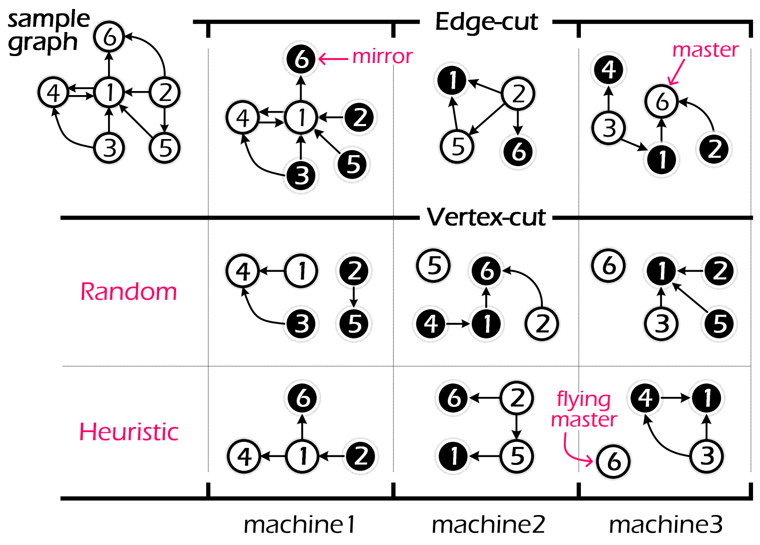


Fig. 3. A comparison of graph partitioning algorithms. The unshaded and shaded circles represent the masters and mirrors respectively.

## 2.3 Issues with Graph Computation

To exploit *locality* during computation, both Pregel and GraphLab use edge-cut to accumulate all resources (i.e., messages or replicas) of a vertex in a single machine. However, a skewed distribution of degrees among vertices implies skewed workload, which leads to substantial imbalance when being accumulated on a single machine. Even if the number of high-degree vertices is much more than the number of machines to balance workload [29], it still incurs heavy network traffic among machines to accumulate all resources for high-degree vertices. Further, high-degree vertices would be the center of contention when performing scatter operations on all edges in a single machine. As shown in Figure 3, there is significant load imbalance for edge-cut in Pregel and GraphLab, as well as high contention on vertex 1 (high-degree) when its neighboring vertices activate it in parallel. This situation will be even worse with the increase of machines and degrees of vertices.

PowerGraph and GraphX address the load imbalance issue using vertex-cut and decomposition under the GAS model, which split a vertex into multiple replicas across machines. However, this splitting also comes at a cost, including more computation, communication, and synchronization required to gather values and scatter the new value from/to its replicas (see Figure 2). However, as a large fraction of vertices only has a small degree in a skewed graph, splitting such vertices is not worthwhile. Further, while the GAS model provides a general abstraction, many algorithms only gather or scatter in one direction (e.g., PageRank). Unfortunately, both PowerGraph and GraphX still require redundant communications and data movements. The workload is always distributed to all replicas even without such edges. Under the Random vertex-cut in Figure 3, the computation on vertex 4 still needs to follow the GAS model, even though all in-edges are located together with the master of vertex 4.

## 2.4 Issues with Graph Partitioning

Graph partitioning plays a vital role in reducing communication and ensuring load balance. Traditional balanced $p$-way **edge-cut** [61, 68, 72] evenly assigns *vertices* of a graph to $p$ machines to minimize the number of *edges* spanning machines. Under edge-cut in Figure 3, six vertices are

Table 2. A comparison of various vertex-cuts for 48 partitions using PageRank (10 iterations) on the Twitter follower graph and ALS ($d$=20, the magnitude of latent dimension.) on the Netflix movie recommendation graph. $\lambda$ presents replication factor. Pre-processing times include loading graph into memory and building local graph structures.

| **Algorithm & Graph** | **Vertex-cut** | $\lambda$ | **Time (Sec)** | |
|---|---|---|---|---|
| | | | **Pre-processing** | **Execution** |
| PageRank & Twitter follower | Random | 16.0 | 263 | 823 |
| | Coordinated | 5.5 | 391 | 298 |
| | Oblivious | 12.8 | 289 | 660 |
| | Grid | 8.3 | 123 | 373 |
| | **Hybrid** | **5.6** | 138 | **155** |
| ALS ($d$=20) & Netflix movie recommendation | Random | 36.9 | 21 | 547 |
| | Coordinated | 5.3 | 31 | 105 |
| | Oblivious | 31.5 | 25 | 476 |
| | Grid | 12.3 | 12 | 174 |
| | **Hybrid** | **2.6** | 14 | **67** |

randomly (i.e., hash modulo #machine ) assigned to three machines. Edge-cut creates replicated vertices (e.g., mirrors) and edges to form a locally consistent graph state in each machine. However, natural graphs with skewed distributions are difficult to partition using edge-cut [1], since skewed vertices will cause a burst of communication cost and work imbalance. Vertex 1 in Figure 3 contributes about half of the replicas of vertices and edges, and incurs load imbalance on machine 1, which has close to half of edges.

The balanced $p$-way **vertex-cut** [24] evenly assigns *edges* to $p$ machines to minimize the number of *vertices* spanning machines. Compared to edge-cut, vertex-cut avoids replication of edges and achieves load balance by allowing edges of a single vertex to be split over multiple machines. However, randomly constructed vertex-cut leads to much higher replication factor ($\lambda$) (i.e., the average number of replicas for a vertex), since it incurs poor placement of low-degree vertices. In Figure 3, Random vertex-cut creates a mirror for vertex 3 even if it has only two edges[6].

To reduce replication factor, PowerGraph uses a *greedy* heuristic [24] to accumulate adjacent edges on the same machine. In practice, applying the greedy heuristic to all edges (i.e., Coordinated [24]) incurs a significant penalty during graph partitioning [29], mainly caused by exchanging vertex information among machines. Yet, using greedy heuristics independently on each machine (i.e., Oblivious [24]) will notably increase the replication factor.

The *constrained* vertex-cut [31] (e.g., Grid) is proposed to strike a balance between pre-processing and execution time. It follows the classic 2D partitioning [9, 89] to restrict the locations of edges within a small subset of machines to approximate an optimal partitioning. Since the set of machines for each edge can be calculated on each machine independently by hashing, constrained vertex-cut can significantly reduce the pre-processing time[7]. However, the ideal upper bound of replication factor is still too large for a good placement of low-degree vertices (e.g., $2\sqrt{N}-1$ for Grid [31]). Further, constrained vertex-cut necessitates the number of partitions ($N$) close to being a square number for a reasonably balanced graph partitioning.

[6]PowerGraph mandates the creation of a *flying* master of vertex (e.g., vertex 5 and 6) in its hash-based location to support simple external querying for some algorithms even without edges. PowerLyra also follows this rule.
[7]Coordinated greedy vertex-cut has been deprecated due to its excessive pre-processing time and buggy, meanwhile both PowerGraph and GraphX have adopted Grid-like vertex-cut as the preferential partitioning algorithm.

Some prior work (e.g., [29]) argues that intelligent graph placement schemes may dominate and hurt the total execution time. However, such an argument just partially[8] holds for *greedy heuristic partitioning and simple graph algorithms*. First, a naive random partitioning scheme does not necessarily imply high efficiency in pre-processing time due to a lengthy time to create an excessive number of mirrors. Second, with the increasing sophistication of graph algorithms (e.g., MLDM), the pre-processing time will become relatively small to the overall computation time. In addition, unlike graph computation, graph partitioning only needs to be performed once for each graph, and the resulting partitions can be reused later to amortize the pre-processing time across multiple runs.

Table 2 illustrates a comparison of various state-of-the-art vertex-cuts of PowerGraph for 48 partitions. *Random* vertex-cut performs worst in both pre-processing and computation time due to the highest replication factor. *Coordinated* vertex-cut achieves both small replication factor and execution time but at the cost of excessive pre-processing time. *Oblivious* vertex-cut reduces pre-processing time (but still slower than Random) while doubling replication factor and overall execution time. *Grid* vertex-cut outperforms coordinated vertex-cuts in pre-processing time by 2.8X but decreases graph computation performance. Besides, the percent of pre-processing time for PageRank on the Twitter follower graph with 10 iterations[9] of graph computation ranges from 24.2% to 56.8%, while for ALS on Netflix movie recommendation graph it only ranges from 3.6% to 22.8%. The random *Hybrid-cut* of PowerLyra (§4) provides optimal performance by significantly decreasing execution time while only slightly increasing pre-processing time (even compared to Grid).

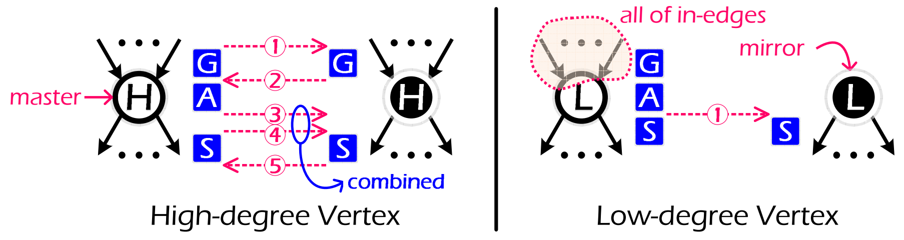


Fig. 4. The computation model on high-degree and low-degree vertex for algorithms gathering along in-edges and scattering along out-edges.

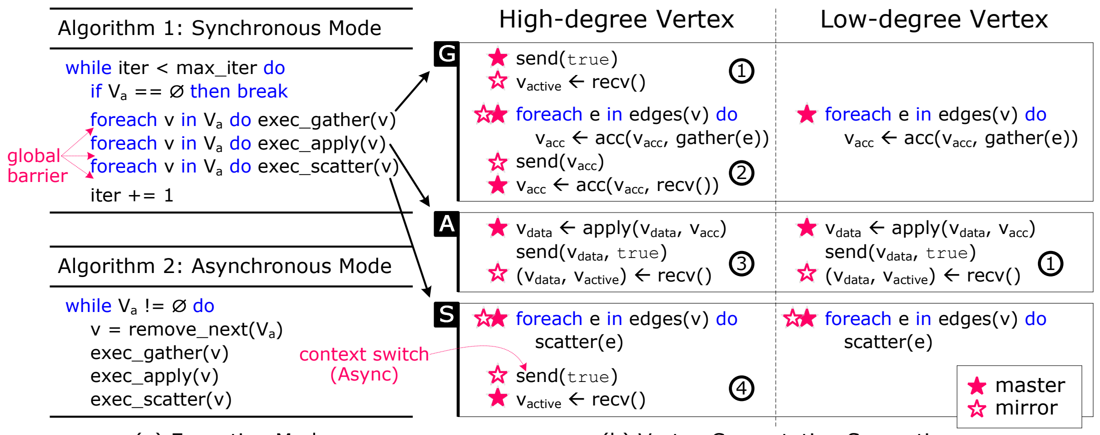
## 3 GRAPH COMPUTATION IN POWERLYRA

This section describes the graph computation model in PowerLyra, which combines the best from prior systems by differentiating the processing on high-degree and low-degree vertices. Moreover, PowerLyra supports both the highly parallel synchronous execution mode as well as the computationally efficient asynchronous execution mode. Finally, to preserve generalization, PowerLyra proposes an adaptive way to downgrade the computation model for low-degree vertices. Without loss of generality, the rest of this paper will use *in-degree* of the vertex to introduce the design of PowerLyra's hybrid computation model.

### 3.1 Graph-parallel Abstraction

Like others, a vertex-program $P$ in PowerLyra runs on a directed graph $G = \{V, E\}$ and computes in parallel on each vertex $v \in V$. Users can associate arbitrary vertex data $D_v$ where $v \in V$, and edge data $D_{s,t}$ where $(s,t) \in E$. Computation on vertex $v$ can gather and scatter data from/to neighboring vertex $n$ where $(v,n) \in E$. During graph partitioning (§4), PowerLyra replicates vertices to construct

[8]GraphLab has been highly optimized in the v2.2 release, especially for pre-processing time with parallel loading.
[9]Increasing iterations, like [29, 47], will further reduce the proportion.



```
Algorithm 1: Synchronous Mode
while iter < max_iter do
    if V_a == ∅ then break
    foreach v in V_a do exec_gather(v)
    foreach v in V_a do exec_apply(v)
    foreach v in V_a do exec_scatter(v)
    iter += 1
```

```
Algorithm 2: Asynchronous Mode
while V_a != ∅ do
    v = remove_next(V_a)
    exec_gather(v)
    exec_apply(v)
    exec_scatter(v)
```

```
High-degree Vertex
G  ★ send(true)                                ①
   ☆ v_active ← recv()
   ☆★ foreach e in edges(v) do
          v_acc ← acc(v_acc, gather(e))
   ☆ send(v_acc)                               ②
   ★ v_acc ← acc(v_acc, recv())
A  ★ v_data ← apply(v_data, v_acc)
     send(v_data, true)                        ③
   ☆ (v_data, v_active) ← recv()
S  ☆★ foreach e in edges(v) do
          scatter(e)
   ☆ send(true)                                ④
   ★ v_active ← recv()
```

```
Low-degree Vertex
G  ★ foreach e in edges(v) do
          v_acc ← acc(v_acc, gather(e))
A  ★ v_data ← apply(v_data, v_acc)
     send(v_data, true)                        ①
   ☆ (v_data, v_active) ← recv()
S  ☆★ foreach e in edges(v) do
          scatter(e)
```

(a) Execution Mode (b) Vertex Computation Semantics

Fig. 5. The algorithms of execution engine in synchronous and asynchronous mode, and the vertex execution semantics of high-degree and low-degree vertex. The solid and hollow stars indicate the master and mirror of vertex respectively, and the numbered circles indicate the message passing.

a local graph on each machine, all of which are called **replicas**. Like PowerGraph, PowerLyra also elects a replica randomly (using vertex's hash) as **master** and other replicas as **mirrors**. PowerLyra still strictly conforms to the GAS model, and hence can seamlessly run all existing applications in PowerGraph.

## 3.2 Differentiated Vertex Computation

PowerLyra employs a simple loop to express iterative computation of graph algorithms and processes vertices differently according to the vertex degrees. Figure 5(b) illustrates the detailed vertex computation semantics of high-degree and low-degree vertex.

**Processing high-degree vertex:** To exploit the **parallelism** of vertex computation, PowerLyra follows the GAS model in PowerGraph to process high-degree vertices. In the Gather phase, two messages are sent by the master vertex (hereinafter master for short) to activate all mirrors to run the `gather` function locally and accumulate results back to the master. In the Apply phase, the master runs the `apply` function and then sends the updated vertex data to all its mirrors. Finally, all mirrors execute the `scatter` function to activate their neighbors, and the master will similarly receive notification from activated mirrors. Unlike PowerGraph, PowerLyra groups the two messages from master to mirrors in the Apply and Scatter phases (see the left part of Figure 4), to reduce message exchanges.

**Processing low-degree vertex:** To preserve access **locality** of vertex computation, PowerLyra introduces a new GraphLab-like computation model to process low-degree vertices. However, PowerLyra does not provide *bidirectional* (i.e., both *in* and *out*) access locality like GraphLab, which necessitates edge replicas and doubles messages. We observe that *most graph algorithms only gather or scatter data in only one direction*. For example, PageRank only gathers data along in-edges and scatters data along out-edges. PowerLyra leverages this observation to provide *unidirectional* access locality by placing vertices along with edges in only one direction (which has already been indicated by application code). As the update message from master to mirrors is inevitable after computation (in the Apply phase), PowerLyra adopts *local* gathering and *distributed* scattering to minimize the communication overhead.

As shown in the right part of Figure 4, since all edges required by gathering has been placed locally, both the Gather and Apply phases can be done locally by the master without the help of its mirrors. The message to activate mirrors (that further scatter their neighbors along out-edges) is combined with the message for updating vertex data (sent from master to mirrors) in Apply phase.

Finally, the notifications from mirrors to master in the Scatter phase are not necessary anymore, since only the master will be activated along in-edges by its neighbors. Compared to GraphLab, PowerLyra requires no replication of edges and only incurs up to ***one*** (update) message per mirror in each iteration for low-degree vertices. In addition, the unidirectional message from master to mirrors avoids potential contention on the receiving end of communication [11], since each mirror will receive at most one message (from its master) in each iteration.

## 3.3 Execution Mode

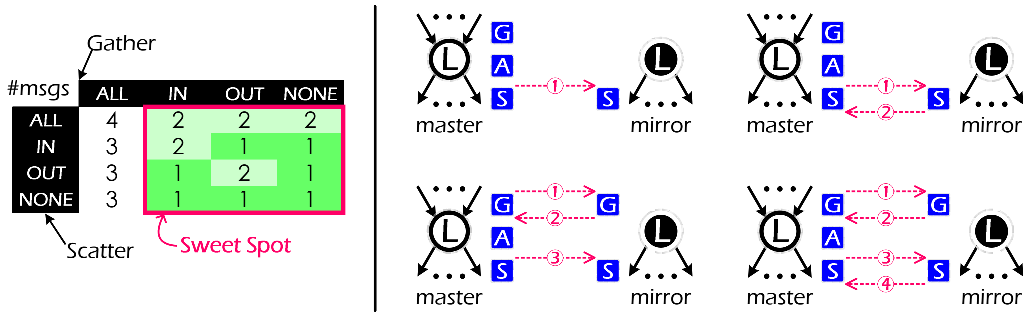
PowerLyra separates the computation mode of each vertex from the execution mode of the program. The PowerLyra engine maintains a set of *active* vertices $V_a$ regardless of high-degree or low-degree vertex and executes the vertex-program $P$ on each of them until none remains *active*. A vertex $v$ will be added to the set $V_a$ when activated by neighboring vertices or itself and removed from the set $V_a$ when its computation finished. The PowerLyra engine can execute the *active* vertices in both *synchronous* (`Sync`) and *asynchronous* (`Async`) modes. The main difference between two modes is the scheduling order of vertex computation, which provides difference tradeoffs in convergence rate, runtime overhead, and execution determinism [85].

**Synchronous Execution:** In synchronous mode, PowerLyra abstracts graph processing as a sequence of iterations (i.e., super-step) [47], in which all active vertices execute the vertex program in parallel using the values of neighboring vertices updated in the prior iteration. Inspired by PowerGraph [24], PowerLyra divides a super-step into several *mini-steps*, each of which synchronously completes the same phase of all active vertices with a global barrier at the end (see Algorithm 1 in Figure 5). Unlike PowerGraph, the low-degree vertices in PowerLyra can skip some operations (messages) in mini-steps, e.g., accumulation and networking in the Gather phase for all mirrors.

**Asynchronous Execution:** In asynchronous mode, PowerLyra abstracts graph processing as an iterative computation on the set of active vertices (see Algorithm 2 in Figure 5). Compared to the synchronous execution, the computation on an active vertex will directly use the latest value of its neighboring vertices. Since the execution on a vertex may be blocked for communication with its mirrors, a large number of user-mode worker threads are spawned to hide the effects of network latency. When the current worker thread is blocked, the underlying system thread will switch to another user-mode worker thread. The blocked thread will be re-scheduled after receiving the required number of results from mirrors. Unlike PowerGraph, the worker thread on low-degree vertices does not need to send messages in the Gather phase and to wait for messages from the mirrors in the Scatter phase. Therefore, in asynchronous mode, the hybrid approach can simplify the task scheduling and further reduce the cost from context switches for the low-degree vertices.

Table 3. A classification of various graph algorithms.

| **Type** | **`gather_edges`** | **`scatter_edges`** | **Example Algorithms** |
|---|---|---|---|
| **Natural** | `in_edges` or `none` | `out_edges` or `none` | PR, SSSP |
| | `out_edges` or `none` | `in_edges` or `none` | DIA [35] |
| **Other** | `any` | `any` | CC, ALS [99] |



| #msgs | ALL | IN | OUT | NONE |
|---|---|---|---|---|
| ALL | 4 | 2 | 2 | 2 |
| IN | 3 | 2 | 1 | 1 |
| OUT | 3 | 1 | 2 | 1 |
| NONE | 3 | 1 | 1 | 1 |

Fig. 6. The matrix of communication cost and sample computation models on low-degree vertex for various algorithms.

### 3.4 Generalization

PowerLyra applies a simplified model for low-degree vertices to minimize communication overhead. However, this may limit its expressiveness to some graph algorithms that may require gathering or scattering data in *both in and out* directions. PowerLyra introduces an adaptive approach to handling different graph algorithms. Note that PowerLyra only needs to use such an approach for low-degree vertices, since communication on high-degree vertices is already bidirectional.

PowerLyra classifies algorithms according to the directions of edges accessed in gathering and scattering, which are returned from the `gather_edges` and `scatter_edges` interfaces[10] of PowerGraph accordingly. Hence, it can be checked at runtime without any changes to applications. Table 3 summarizes the classification of graph algorithms. PowerLyra seamlessly supports the *Natural* algorithms that gather data along one direction (e.g., `in/out_edges`) or `none` and scatter data along another direction (e.g., `out/in_edges`) or `none`, such as *PageRank* (PR), *Single-Source Shortest Paths* (SSSP) and *Approximate Diameter* (DIA) [35][11]. For such algorithms, PowerLyra needs up to one message per mirror for a low-degree vertex in each iteration.

For *Other* algorithms that gather or scatter data via any edges, PowerLyra requires mirrors to do gathering or scattering operations like those of high-degree vertices, but only *on demand*. For example, the *Connected Components* (CC) application gathers data via `none` edges and scatters data via `all_edges`, so that PowerLyra only requires one additional message in the Scatter phase to notify the master by the activated mirrors, and thus still avoids unnecessary communication in the Gather phase.

Figure 6 lists the detailed communication cost for all combinations of the access models in Gather and Scatter phases, and illustrates four corresponding computation models with the unidirectional access locality along in-edges. PowerLyra only requires at most two messages in a large *sweet spot* (see the left of Figure 6) as long as the algorithm does not gather data in both directions (i.e., in and out). The two messages in Gather phase can be avoided, relying on unidirectional access locality provided by PowerLyra.

## 4 DISTRIBUTED GRAPH PARTITIONING

For distributed graph processing systems, graph partitioning plays a vital role in reducing communication and ensuring workload balance. Existing edge-cut and vertex-cut commonly use a "one

[10]The `gather/scatter_edges` interface returns the set of edges on which to run the gather/scatter function. The default edge direction is in/out edges.
[11]The detailed description of algorithms can be found in §6.1.

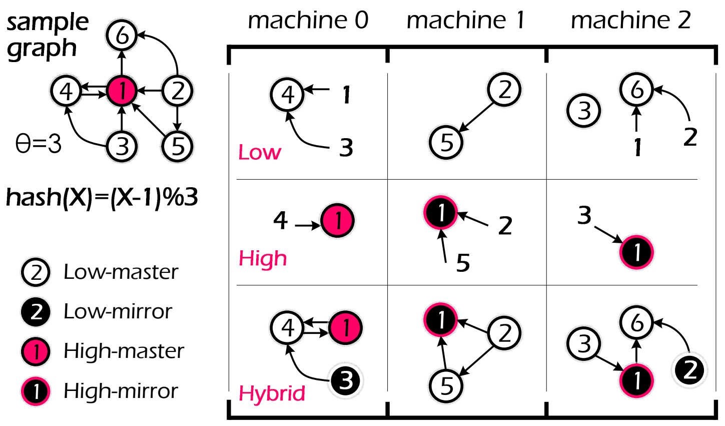


Fig. 7. The hybrid-cut on sample graph. The "Low" and "High" in each row represent the partitioning on low-degree and high-degree vertices respectively, and the combination of "Low" and "High" in each column (i.e., machine) means "Hybrid".

size fits all" design that uniformly places all vertices and edges. However, the skewed power-law degree distribution in natural graphs calls for differentiated mechanisms to process high-degree and low-degree vertices along with edges. In addition, the PowerLyra's abstraction relies on graph partitioning to provide unidirectional access locality for low-degree vertices.

This section describes a new hybrid-cut algorithm that uses differentiated partitioning strategies for low-degree and high-degree vertices, which embraces the locality of edge-cut and the parallelism of vertex-cut. Based this, a new heuristic, called *Ginger*, is provided to optimize partitioning for PowerLyra further. Finally, theoretical analysis, as well as empirical validation, are provided to compare new hybrid approaches with prior vertex-cut algorithms.

## 4.1 Balanced $p$-way Hybrid-Cut

Since vertex-cut evenly assigns edges to machines and only replicates vertices to construct a local graph within each machine, the memory and communication overhead highly depend on the replication factor ($\lambda$). Hence, existing vertex-cut algorithms mostly aim at reducing the overall $\lambda$ of all vertices. However, we observe that the key is to reduce $\lambda$ of *low-degree* vertices, since high-degree vertices inevitably need to be replicated on most of the machines. Distributing massive edges of high-degree vertex may incur a bursting increase of replicas for low-degree vertices. Nevertheless, many current heuristics for vertex-cuts have a bias towards high-degree vertices, while paying little attention to low-degree vertices.

We propose a balanced $p$-way **hybrid-cut** that focuses on reducing $\lambda$ of low-degree vertices. It uses differentiated partitioning to low-degree and high-degree vertices. To minimize replication of edges, each edge exclusively belongs to its target vertex (the destination of the edge)[12]. For low-degree vertices, hybrid-cut adopts *low-cut* to evenly assign vertices along with in-edges to machines by hashing their *target* vertices. For high-degree vertices, hybrid-cut adopts *high-cut* to distribute all

[12]The edge could also exclusively belong to its source vertex, which depends on the direction of locality preferred by the graph algorithm. Without loss the generality, we assume the unidirectional access locality along in-edges in the rest of this paper.

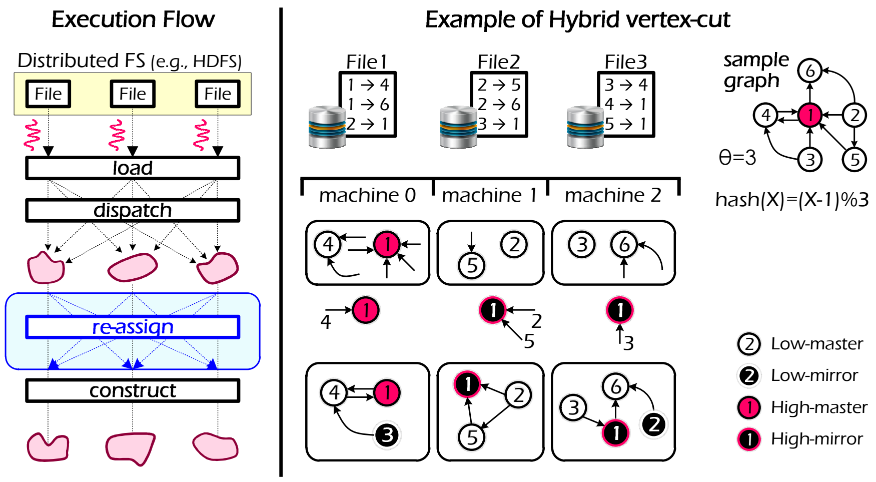


Fig. 8. The execution flow of hybrid-cut.

in-edges of vertices to machines by hashing their *source* vertices. After that, hybrid-cut creates replicas and constructs local graphs, as done in typical vertex-cut. One of the replicas is also randomly nominated as the master (by hashing), and the rest are mirrors.

As shown in Figure 7, assume that the user-defined threshold ($\theta$) is 3 and the hash function is $hash(X) = (X - 1)\%3$. All vertices along with their in-edges are assigned as low-degree vertices except vertex 1, whose in-edges are assigned as a high-degree vertex. For example, the edge $(1, 4)$ and $(3, 4)$ are placed in machine 0 with the master of low-degree vertex 4, while the edge $(2, 1)$ and $(5, 1)$ are placed in machine 1 with the mirror of high-degree vertex 1. The partition constructed by the hybrid-cut only yields four mirrors and achieves good load balance.

The hybrid-cut addresses the major issues in edge-cut and vertex-cut on skewed graphs. First, the hybrid-cut can provide a much lower replication factor. For low-degree vertices, all in-edges are grouped with their target vertices, and there is no need to create mirrors for them. For high-degree vertices, the upper bound of increased mirrors due to assigning a new high-degree vertex along with all its in-edges is equal to the number of partitions (i.e., machines) rather than the degree of vertex; this completely avoids new mirrors of low-degree vertices and restricts the bursting increase of replication factor. Second, the hybrid-cut provides unidirectional access locality for low-degree vertices, which can be used by hybrid computation model (§3) to reduce communication cost at runtime. Third, hybrid-cut is very efficient in distributed pre-processing, since it is a wholly hash-based partitioning scheme for both low-degree and high-degree vertices. All vertices and edges can be independently assigned to the designated machines without coordination. Finally, the partitions constructed by the hash-based (random) hybrid-cut are naturally balanced on both vertices and edges. The randomized placement of low-degree vertices leads to the balance of vertices, which is almost equivalent to the balance of edges for low-edge vertices. For high-degree vertices, all in-edges are assigned to the owner machine of source vertices, which are also assigned to random machines by hashing.

**Constructing hybrid-cut:** A natural approach to constructing a hybrid-cut is adding an extra reassignment phase for high-degree vertices to the original streaming graph partitioning. The left part of Figure 8 illustrates the execution flow of pre-processing using hybrid-cut, and the right part

```
HYBRID_INGRESS()
    . . .
 1  //Load & Dispatch
 2  while e = get_edge() do
 3    p = HASH(e.target)
 4    SEND(p, e)

 5  while e = RECV() do
 6    vset[e.target].add(e)

 7  foreach v in vset
 8    degree = v.size
 9    if (degree <= threshold)
10      foreach e in v.list
11        elist.add(e)

12  //Reassign (high-degree)
13  foreach v in vset
14    degree = v.size
15    if (degree > threshold)
16      foreach e in v.list
17        p = HASH(e.source)
18        SEND(p, e)

19  while e = RECV() do
20    elist.add(e)

21  //Construct
    . . .
```

Fig. 9. The pseudo-code of random hybrid-cut.

shows the results of sample graph in each stage. Figure 9 further shows the pseudo-code of the hash-based (random) hybrid-cut. First, the worker thread on each machine *loads* a piece of raw graph data (e.g., edge list) from the underlying distributed file system (e.g., HDFS) in parallel, and *dispatches* edges to machines by hashing their target vertices (Line 2-4). Each worker counts the in-degree of vertices (Line 5-6) and compares it with a user-defined threshold ($\theta$)[13] to identify high-degree vertices (Line 13-15). After that, in-edges of high-degree vertices are *reassigned* by hashing their source vertices (Line 16-18). Finally, each worker thread creates replicas to construct a local graph as normal vertex-cut.

This approach is compatible with existing formats of raw graph data, but incurs some network transmission cost due to reassigning in-edges of high-degree vertices. For some graph file format (e.g., adjacent list), the worker can directly identify high-degree vertices and distribute edges in the loading stage to avoid extra reassignment, since the in-degree and a list of all source vertices are grouped in one line.

## 4.2 Heuristic Hybrid-Cut

To further reduce the replication factor of low-degree vertices, we propose a new greedy heuristic algorithm, namely **Ginger**, inspired by Fennel [72], which is a greedy streaming edge-cut framework. Ginger places the next *low-degree* vertex along with in-edges on the machine that minimizes the expected replication factor.

Formally, given that the set of partitions for assigned low-degree vertices are $P = (S_1, S_2, \ldots, S_p)$, a low-degree vertex $v$ is assigned to partition $S_i$ such that $\delta g(v, S_i) \geq \delta g(v, S_j), for\ all\ j \in \{1, 2, \ldots, p\}$. We define the **score formula** $\delta g(v, S_i) = |N(v) \cap S_i| - \delta c(|S_i|^V)$, where $N(v)$ denotes the set of neighbors along in-edges of vertex $v$, and $|S_i^V|$ denotes the number of vertices in $S_i$. The former component $|N(v) \cap S_i|$ corresponds to the degree of vertex $v$ in the subgraph induced by $S_i$. The **balance formula** $\delta c(x)$ can be interpreted as the marginal cost of adding vertex $v$ to partition $S_i$, which is used to balance the size of partitions.

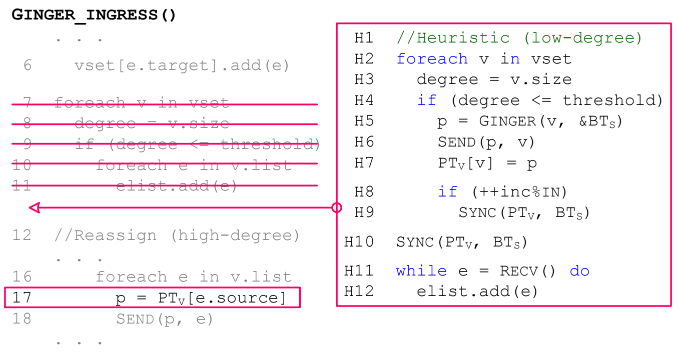
Considering the special requirements of hybrid-cut, Ginger differs from Fennel in several aspects to improve the performance and balance, as shown in Table 4. First, Fennel is inefficient to partition skewed graphs due to high-degree vertices. Hence, Ginger just uses this heuristic to improve the placement of low-degree vertices. Second, as Fennel is designed to minimize the fraction of edges being cut, it estimates all adjacent edges in *both* directions to determine the host machine. By contrast, Ginger only estimates edges in one direction to decrease the pre-processing time. Finally,

[13]A detailed discussion about the optimal threshold can be found in §6.6.

Fennel focuses only on the balance of vertices, by using the number of vertices $|S_i^V|$ as the only parameter of the balance formula ($\delta c(x)$). Consequently, it usually causes a significant imbalance of edges even for regular graphs due to the de-randomized placement of low-degree vertices. To improve the balance of edges, we add the normalized number of edges $\mu|S_i^E|$ into the parameter of the balance formula, where $\mu$ is the ratio of vertices to edges, and $|S_i^E|$ is the number of edges in $S_i$. The composite balance parameter becomes $(|S_i^V| + \mu|S_i^E|)/2$.

Table 4. A comparison of the heuristic in Fennel and Ginger.

| | Fennel | Ginger |
|---|---|---|
| What to partition? | Low & High | Low |
| How to partition? | In-edge & Out-edge | In-edge |
| How to balance? | Vertex | Vertex & Edge |



```
GINGER_INGRESS()
    . . .
6     vset[e.target].add(e)

~~7   foreach v in vset~~
~~8     degree = v.size~~
~~9     if (degree <= threshold)~~
~~10      foreach e in v.list~~
~~11        elist.add(e)~~

12  //Reassign (high-degree)
    . . .
16      foreach e in v.list
17        p = PT_V[e.source]
18        SEND(p, e)
    . . .

H1   //Heuristic (low-degree)
H2   foreach v in vset
H3     degree = v.size
H4     if (degree <= threshold)
H5       p = GINGER(v, &BT_S)
H6       SEND(p, v)
H7       PT_V[v] = p

H8       if (++inc%IN)
H9         SYNC(PT_V, BT_S)

H10  SYNC(PT_V, BT_S)

H11  while e = RECV() do
H12    elist.add(e)
```

Fig. 10. The pseudo-code of heuristic hybrid-cut.

**Constructing heuristic hybrid-cut:** Based on the randomized hybrid-cut (Figure 9), a natural approach to constructing a greedy hybrid-cut is adding an extra *heuristic* phase between the *dispatch* and *reassign* phase, which runs Ginger heuristic on all low-degree vertices (Line H1-H7). Similar to Coordinated vertex-cut [24], Ginger also requires coordination among machines (Line H8-H10). The distributed tables $PT_V$ and $BT_S$ are used to store the current vertex partitions $P = (S_1, S_2, \ldots, S_p)$ and the balance of $p$ partitions respectively, and will be periodically updated by all machines. Each machine also maintains a local cache to reduce communication at the expense of freshness of $P$. In the subsequent *reassigning* phase, instead of hashing, the table $PT_V$ is used to locate the target machine for in-edges of high-degree vertices (Line 17).

## 4.3 Theoretical Comparison

We perform a theoretical analysis to compare hybrid-cut with prior vertex-cut[14], which partially follows that of [24]. We suppose that vertex $v$ spans a set of machines $A(v)$, where $A(v) \subseteq \{1, 2, \ldots, p\}$ containing its adjacent edges. $|A(v)|$ is used to denote the expected number of replicas of vertex $v$. Therefore, by linearity of the expected replication factor for a graph partition, it is equal to the average of expected replication factor of all vertices (Equation 1).

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v\in V}|A(v)|\right] = \frac{1}{|V|}\sum_{v\in V}\mathbb{E}\left[|A(v)|\right] \tag{1}$$

The expected replication factor $\mathbb{E}[|A(v)|]$ can be computed by considering the process of randomly assigning a set of adjacent edges $S(v)$, which *may incur replicas*. Let the indicator $P_i$ denotes the event that vertex $v$ has at least one of $S(v)$ on machine $i$. The expectation $\mathbb{E}[P_i]$ is then:

$$\mathbb{E}[P_i] = 1 - \left(1 - \frac{1}{p}\right)^{D_s[v]}$$

[14]Since the results of greedy heuristics highly depend on the sequence of edges in raw graph data and their coordinated strategies among machines, we only consider random (hash-based) graph partitioning algorithms, which are the current preferential partitioning algorithm of most distributed graph-parallel systems, such as Giraph, PowerGraph and GraphX.

*where $D_s[v]$ denotes the number of edges in $S(v)$. For various vertex-cut algorithms, $S(v)$ denotes various candidate sets.* In the following equations, we relate the expected normalized replication factor of various vertex-cut algorithms to the number of machines $p$ and the power-law constant $\alpha$.

***Random vertex-cut.*** All edges of vertex $v$ are randomly assigned to $p$ machines, so the expected replication factor of Random vertex-cut [24] is:

$$\mathbb{E}\Big[|A(v)|\Big] = \sum_{i=1}^{p} \mathbb{E}[P_i] = p\left(1 - \left(1 - \frac{1}{p}\right)^{D[v]}\right) \tag{2}$$

*where $D[v]$ denotes the degree of vertex v, which is treated as a Zipf random variable:*

$$\mathbb{E}\Big[D[v]\Big] = \frac{\mathbf{h}_{|V|}(\alpha - 1)}{\mathbf{h}_{|V|}(\alpha)}$$

*where $\mathbf{h}_{|V|}(\alpha) = \sum_{d=1}^{|V|-1} d^{-\alpha}$ is the normalizing constant of the power-law Zipf distribution [24].*

***Grid vertex-cut.*** All edges of vertex $v$ are randomly assigned to a grid-based constrained set on $p$ machines then the expected replication factor of Grid vertex-cut [31] is:

$$\mathbb{E}\Big[|A(v)|\Big] = \mathbf{f}(p)\left(1 - \left(1 - \frac{1}{\mathbf{f}(p)}\right)^{D[v]}\right) \tag{3}$$

*where $\mathbf{f}(p)$ denotes the size of constrained set on p machines to host edges of vertex v, and equals $2\sqrt{p} - 1$ in Grid vertex-cut. In addition, the reduction of upper bound from p to $2\sqrt{p} - 1$ may incur load imbalance of both vertices and edges.*

***Random hybrid-cut.*** For *low-degree* vertex $v$, all of in-edges and the out-edges linked with high-degree vertices (i.e., high-degree edges) are assigned to the master of vertex $v$. Consequently, only the out-edges linked with low-degree vertices (i.e., low-degree edges) may incur replicas, then the expected replication factor on $p$ machines is:

$$\mathbb{E}\Big[|A(v)|\Big] = p\left(1 - \left(1 - \frac{1}{p}\right)^{D_{out}[v](1-P_{E_H})}\right) \tag{4}$$

*where $D_{out}[v]$ denotes the degree of out-edges of vertex v and $P_{E_H}$ denotes the percentage of high-degree edges where the target vertex has high-degree.*

For *high-degree* vertex $v$, all of in-edges and the out-edges linked with low-degree vertices may incur replicas. Therefore, the expected replication factor on $p$ machines is:

$$\mathbb{E}\Big[|A(v)|\Big] = p\left(1 - \left(1 - \frac{1}{p}\right)^{D_{in}[v]+D_{out}[v](1-P_{E_H})}\right) \tag{5}$$

*where $D_{in}[v]$ denotes the degree of in-edges of vertex v.*

Finally, the expected replication factor of Random hybrid-cut is:

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v\in V}|A(v)|\right] = (1 - P_{V_H})\frac{1}{|V_L|}\sum_{v\in V_L}\mathbb{E}[|A(v)|] \; + \; P_{V_H}\frac{1}{|V_H|}\sum_{v\in V_H}\mathbb{E}[|A(v)|] \tag{6}$$

*where $V_L$ and $V_H$ denotes the set of low-degree and high-degree vertices respectively, and $P_{V_H}$ denotes the percentage of high-degree vertices. For a* power-law *graph with constant $\alpha$ and threshold*

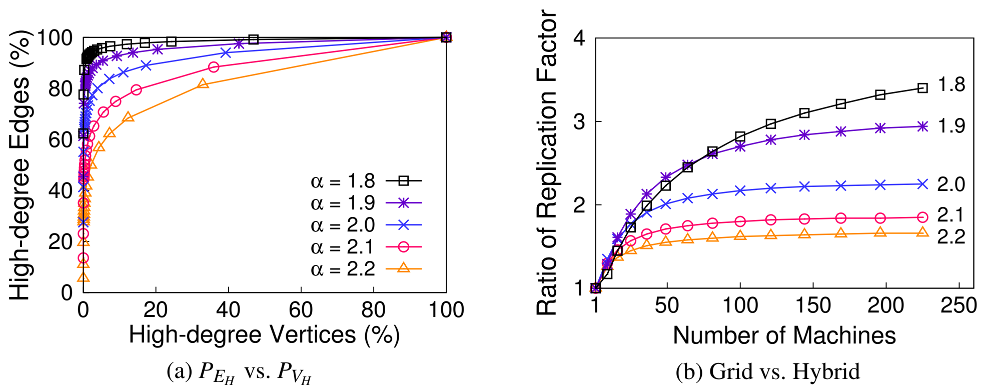


Fig. 11. (a) The relation between the percentage of high-degree vertices and edges for power-law graphs with different constants ($\alpha$). (b) The ratio of the expected replication factor of Grid vertex-cut to Random hybrid-cut with increasing machines.

*$\theta$, $P_{V_H}$ and $P_{E_H}$ are:*

$$P_{V_H} = 1 - \frac{\mathbf{h}_{|\theta|}(\alpha)}{\mathbf{h}_{|V|}(\alpha)}, \text{ and } P_{E_H} = 1 - \frac{\mathbf{h}_{|\theta|}(\alpha - 1)}{\mathbf{h}_{|V|}(\alpha - 1)}$$

For a skewed power-law graph, a small percentage of high-degree vertices ($P_{V_H}$) corresponds to a large percentage of high-degree edges ($P_{E_H}$). Figure 11(a) illustrates the relation between them for various power-law constants, where 1% vertices adjacent to more than 45% and 84% edges for $\alpha$=2.2 and 1.8 respectively. With the increase of $P_{V_H}$, the expected replication factor of both low-degree and high-degree vertex will rapidly decrease, while the percentage of high-degree vertices will also slowly increase. Hence, the expected replication factor of Random hybrid-cut for power-law graphs will first rapidly decrease dominated by low-degree vertices (Equation 4) and then slowly increase dominated by high-degree vertices (Equation 5).

Compared to Random vertex-cut (Equation 2), the expected replication factor of Random hybrid-cut (Equation 6) is always better, since the set of adjacent edges $S(v)$ of Random hybrid-cut that may incur replicas is merely a subset of that of Random vertex-cut. Compared to Grid vertex-cut, since the strategies adopted by Grid vertex-cut (Equation 3) and Random hybrid-cut are much different, it is rather difficult to directly compare the equation of expected replication factor. Therefore, we simulate the ratio of the expected replication factor of Grid vertex-cut to Random hybrid-cut using 10-million vertex power-law graphs with different constants. Random hybrid-cut uses a fixed threshold ($\theta$=100). As shown in Figure 11(b), Random hybrid-cut can outperform Grid vertex-cut in all cases, and the effective gains increase with lower power-law constant ($\alpha$). For example, the replication factor on 100 machines decreases from 5.76 to 3.55 and from 18.54 to 6.59 for $\alpha$=2.2 and 1.8 respectively.

## 4.4 Empirical Comparison

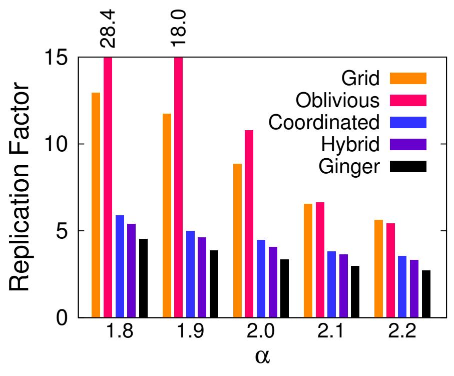
We use a collection of real-world and synthetic power-law graphs to compare various graph partitioning algorithms, as shown in Table 5. Most real-world graphs were from the Laboratory for Web Algorithmics[15] and Stanford Large Network Dataset Collection[16]. Each synthetic graph has 10 million vertices and a power-law constant ($\alpha$) ranging from 1.8 to 2.2. Smaller $\alpha$ produces denser

[15]LAW: http://law.di.unimi.it/datasets.php
[16]SNAP: http://snap.stanford.edu/data/

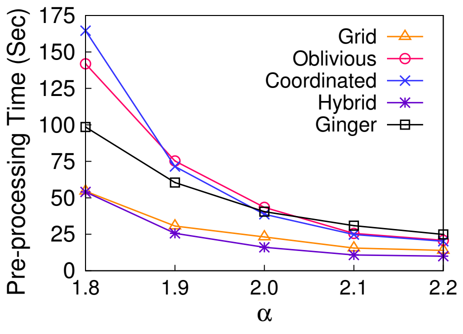
Table 5. A collection of real-world graphs and randomly constructed power-law graphs with varying $\alpha$ and fixed 10-million vertices. Smaller $\alpha$ produces denser graphs.

| **Real-world graphs** | $\|V\|$ | $\|E\|$ |
|---|---|---|
| Twitter (TW) [38] | 42M | 1.47B |
| UK-2005 (UK) [4] | 40M | 936M |
| Wiki (WK) [28] | 5.7M | 130M |
| LJournal (LJ) [18] | 5.4M | 79M |
| GWeb (GW) [42] | 0.9M | 5.1M |

| $\alpha$ | $\|V\|$ | $\|E\|$ |
|---|---|---|
| 1.8 | 10M | 673M |
| 1.9 | 10M | 249M |
| 2.0 | 10M | 105M |
| 2.1 | 10M | 53.8M |
| 2.2 | 10M | 39.0M |



(a) Replication Factor



(b) Pre-processing Time

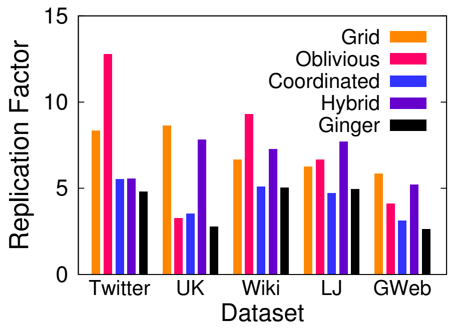
Fig. 12. The replication factor and pre-processing time of the power-law graphs with different constants ($\alpha$) on the 48-node 1GbE cluster.

graphs. They were generated by tools in PowerGraph, which randomly sample the in-degree of each vertex from a Zipf distribution [2] and then add in-edges such that the out-degrees of each vertex are nearly identical. Detailed testbed configurations can be found in (§6). Note that hybrid-cuts use a fixed threshold ($\theta$=100) and retain balanced load ($\rho \leq 1.01$)[17] for both edges and vertices in all cases.

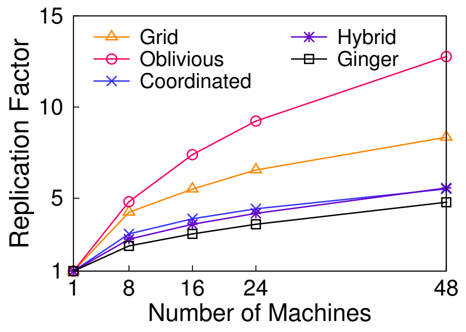
In Figure 12, we compare the replication factor and pre-processing time of hybrid-cut against various vertex-cuts for the power-law graphs with different constants ($\alpha$) on our 48-node cluster. Random hybrid-cut notably outperforms Grid vertex-cut with slightly less pre-processing time, and the gap increases with the growing of skewness of the graph, reaching up to 2.4X ($\alpha$=1.8). Oblivious vertex-cut has larger replication factor and more pre-processing time for the power-law graphs. Though Coordinated vertex-cut provides comparable replication factor with Random hybrid-cut (10% higher), it triples the pre-processing time. Ginger can further reduce the replication factor by more than 20% (see Figure 12(a)), but also increases pre-processing time like Coordinated vertex-cut. The overhead of Ginger is mainly from the exchange of mapping tables for low-degree vertices, which can be amortized with more edges in denser graphs (e.g., $\alpha$=1.8).

For real-world graphs (see Figure 13(a)), the improvement of Random hybrid-cut against Grid is smaller and sometimes slightly negative since the skewness of some graphs is moderate and randomized placement is not suitable for highly adjacent low-degree vertices (e.g., UK and GWeb). However, Ginger still performs much better in such cases, up to 3.11X improvement over Grid on

[17]Normalized maximum load $\rho = \frac{\textit{maximum load}}{\textit{average load}}$. Grid vertex-cut reaches up to 1.50 on 48 machines.

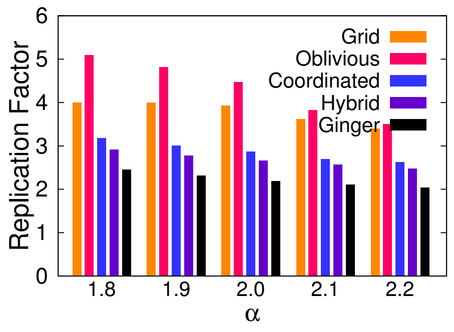


(a) Real-world Graphs

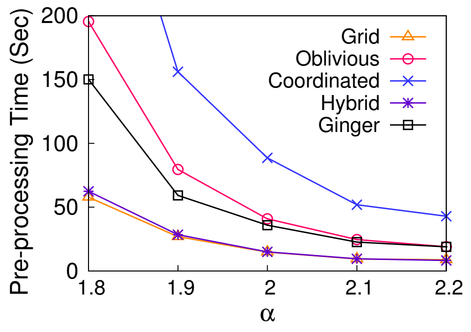


(b) Twitter Follower Graph

Fig. 13. The replication factor of the real-world graphs on the 48-node 1GbE cluster and the replication factor of the Twitter follower graph with increasing machines.



(a) Replication Factor



(b) Pre-processing Time

Fig. 14. The replication factor and pre-processing time of the power-law graphs with different constants ($\alpha$) on the 6-node 10GbE cluster.

UK. Figure 13(b) compares the replication factor with an increasing number of machines on the Twitter follower graph. Random hybrid-cut provides comparable results to Coordinated vertex-cut with just 35% pre-processing time, and outperforms Grid and Oblivious vertex-cut by 1.74X and 2.67X respectively.

To understand the influence of fewer machines and high-performance networking for the replication factor and pre-processing time on skewed graphs, we evaluate various graph partitioning algorithms for the power-law graphs with different constants ($\alpha$) on our 6-node cluster connected via 10Gb Ethernet. As shown in Figure 14(a), hybrid-cuts still outperform prior vertex-cuts even on fewer machines. For example, Random hybrid-cut and Ginger can reduce up to 32.2% (from 27.1%) and 24.2% (from 21.9%) replicas compared to Grid and Coordinated vertex-cut respectively. Figure 14(b) shows a similar trend of pre-processing time (see Figure 12(b)). Further, fewer machines will increase the time for graph loading due to less parallelism, while high-performance networking will decrease the time for dispatching edges. Since the main cost of greedy algorithms is from the heuristic edge-placement during graph loading, the pre-processing phase of them is relatively more efficient on large-scale clusters, especially for denser graphs (e.g., $\alpha$=1.8). Note that the parallel loading is disabled for Coordinated vertex-cut due to unknown bugs for 6 machines.

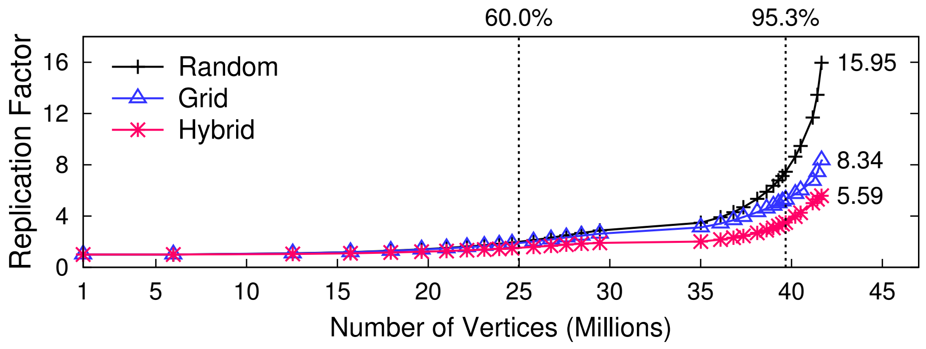


Fig. 15. The replication factor of the Twitter follower graph on 48 machines with the increase of vertices.

We further evaluate the effectiveness of various partitioning algorithms on low-degree and high-degree vertices. Figure 15 illustrates the growth of replication factor ($\lambda$) on 48 machines with the increase of vertices from the Twitter follower graph according to the order of their in-degrees. We set 100 as the threshold of Random hybrid-cut ($\theta$=100), and then the percentage of low-degree vertices is about 95.3% in the Twitter follower graph. For low-degree vertices, the replication factor of Random hybrid-cut is only 3.50, much lower than that of Random (7.46) and Grid (5.27) vertex-cut. The main reason is that there are few adjacencies among low-degree vertices in natural graphs, and assigning one vertex with in-edges to a single machine (i.e., hybrid-cut) introduces fewer replicas compared to assigning edges of one vertex to different machines (i.e., vertex-cut). For the first 60% vertices with the lowest degree, the difference of replication factor between Grid and Random vertex-cut is only less than 5% (1.85 vs. 1.93), because the upper bound of Grid (i.e., $2\sqrt{N}-1$) is still too large for most low-degree vertices. In contrast, the replication factor of Random hybrid-cut is 1.49 for these vertices.

For the 5% of vertices with the highest degree, the replication factor for Random vertex-cut increases dramatically from 7.46 to 15.96, since the edges are randomly assigned to machines, resulting in a large number of new replicas of its neighboring vertices (major low-degree vertices). Using Grid heuristics could relatively mitigate the increase of replication factor (from 5.27 to 8.34) through constraining the target machines. However, it still cannot avoid introducing new replicas of low-degree vertices. The replication factor of Random hybrid-cut is confined to 5.59 (from 3.50), thanks to assigning all edges of high-degree vertices to the master of their neighboring vertices.

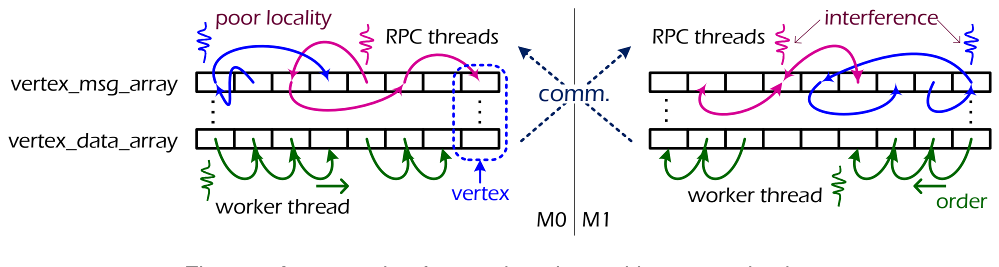


Fig. 16. An example of execution along with communication

## 5 LOCALITY-CONSCIOUS GRAPH LAYOUT

Graph computation usually exhibits poor data access locality [44, 50], due to irregular traversal of neighboring vertices along edges as well as frequent message exchanges between master and mirror vertices. The internal data structure of PowerLyra is organized to improve data access locality. Specifically, PowerLyra splits different (meta)data for both masters and mirrors to separated arrays and assigns unique local IDs sequentially in each machine to the local vertices for indexing, which are mapped to global vertex IDs. As shown in Figure 16, in each phase, the worker thread sequentially traverses vertices and executes user-defined functions. The messages across machines are batched and sent periodically.

After the synchronization in each phase, all messages received from different machines will be updated to vertices in parallel by multiple RPC threads, and the order of accessing vertices is only determined by order of the senders. However, since the order of messages is predefined by the traversal order, the accesses to vertices have poor locality due to a mismatch of orders between senders and receivers. Even worse, messages from multiple machines are processed in parallel and heavily interfere with each other (shown in the upper part of Figure 16). Though it appears that both problems could be addressed partially at runtime by sorting or dispatching messages on the fly [11], our experience shows that this will cause notable overhead instead of performance boost, due to non-trivial additional CPU cycles.

PowerLyra mitigates the above problems by extending hybrid-cut in four steps, as shown in Figure 17. The left part shows the arrangement of masters and mirrors in each machine after each step, and the right part provides a thumbnail with some hints about the ordering. Before the relocation, since the input graph files will be sequentially processed without additional sorting, all masters and mirrors of high-degree and low-degree vertices are mixed and stored in random orders. For example, the order of update messages from masters (i.e., 7, 1 and 4) in machine 0 (`M0`) mismatches the order of their mirrors stored in machine 1 (`M1`) (see Figure 17).

First, hybrid-cut divides the vertex space into 4 **zones** to store high-degree masters (`Z0`), low-degree masters (`Z1`), high-degree mirrors (`Z2`) and low-degree mirrors (`Z3`) respectively. This is friendly to the message batching since the processing on vertices in the same zone is similar. Further, it also improves the locality of worker threads by skipping the unnecessary vertex traversal and avoids interference between worker and RPC threads. For example, in the Apply phase, only masters (`Z0` and `Z1`) participate in the computation and only mirrors (`Z2` and `Z3`) receive messages.

Second, the mirrors in `Z2` and `Z3` are further **grouped** according to the location of their masters, which could further reduce the working set and the interference when multiple RPC threads update mirrors in parallel. For example, in `M1`, mirror 4 and 7 are grouped in `l0` while mirror 9 and 3 are grouped in `l2`. The processing of messages from the master of low-degree vertices in `M0` (`L0`) and `M2` (`L2`) is restricted to different groups (`l0` and `l2`) on `M1`.

Third, hybrid-cuts **sort** the masters and mirrors within a group according to the global vertex IDs and sequentially assign their local IDs. Because the order of messages follows the order of local IDs, sorting ensures that both masters and mirrors have the consistent relative order of local IDs to exploit spatial locality. For example, the low-degree master in `M0` (`L0`) and their mirrors in `M1` (`l0`) are sorted in the same order (i.e., 4 followed by 7). The message from `L0` in `M0` and `L2` in `M2` would be sequentially applied to mirrors (`l0` and `l2`) in `M1` in parallel.

Finally, since messages from different machines are processed simultaneously after synchronization, if the mirror groups in each machine have a similar order, it would lead to contention and interference on the master zones (`Z0` and `Z1`). For example, messages from mirrors in `M1` and `M2` (`h0` and `l0`) will be updated simultaneously to the master in `M0` (`H0` and `L0`). Therefore, hybrid-cuts place mirror groups in a **rolling** order: the mirror groups in machine *n* for *p* partitions start from

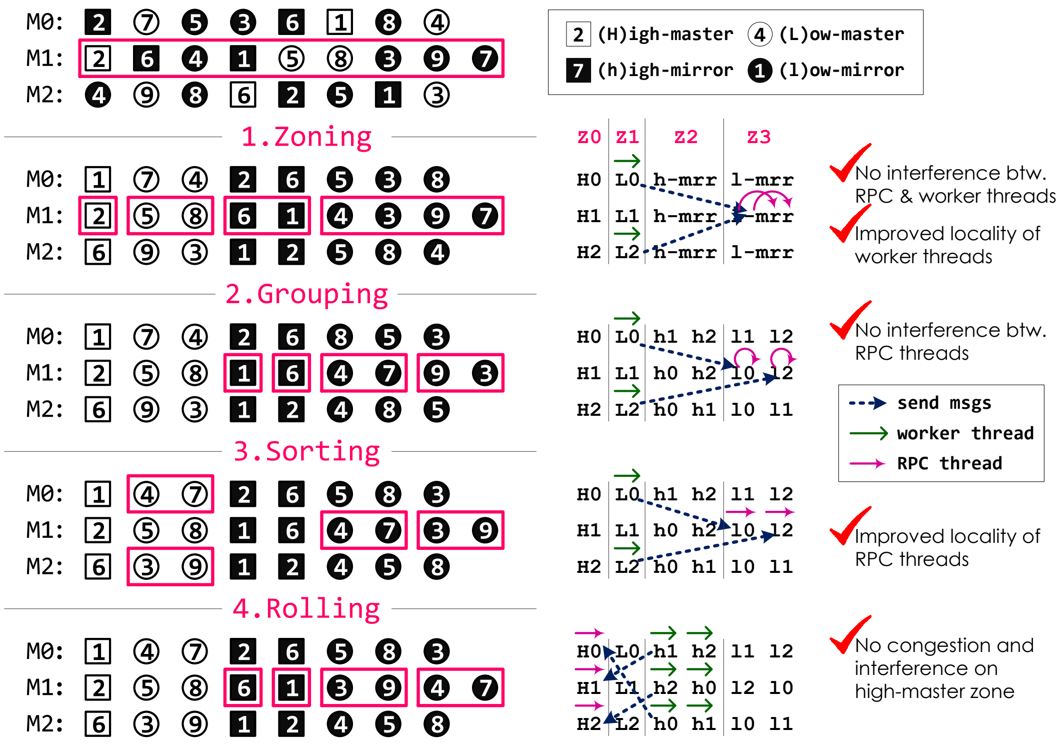

M0:
M1:
M2:
(H)igh-master
(L)ow-master
(h)igh-mirror
(l)ow-mirror
1.Zoning
2.Grouping
3.Sorting
4.Rolling
Z0 Z1 Z2 Z3
H0 L0 h-mrr l-mrr
H1 L1 h-mrr l-mrr
H2 L2 h-mrr l-mrr
No interference btw. RPC & worker threads
Improved locality of worker threads
No interference btw. RPC threads
send msgs
worker thread
RPC thread
Improved locality of RPC threads
No congestion and interference on high-master zone

Fig. 17. An example of the locality-conscious data layout optimization.

$(n+1) \bmod p$. For example, the mirror groups of high-degree vertices in `M1` start from `h2` then `h0`, where $n = 2$ and $p = 3$.

Though we separately describe above four steps, they are actually implemented as one step of hybrid-cuts after reassignment of high-degree vertices. All operations are executed *independently* on each machine, and there is no additional communication and synchronization. Hence, the increase of pre-processing time due to the above optimization is modest (less than 5% for the power-law graphs and around 10% for real-world graphs), resulting in usually more than 10% speedup (21% for the Twitter follower graph), as shown in Figure 18. The speedup for Google Web graph is negligible, as the number of vertices is very small. Since locality-conscious layout essentially trades off the pre-processing time for faster graph computation, it should be worthwhile for graph computation that processes a graph with multiple iterations and even multiple times in memory.

## 6 EVALUATION

We have implemented PowerLyra based on the latest GraphLab PowerGraph v2.2 (released in February 2015), and can seamlessly run all existing graph algorithms in GraphLab and respect the fault tolerance model. PowerLyra currently supports both synchronous and asynchronous execution modes. To illustrate the efficiency and generality of PowerLyra, we further port the Random hybrid-cut to GraphX.

We evaluate PowerLyra with hybrid-cut (Random and Ginger) against PowerGraph with vertex-cut (Grid, Oblivious and Coordinated), and report the average results of five runs for each experiment. Most experiments are performed on a dedicated, VM-based 48-node EC2-like cluster. Each

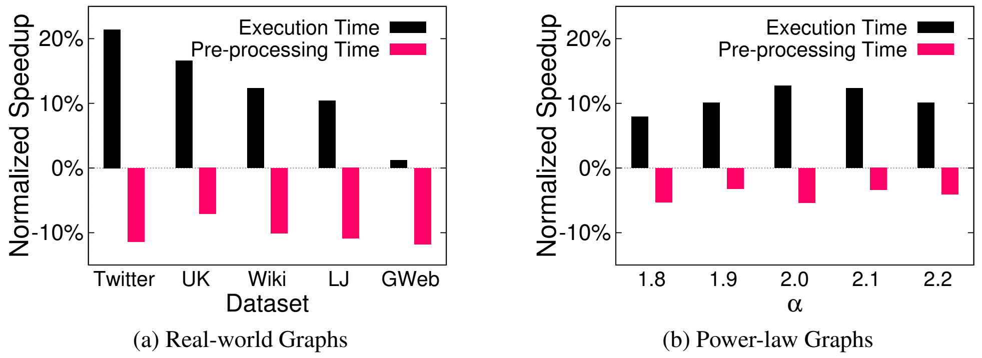


Fig. 18. The effect of locality-conscious optimization on the real-world and power-law graphs for PageRank using 48 machines.

machine has 4 AMD Opteron cores and 12GB DRAM. All machines are connected via 1Gb Ethernet. To avoid exhausting memory and slow convergence, a 6-node in-house physical 1GbE cluster with a total of 144 AMD Opteron cores and 384GB DRAM is used to evaluate the scalability in terms of data size (§6.3), the performance of the asynchronous engine (§6.9) and the comparison with other systems (§6.11). We further use a new cluster of 6 machines connected by 10Gb Ethernet to understand the influence of high-performance networking for distributed graph-parallel systems. Each machine has two 10-core Intel Xeon E5-2650 v3 processors and 64GB of DRAM. We use the graphs listed in Table 5 and set 100 and 20 as the default threshold of hybrid-cut during our evaluation for the 48-node and 6-node clusters respectively.

## 6.1 Graph Algorithms

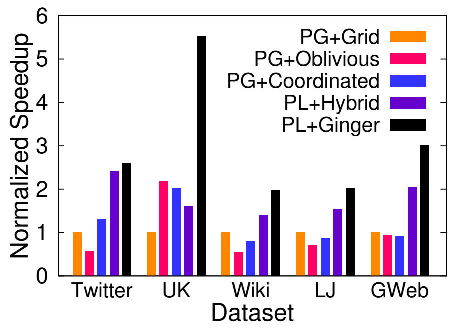
We choose three different typical graph-analytics algorithms representing three types of algorithms regarding the set of edges in the Gather and Scatter phases:

*PageRank (PR)* computes the rank of each vertex based on the ranks of its neighbors [5], which belongs to *Natural* algorithms that gather data along in-edges and scatter data along out-edges. PowerLyra should have significant speedup for both synchronous and asynchronous engines. Unless specified, the execution time of PageRank is the average of 10 iterations of synchronous execution.

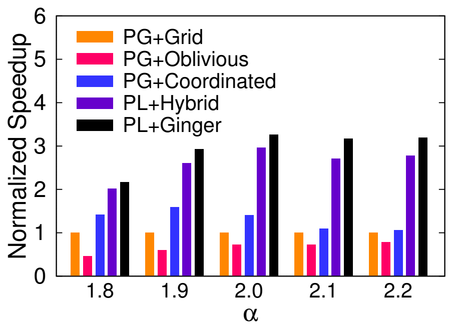
*Approximate Diameter (DIA)* estimates an approximation of diameter for a graph by probabilistic counting, which is the maximum length of shortest paths between each pair of vertices [35]. DIA belongs to the inverse *Natural* type of algorithms that gather data along out-edges and scatter none, which prefers the unidirectional access along out-edges. In such a case, PowerLyra is still expected to show notable improvements.

*Connected Components (CC)* calculates a maximal set of vertices that are reachable from each other by iterative label propagation. CC belongs to *Other* algorithms that gather none and scatter data via all edges. It benefits less from PowerLyra's computation model since the execution on PowerLyra is similar to that on PowerGraph. Fortunately, PowerLyra still outperforms PowerGraph due to hybrid-cut and locality-conscious data layout optimizations.

*Graph Coloring (GC)* assigns a color to each vertex and ensures that no adjacent vertices share the same color. The greedy implementation [23] simultaneously picks minimum colors not used by any of their neighbors for all vertices, which cannot converge in synchronous execution due to using the stale colors of neighbors. GC belongs to *Other* algorithms that gather and scatter data via



(a) Real-world Graphs



(b) Power-law Graphs

Fig. 19. Overall performance comparison between PowerLyra and PowerGraph on the real-world and power-law graphs for PageRank using 48 machines.

all edges. PowerLyra still outperforms PowerGraph due to hybrid-cut and lower scheduling cost of the asynchronous engine.

## 6.2 Performance

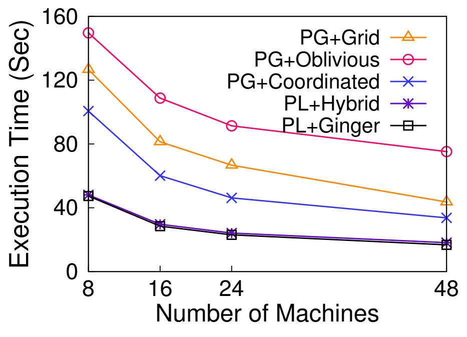
We compare the execution time of different systems and partitioning algorithms as in the PowerGraph paper [24]. Figure 19(a) shows the speedup of PowerLyra over PowerGraph on real-world graphs. The largest speedup comes from UK graph for the Ginger hybrid-cut due to a relatively high reduction of replication factor (from 8.62 to 2.77, Figure 13). In this case, PowerLyra using Ginger outperforms PowerGraph with Grid, Oblivious and Coordinated vertex-cut by 5.53X, 2.54X and 2.72X accordingly. For Twitter, PowerLyra also outperforms PowerGraph by 2.60X, 4.49X and 2.01X for Grid, Oblivious and Coordinated vertex-cut accordingly. Even though the replication factor of Wiki and LJournal using Random hybrid-cut is slightly higher than that of Grid and Coordinated, PowerLyra still outperforms PowerGraph using Grid by 1.40X and 1.73X for Wiki and 1.55X and 1.81X for LJournal accordingly, due to the better computing efficiency of low-degree vertices.

As shown in Figure 19(b), PowerLyra performs better for the power-law graphs using hybrid-cut, especially for high power-law constants (i.e., $\alpha$) due to the higher percentage of low-degree vertices. In all cases, PowerLyra outperforms PowerGraph with Grid vertex-cut by more than 2X, from 2.02X to 3.26X. Even compared with PowerGraph with Coordinated vertex-cut, PowerLyra still provides a speedup ranging from 1.42X to 2.63X. Though not clearly shown in Figure 19(b), PowerLyra with Ginger outperforms Random hybrid-cut from 7% to 17%. Such a relatively smaller speedup for the power-law graphs is because Random hybrid-cut already has a balanced partition with a small replication factor (see Figure 12).

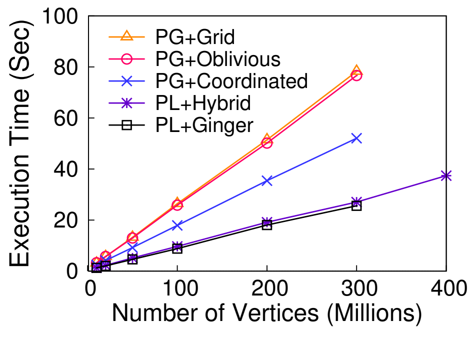
## 6.3 Scalability

We study the scalability of PowerLyra in two aspects. First, we evaluate the performance for a given graph (Twitter follower graph) with the increase of resources. Second, we fix the resources using the 6-node cluster while increasing the size of the graph.

Figure 20 shows that PowerLyra has similar scalability with PowerGraph, and keeps the improvement with increasing machines and data size. With the increase of machines from 8 to 48, the speedup of PowerLyra using Random hybrid-cut over PowerGraph with Grid, Oblivious and Coordinated vertex-cut ranges from 2.41X to 2.76X, 2.14X to 3.78X and 1.86X to 2.09X. For the

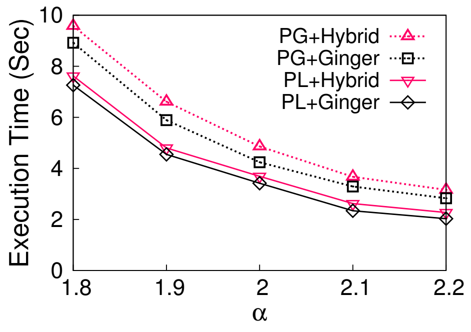


(a) #Machines

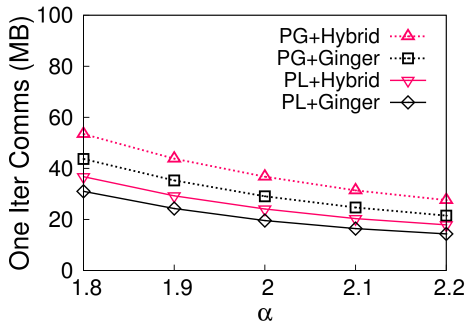


(b) Graph Size

Fig. 20. A comparison between PowerLyra and PowerGraph for the Twitter follower graph with increasing machines and for the power-law graph ($\alpha$=2.2) on the 6-node cluster with increasing data size.



(a) Execution Time



(b) Communication Data Size

Fig. 21. A comparison between PowerLyra and PowerGraph for PageRank using the power-law graphs with different constants ($\alpha$) on 48 machines. Both PowerLyra and PowerGraph use Hybrid and Ginger hybrid-cut.

increase of graph from 10 to 400 million vertices with fixed power-law constant 2.2, PowerLyra with Random hybrid-cut stably outperforms PowerGraph with Grid, Oblivious and Coordinated vertex-cut by up to 2.89X, 2.83X and 1.94X respectively. Note that only PowerLyra with Random hybrid-cut can handle the graph with 400 million vertices due to the reduction of memory in graph computation and partitioning.

### 6.4 Effectiveness of Graph Engine

Hybrid-cut can also be applied to the original PowerGraph engine, which we use to quantify the performance benefit from the hybrid computation model, we run both PowerGraph and PowerLyra engine using the same hybrid-cut on 48 machines for the power-law graphs. As shown in Figure 21(a), PowerLyra outperforms PowerGraph by up to 1.40X and 1.41X using Random and Ginger hybrid-cut respectively, due to the elimination of more than 30% communication cost (see Figure 21(b)).

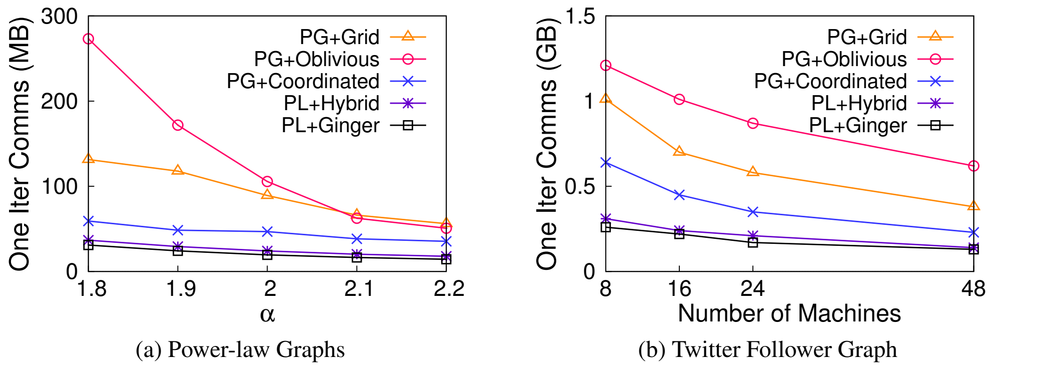


Fig. 22. One iteration communication data size for the power-law graphs with different constants ($\alpha$) on 48 machines and for the Twitter follower graph with increasing machines.

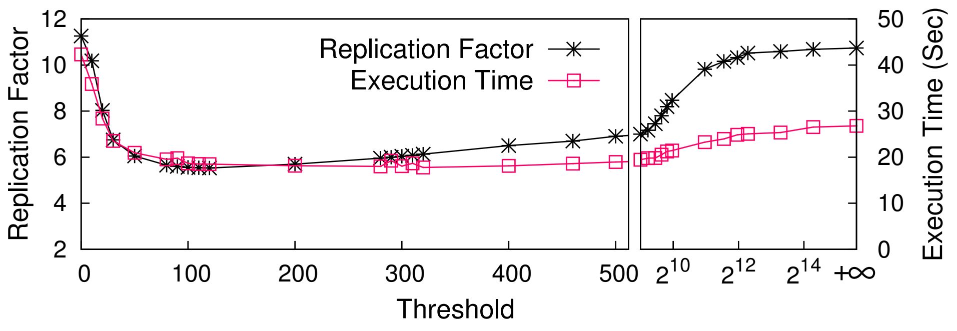


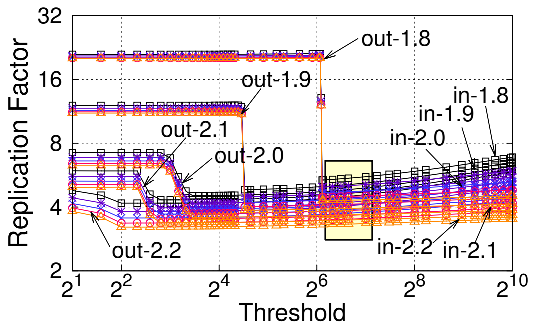
Fig. 23. The impact of threshold in PowerLyra on replication factor and execution time for the Twitter follower graph using PageRank.

## 6.5 Communication Cost

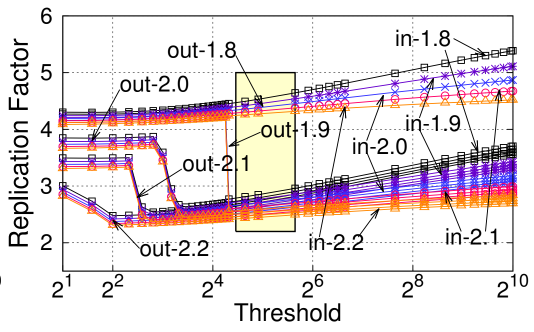
The improvement of PowerLyra is mainly from reducing communication cost. In PowerLyra, only high-degree vertices (a small fraction) require significant communication cost, while low-degree vertices (a large fraction) only require one message exchange in each iteration. As shown in Figure 22, PowerLyra has much less communication cost compared to PowerGraph. For the power-law graphs, PowerLyra can reduce data transmitted by up to 75% and 50% using Random hybrid-cut, and up to 79% and 60% using Ginger, compared to PowerGraph with Grid and Coordinated vertex-cut respectively. PowerLyra also significantly reduces the communication cost for the Twitter follower graph up to 69% and 52% using Random hybrid-cut, and up to 74% 59% using Ginger, compared to PowerGraph with Grid and Coordinated vertex-cut respectively.

## 6.6 Threshold

To study the impact of different thresholds, we run PageRank on the Twitter follower graph with different thresholds. As shown in Figure 23, using high-cut ($\theta$=0) or low-cut ($\theta$=+∞) for all vertices results in poor replication factor due to the negative impact from skewed vertices in terms of out-edge or in-edge. With an increasing threshold, the replication factor ($\lambda$) rapidly decreases and then



(a) 48 partitions



(b) 6 partitions

Fig. 24. The replication factor of random hybrid-cut with the increase of thresholds for 48 and 6 partitions using various power-law graphs. Each graph is combined by two synthetic power-law graphs with skewed in and out degree distributions ($\alpha$=1.8-2.2) separately. The yellow box marks the range of thresholds for acceptable replication factor (less than 10% overhead).

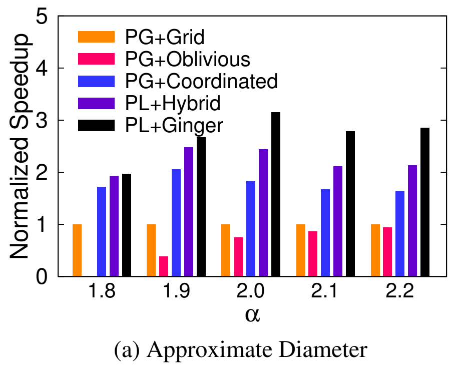
slowly increases. The best runtime performance usually occurs a little bit after the lowest replication factor, since the increase of threshold also reduces the number of high-degree vertices, which is beneficial to the overall performance. Consequently, the execution time is relatively stable for a large range of thresholds. In Figure 23, the difference of execution time under threshold from 100 to 500 is lower than 1 second.

Therefore, an intuitive way to find the threshold ($\theta$) of optimal performance is approximately equal to find the threshold of the lowest replication factor. However, the replication factor is still determined by multiple factors other than the threshold, including the number of machines ($p$) and the skewness of input graphs ($\alpha$). Fortunately, we observe that given the number of machines, there exists a range of thresholds can provide acceptable results (less than 10% overhead) for various power-law graphs. In Figure 24, we measure the replication factor of random hybrid-cut to partition a large number of typical power-law graphs into 48 and 6 partitions. Since the power-law graphs generated by tools in PowerGraph have only one direction skewed degree distribution, we combine two synthetic graphs skewed in different directions to represent real-world graphs. For example, the power-law constant ($\alpha$) of in and out degree distributions for the Twitter follower graph is approximate 1.7 and 2.0 respectively [24]. As shown in Figure 24, the range of thresholds to provide less than 10% overhead of replication factor for all graphs is from 70 to 140 and from 20 to 50 for 48 and 6 partitions separately. Therefore, users can determine a reasonable threshold for a given cluster by offline sampling several typical synthetic graphs.

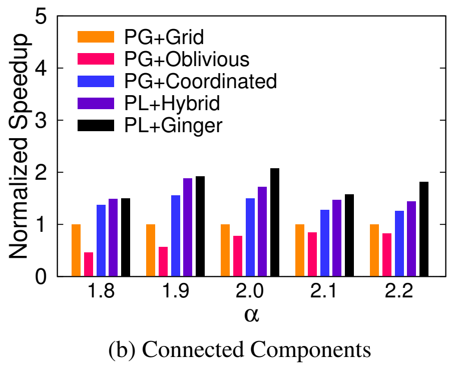
## 6.7 Other Algorithms and Graphs

To study the performance of PowerLyra on different algorithms, we evaluate DIA and CC on the power-law graphs. As shown in Figure 25(a), PowerLyra outperforms PowerGraph with Grid vertex-cut by up to 2.48X and 3.15X using Random and Ginger hybrid-cut respectively for DIA. Even compared with PowerGraph with Coordinated vertex-cut, the speedup still reaches 1.33X and 1.74X for Random and Ginger hybrid-cut. Note that the missing data for PowerGraph with Oblivious is because of exhausted memory.

Since PowerLyra treats execution in the Scatter phase of low-degree vertices the same as high-degree vertices, the improvement on CC is mainly from hybrid-cut, which reduces the communication cost by decreasing replication factor. For the power-law graphs, PowerLyra can still outperform



(a) Approximate Diameter



(b) Connected Components

Fig. 25. A comparison between PowerLyra and PowerGraph on the power-law graphs for Approximate Diameter (DIA) and Connected Components (CC) on 48 machines.

Table 6. A comparison of various graph partitioning algorithms on 48 machines using PageRank (10 iterations) for the RoadUS graph.

| Algorithm & Graph | Vertex-cut | $\lambda$ | Time (Sec) Pre-processing | Time (Sec) Execution |
|---|---|---|---|---|
| PageRank & RoadUS [20] $\|V\|$=23.9M $\|E\|$=58.3M | Coordinated | **2.28** | 26.9 | 50.4 |
| | Oblivious | **2.29** | **13.8** | 51.8 |
| | Grid | 3.16 | 15.5 | 57.3 |
| | Hybrid | 3.31 | **14.0** | **32.2** |
| | Ginger | 2.77 | 28.8 | **31.3** |

PowerGraph with Grid vertex-cut by up to 1.88X and 2.07X using Random and Ginger hybrid-cut respectively (see Figure 25(b)).

We also investigate the performance of PowerLyra for non-skewed graphs like road networks. Table 6 illustrates a performance comparison between PowerLyra and PowerGraph for PageRank with 10 iterations on RoadUS [20], the road network of the United States. The average degree of RoadUS is less than 2.5 (there are no high-degree vertices). Even though Oblivious and Coordinated vertex-cut have lower replication factor due to the greedy heuristic, PowerLyra with hybrid-cut still notably outperforms PowerGraph with vertex-cut by up to 1.78X, thanks to improved computation locality of low-degree vertices.

## 6.8 MLDM Applications

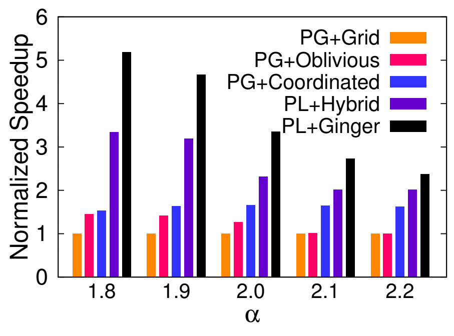
We further evaluate PowerGraph and PowerLyra on machine learning and data mining applications. Two different collaborative filtering algorithms, Alternating Least Squares (ALS) [99] and Stochastic Gradient Descent (SGD) [69], are used to predict the movie ratings for each user on Netflix movie recommendation dataset [99], in which the users and movies are presented as vertices, and the ratings are presented as edges. Both the memory consumption and computational cost depend on the magnitude of latent dimension ($d$), which also impacts the quality of approximation. The higher $d$ produces the higher accuracy of prediction while increasing both memory consumption and computational cost. As shown in Table 7, with the increase of latent dimension ($d$), the speedup of PowerLyra using Random hybrid-cut over PowerGraph with Grid vertex-cut ranges from 1.45X

Table 7. Performance (pre-processing / execution time in seconds) comparison between PowerLyra and PowerGraph on Netflix movie recommendation dataset using collaborative filtering algorithms. The vertex and edge data are measured in bytes and the $d$ is the size of the latent dimension.

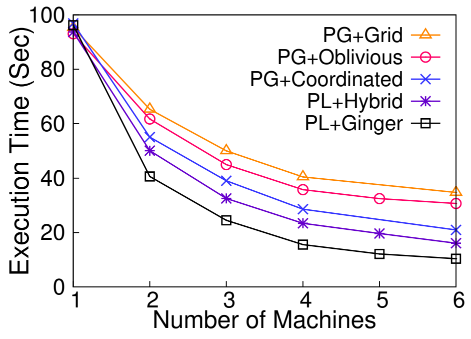
| **Netflix Movie Recommendation** [99] | | | | Replication Factor | |
|---|---|---|---|---|---|
| $\|V\|$ | $\|E\|$ | Vertex Data | Edge Data | Grid | Hybrid |
| 0.5M | 99M | $8d + 13$ | 16 | 12.3 | 2.6 |

| **ALS** [99] | $d$=5 | $d$=20 | $d$=50 | $d$=100 |
|---|---|---|---|---|
| PowerGraph w/ Grid | **10** / 33 | **11** / 144 | 16 / 732 | Failed |
| PowerLyra w/ Hybrid | 13 / **23** | 13 / **51** | **14 / 177** | **15 / 614** |

| **SGD** [69] | $d$=5 | $d$=20 | $d$=50 | $d$=100 |
|---|---|---|---|---|
| PowerGraph w/ Grid | **15** / 35 | **17** / 48 | 21 / 73 | 28 / 115 |
| PowerLyra w/ Hybrid | 16 / **26** | 19 / **33** | **19 / 43** | **20 / 59** |



(a) Graph Coloring



(b) Power-law Graph ($\alpha$=2.0)

Fig. 26. A performance and scalability comparison between PowerLyra and PowerGraph using asynchronous engine for Graph Coloring (GC) on the power-law graphs with different constants ($\alpha$) and increasing machines.

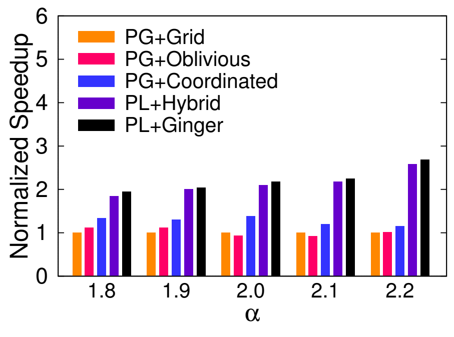
to 4.13X and 1.33X to 1.96X for ALS and SGD accordingly. Note that PowerGraph fails for ALS using $d$=100 due to exhausted memory as well.

## 6.9 Asynchronous Engine (Async)

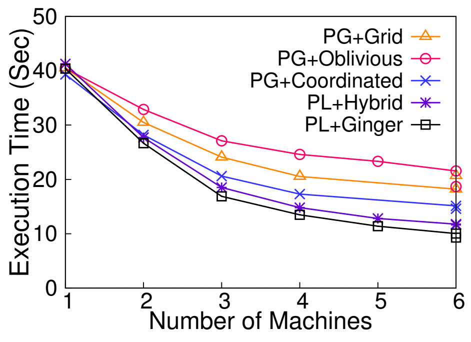
To study the performance improvement of PowerLyra compared to PowerGraph with the asynchronous engine, we run Graph Coloring on our 6-node cluster until convergence, which is hard or impossible in synchronous execution.

Figure 26(a) shows the speedup of PowerLyra over PowerGraph on power-law graphs. In all cases, PowerLyra outperforms PowerGraph with Grid vertex-cut by more than 2X (from 2.01X to 3.34X) due to lower replication factor. Even compared with PowerGraph with Coordinated vertex-cut, PowerLyra can still provide a speedup ranging from 1.22X to 2.18X due to good locality. In addition, PowerLyra with Ginger outperforms Random hybrid-cut from 17% to 55% due to a relatively significant improvement of the replication factor on a smaller cluster (6 nodes), reducing more than 16% replicas.

We also evaluate the performance for a given power-law graph ($\alpha$=2.0) with increasing machines. As shown in Figure 26(b), PowerLyra with hybrid-cuts has a better performance trend compared



(a) PageRank (Sync)



(b) Graph Coloring (Async)

Fig. 27. A performance and scalability comparison between PowerLyra and PowerGraph for PageRank on the power-law graphs with different constants ($\alpha$) using synchronous engine and Graph Coloring on the power-law graph ($\alpha$=2.0) with increasing machines using asynchronous engine.

to PowerGraph with vertex-cuts. Note that Grid vertex-cut performs relatively poor for a smaller cluster due to the upper bound of replication factor is too large (e.g., 3 for 3 machines)[18].

## 6.10 High-performance Networking

To understand the influence of high-performance networking for distributed graph-parallel systems, we evaluate PowerGraph and PowerLyra on a 6-node cluster with a total of 120 cores and 384GB DRAM and connected with 10GbE Infiniband NICs.

Figure 27(a) shows the speedup of PowerLyra over PowerGraph with the synchronous engine on power-law graphs for PageRank. Though the high-bandwidth networking mitigates the communication overhead, PowerLyra still can benefit from less computation cost due to lower replication factor and good locality in both computation and communication. PowerLyra with Random hybrid-cut can outperform PowerGraph with Grid and Coordinated vertex-cut up to 2.58X (from 1.84X) and 2.26X (from 1.37X) respectively. The performance difference between Random and Ginger hybrid-cut merely ranges from 3% to 6%, since the improvement on the replication factor by the greedy heuristic is quite limited on 6 machines. We believe with a larger scale cluster, PowerLyra would have a much larger performance speedup.

We also evaluate the performance of asynchronous engine for Graph Coloring with increasing machines using a power-law graph ($\alpha$=2.0). As shown in Figure 27(b), the overall performance of PowerLyra and PowerGraph are improved due to efficient CPU and networking (see Figure 26(b)). For example, the execution time of Grid vertex-cut on 6 machines decreases from 34.7s to 13.3s. Though the profit from high-performance networking for PowerLyra is relatively smaller than that for PowerGraph because of fewer communication cost (§6.5), PowerLyra with hybrid-cuts can still reserve better performance and scalability.

## 6.11 Comparison with Other Systems

Readers might be interested in how the performance of PowerLyra compares to other graph processing systems, even if they adopt different designs such as graph-parallel abstraction [24, 29, 37, 47, 60], dataflow operators [26], sparse matrix operations [7] or declarative programming [62]. We

[18]The missing data of Grid vertex-cut on 5 machines is because of the prerequisite in PowerGraph, that Grid vertex-cut can only be used to the number of partitions closed to a perfect square number. Further, using Coordinated vertex-cut on 5 machines is also failed due to unknown bugs when pre-processing.

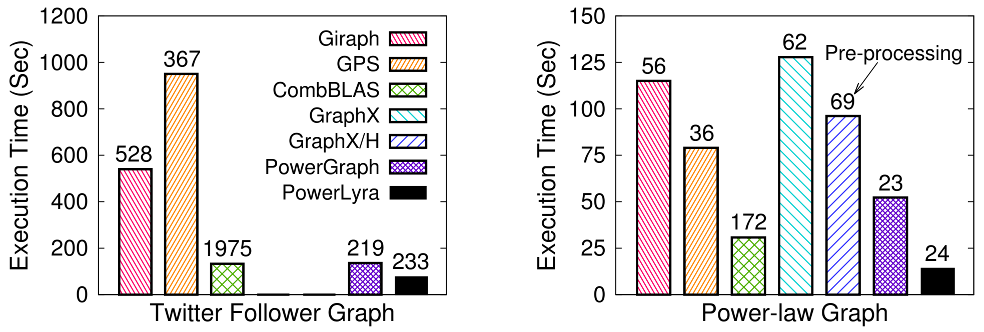


Fig. 28. Performance comparison for various systems on the Twitter follower and power-law ($\alpha$=2.0) graphs using PageRank. GraphX/H indicates GraphX with random hybrid-cut. The labels upon histogram are pre-processing time.

Table 8. Performance (in seconds) comparison with Polymer (PO), Galois (GA), X-Stream (XS) and GraphChi (GC) on PageRank (10 iterations) for both in-memory and out-of-core graphs using one machine of our 6-node cluster. PL/N indicates PowerLyra running on N machines.

| Graph | PL/6 | PL/1 | PO | GA | XS | GC |
|---|---|---|---|---|---|---|
| $\alpha$=2.0 $\lvert V\rvert$=10M | 14 | 45 | 6.3 | 9.8 | 9.0 | 115 |
| $\alpha$=2.2 $\lvert V\rvert$=400M | 186 | – | – | – | 710 | 1666 |

evaluate PageRank on such systems to provide an end-to-end performance comparison, as the implementation of PageRank is almost identical and well-studied on different systems. We deployed the latest Giraph 1.1, GPS, CombBLAS 1.4 and GraphX 1.1[19] on our 6-node cluster[20].

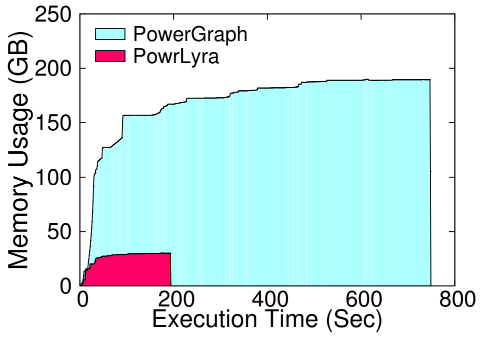
Figure 28 shows the execution time of PageRank with 10 iterations on each system for the Twitter follower graph and the power-law graph with 10 million vertices. The pre-processing time is also labeled upon histogram separately. PowerLyra outperforms other systems by up to 9.01X (from 1.73X), due to less communication cost and improved locality from differentiated computation and partitioning. Though CombBLAS has closest runtime performance (around 50% slower), its pre-processing stage takes a very long time (1,975 and 172 seconds) for data transformation due to the limitation of the programming paradigm (sparse matrix), reaching 1,975 and 172 seconds for Twitter and power-law graph respectively. We further port the Random hybrid-cut to GraphX (i.e., GraphX/H), leading to a 1.33X speedup even without heuristics[21] and differentiated computation engines. Compared to default 2D partitioning in GraphX, random hybrid-cut can reduce vertex replication by 35.3% and data transmitted by 25.7% for the power-law graph.

We further change the comparison targets to systems on single-machine platform. First, we compare the performance using a simple graph-analytics application (PageRank). One machine of our 6-node cluster (24 cores and 64GB DRAM) is used to run in-memory (Polymer [92] and Galois [53]) and out-of-core (X-Stream[22] [57] and GraphChi [39]) systems for both in-memory and out-of-core

[19]The source code of LFGraph [29] is not available, and Naiad [51] only provides a C# version. SociaLite [62] and Mizan [37] have some bugs to run on our clusters, which cannot be fixed in time by their authors.
[20]Both Giraph and GraphX ran out of memory on our 48-node cluster for the Twitter follower graph. Missed data for GraphX and GraphX/H on the Twitter follower graph is because of exhausted memory.
[21]We only implement Random hybrid-cut on GraphX for preserving its graph partitioning interface.
[22]X-Stream provides both in-memory and out-of-core engines. We use the latest release from authors, which can disable direct I/O and sufficiently leverage page cache.

Table 9. Performance (in seconds) comparison between PowerLyra (PL) and X-Stream (XS) using Alternating Least Squares (ALS) on Netflix movie recommendation dataset. X-Stream runs on an AWS EC2 r3.8xlarge instance (32 vCPUs and 244GB DRAM). PowerLyra runs on two 6-node in-house clusters with 1GbE and 10GbE networking respectively. The $d$ is the size of the latent dimension.

| **ALS** [99] | $d$=5 | $d$=20 | $d$=50 | $d$=100 |
|---|---|---|---|---|
| X-Stream (in-memory/244GB) | 10.3 | 30.8 | 130 | 318 |
| PowerLyra w/ Hybrid + 1GbE | 17.5 | 31.6 | 93 | 291 |
| PowerLyra w/ Hybrid + 10GbE | 9.2 | 15.8 | 46 | 145 |



|  | **GraphX** | **GraphX/H** |
|---|---|---|
| Memory Usage (GB) | | |
| RDD | 30.7 | 25.4 |
| Garbage Collection | | |
| Number | 33 | 18 |
| Time (Sec) | 17.6 | 10.0 |

Fig. 29. (a) Comparison of memory footprint between PowerLyra and PowerGraph for Netflix movie recommendation graph using ALS ($d$=50) on the 48-node cluster. (b) The memory and GC behavior of GraphX w/ and w/o hybrid-cut on the power-law graph ($\alpha$=2.0) for PageRank using the 6-node cluster

graphs using PageRank with 10 iterations. As shown in Table 8, PowerLyra performs comparably to Polymer and Galois for the 10-million vertex graph, while significantly outperforming X-Stream and GraphChi for the 400-million vertex graph. Considering six times resources used by PowerLyra, single-machine systems would be more economical for graphs that can fit within the memory of a single machine, while distributed ones are more efficient for larger graphs that cannot fit in the memory of a single machine. The current PowerLyra focuses on the distributed platform, resulting in a relatively poor performance on a single machine (45s of PL/1). We believe that PowerLyra can further improve the performance for both in-memory and out-of-core graphs by adopting the novel techniques of single-machine systems, such as NUMA-aware accesses strategy [92, 100]. In addition, a recent system, namely Musketeer [22][23], can automatically choose a right execution engine depending on the properties of input data, such as the size of the graph. Finally, the prevalent of cloud computing drives an easy, cost efficient way to launching distributed systems (e.g., PowerLyra) over flexible and elastic computing resources, even crossing multiple clouds (e.g., JointCloud [76]).

Second, we compare the performance using a popular MLDM application (ALS). A high-performance instance of AWS EC2 (r3.8xlarge) with 244GB DRAM is used to conduct the in-memory execution of X-Stream. As shown in Table 9, compared to X-Stream, PowerLyra can provide a comparable performance on the 6-node cluster with 1GbE and up to 2.83X speedup (from 1.11X) with high-performance networking.

[23]Musketeer has announced to support PowerLyra.

## 6.12 Memory Consumption

Besides performance improvement, PowerLyra can also mitigate the memory pressure due to significantly fewer vertex replicas and messages. The overall effectiveness depends on the ratio of vertices to edges and the size of vertex data. As shown in the left part of Figure 29, both the size and the duration of memory consumption on PowerLyra is notably fewer than that on PowerGraph for ALS ($d$=50) with Netflix movie recommendation graph, reducing near 85% peak memory consumption (30GB vs. 189GB) and 75% elapsed time (194s vs. 749s). We also use *jstat*, a memory tool in JDK, to monitor the GC behavior of GraphX and GraphX/H. Integrating hybrid-cut to GraphX also reduces about 17% memory usage for RDD and causes fewer GC operations even on only 6 machines for PageRank with a power-law graph ($\alpha$=2.0). We believe the measured reduction of memory would be significantly larger if GraphX executes on a larger cluster or memory-intensive algorithms.

## 7 OTHER RELATED WORK

PowerLyra is inspired by and departs from prior graph-parallel systems [24, 26, 43, 47], but differs from them in adopting a novel differentiated graph computation and partitioning scheme for vertices with different degrees.

**Distributed graph processing systems:** LFGraph [29] proposes a publish-subscribe mechanism to reduce communication cost but restricts graph algorithms just to the one-way access. Mizan [37] leverages vertex migration for dynamic load balancing. Imitator [16, 79] reuses computational replication for fault tolerance in large-scale graph processing to provide low-overhead normal execution and fast crash recovery. Giraph++ [71] and Blogel [87] provide several algorithm-specific optimizations for graph traversal and aggregation applications relying on the graph-centric and block-centric models with partitioning information. PowerSwitch [85] embraces the best of both synchronous and asynchronous execution modes by adaptively switching graph computation between them. GPS [60] also features an optimization on skewed graphs by partitioning the adjacency lists of high-degree vertices across multiple machines, while it overlooks the locality of low-degree vertices and still uniformly processes all vertices. Besta and Hoefler [3] propose Atomic Active Messages that leverages hardware transactional memory (HTM) to accelerate irregular graph computation. Chaos [56] enables graph processing built on the aggregate secondary storage of a cluster, which is shown capable of processing trillion-edge graphs. There are also a few systems considering GPU [45, 80, 81], NUMA [100], RDMA [64, 83], stream processing [17, 55, 96], and temporal graphs [27].

**Single-machine graph processing systems:** There are also several efforts aiming at leveraging multicore platforms for graph processing [39, 46, 53, 57, 65, 66, 92, 101], which focus on such as improving out-of-core accesses [39], selecting appropriate execution modes [65], supporting sophisticated task scheduler [53], reducing random operations on edges [57], adopting NUMA-aware data layout and access strategy [92], leveraging fine-grained partitioning [101], saving memory consumption [66], and exploiting heterogeneous devices [46]. Malicevic et al.[48] provide an end-to-end study on existing multicore graph processing systems, including various data structures, preprocessing approaches, as well as optimizations to improve cache locality, synchronization, and NUMA-awareness. Such techniques should be useful to enhance the performance of PowerLyra on each machine in the cluster.

**Graph replication and partitioning:** Generally, prior graph partitioning approaches can be categorized into vertex-cut [24, 31] and edge-cut [49, 61, 68, 72] according to their partition mechanism. Several greedy heuristics [24] and 2D mechanisms [9, 31, 89] are proposed to reduce communication cost and partitioning time on skewed graphs. Surfer [15] exploits the underlying heterogeneity of a public cloud for graph partitioning to reduce communication cost. Cube [94] uses 3D graph partitioning by dividing and assigning vertex data to different machines for MLDM applications. HotGraph [97] extracts a backbone structure from original graph to remove the cross-partition bottleneck for asynchronous graph processing. LazyGraph [78] proposes a lazy data coherency approach to avoid frequent global synchronizations and communications among vertex replicas. However, most of them are degree-oblivious but focus on using a general propose for all vertices or edges. Degree-based hashing [86] also considers the vertex degree but still adopts a uniform partitioning strategy. Based on the skewed degree distribution of vertices, PowerLyra is built with a hybrid graph partitioning as well as a new heuristic that notably improves performance.

**Other graph processing systems:** Besides the vertex-centric model, various programming paradigms are extended to handle graph processing. SociaLite [62] stores the graph data in tables and abstracts graph algorithms as declarative rules on the tables by Datalog. CombBLAS [7] expresses the graph computation as operations on sparse matrices and vectors, resulting in efficient computation time but also lengthy pre-processing time for data transformation. It also uses 2D partitioning to distribute the matrix for load balance. Trinity [63] uses a distributed in-memory key-value table abstraction to support graph processing. Naiad [51] provides timely dataflow abstraction to support low-latency streaming and cyclic computations, which allows the efficient implementation of iterative graph processing. None of existing graph processing systems use differentiated computation and partitioning. Besides, PowerLyra is orthogonal to above techniques and can further improve the performance of these systems on skewed graphs.

## 8 CONCLUSION

This paper argued that the “one size fits all” design in existing graph-parallel systems may result in suboptimal performance and introduced PowerLyra, a new graph-parallel system. PowerLyra used a hybrid and adaptive design that differentiated the computation and partitioning on high-degree and low-degree vertices. Based on PowerLyra, we also design locality-conscious data layout optimization to improve locality during communication. Experimental results showed that PowerLyra improved over PowerGraph and other graph-parallel systems substantially, yet fully preserved the compatibility with various graph algorithms.

## ACKNOWLEDGMENTS

We thank Kaiyuan Zhang for evaluating graph-parallel systems on single machine platform and Di Xiao for porting Random hybrid-cut to GraphX, as well as members of our research group for their thoughts and comments on this work at various stages. This work was supported in part by the National Key Research & Development Program of China (No. 2016YFB1000104) and the National Natural Science Foundation of China (No. 61772335 and 61572314).

## REFERENCES

[1] Amine Abou-Rjeili and George Karypis. 2006. Multilevel Algorithms for Partitioning Power-law Graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE Computer Society, Washington, DC, USA, 124–124. http://dl.acm.org/citation.cfm?id=1898953.1899055

[2] Lada A Adamic and Bernardo A Huberman. 2002. ZipfâĂŹs law and the Internet. *Glottometrics* 3, 1 (2002), 143–150.

[3] Maciej Besta and Torsten Hoefler. 2015. Accelerating Irregular Computations with Hardware Transactional Memory and Active Messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'15)*. ACM, New York, NY, USA, 161–172. https://doi.org/10.1145/2749246.2749263

[4] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Softw. Pract. Exper.* 34, 8 (July 2004), 711–726. https://doi.org/10.1002/spe.587

[5] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. In *WWW*. 107–117.

[6] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*. 49–60.

[7] Aydin Buluc and John R Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov. 2011), 496–509. https://doi.org/10.1177/1094342011403516

[8] Umit Catalyurek and Cevdet Aykanat. 1996. Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication. In *Proceedings of the 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96)*. Springer-Verlag, London, UK, UK, 75–86. http://dl.acm.org/citation.cfm?id=646010.676990

[9] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. 2010. On Two-Dimensional Sparse Matrix Partitioning: Models, Methods, and a Recipe. *SIAM J. Sci. Comput.* 32, 2 (Feb. 2010), 656–683. https://doi.org/10.1137/080737770

[10] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. 2017. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *ACM Transactions on Storage* 13, 3, Article 25 (Sept. 2017), 30 pages. https://doi.org/10.1145/3129900

[11] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2014. Computation and Communication Efficient Graph Processing with Distributed Immutable View. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC'14)*. ACM, New York, NY, USA, 215–226. https://doi.org/10.1145/2600212.2600233

[12] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. ACM, New York, NY, USA, Article 1, 15 pages. https://doi.org/10.1145/2741948.2741970

[13] Rong Chen, Jiaxin Shi, Binyu Zang, and Haibing Guan. 2014. Bipartite-oriented Distributed Graph Partitioning for Big Learning. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys '14)*. ACM, New York, NY, USA, Article 14, 7 pages. https://doi.org/10.1145/2637166.2637236

[14] Rong Chen, Jia-Xin Shi, Hai-Bo Chen, and Bin-Yu Zang. 2015. Bipartite-oriented distributed graph partitioning for big learning. *Journal of Computer Science and Technology* 30, 1 (2015), 20–29.

[15] Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. 2012. Improving Large Graph Processing on Partitioned Graphs in the Cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC'12)*. ACM, New York, NY, USA, Article 3, 13 pages. https://doi.org/10.1145/2391229.2391232

[16] Rong Chen, Youyang Yao, Peng Wang, Kaiyuan Zhang, Zhaoguo Wang, Haibing Guan, Binyu Zang, and Haibo Chen. 2018. Replication-Based Fault-Tolerance for Large-Scale Graph Processing. *IEEE Transactions on Parallel and Distributed Systems* 29, 7 (2018), 1621–1635.

[17] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, USA, 85–98. https://doi.org/10.1145/2168836.2168846

[18] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On Compressing Social Networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'09)*. ACM, New York, NY, USA, 219–228. https://doi.org/10.1145/1557019.1557049

[19] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[20] DIMACS. 2006. The 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/challenge9/

[21] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On Power-law Relationships of the Internet Topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'99)*. ACM, New York, NY, USA, 251–262. https://doi.org/10.1145/316188.316229

[22] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: All for One, One for All in Data Processing Systems. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15)*. ACM, New York, NY, USA, Article 2, 16 pages. https://doi.org/10.1145/2741948.2741968

[23] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. 2011. Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS'11)*. 324–332.

[24] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on*

*Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883

[25] Joseph E. Gonzalez, Yucheng Low, Carlos Guestrin, and David O'Hallaron. 2009. Distributed Parallel Inference on Large Factor Graphs. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI'09)*. AUAI Press, Arlington, Virginia, United States, 203–212. http://dl.acm.org/citation.cfm?id=1795114.1795139

[26] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 599–613. http://dl.acm.org/citation.cfm?id=2685048.2685096

[27] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2592798.2592799

[28] Henry Haselgrove. 2010. Wikipedia page-to-page link database. http://haselgrove.id.au/wikipedia.htm

[29] Imranul Hoque and Indranil Gupta. 2013. LFGraph: Simple and Fast Distributed Graph Analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS'13)*. ACM, New York, NY, USA, Article 9, 17 pages. https://doi.org/10.1145/2524211.2524218

[30] Andy Huang and Wei Wu. 2014. Mining Ecommerce Graph Data with Apache Spark at Alibaba Taobao. Retrieved April 1, 2018 from https://databricks.com/blog/2014/08/14/mining-graph-data-with-spark-at-alibaba-taobao.html

[31] Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. 2013. GraphBuilder: Scalable Graph ETL Framework. In *Proceedings of the 1st International Workshop on Graph Data Management Experiences and Systems (GRADES'13)*. ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2484425.2484429

[32] Saehan Jo, Jaemin Yoo, and U Kang. 2018. Fast and Scalable Distributed Loopy Belief Propagation on Real-World Graphs. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining (WSDM '18)*. ACM, New York, NY, USA, 297–305. https://doi.org/10.1145/3159652.3159722

[33] Xiaoen Ju, Hani Jamjoom, and Kang G. Shin. 2017. Hieroglyph: Locally-Sufficient Graph Processing via Compute-Sync-Merge. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 1, Article 9 (June 2017), 25 pages. https://doi.org/10.1145/3084446

[34] Tim Kaler, William Hasenplaugh, Tao B. Schardl, and Charles E. Leiserson. 2014. Executing Dynamic Data-graph Computations Deterministically Using Chromatic Scheduling. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'14)*. ACM, New York, NY, USA, 154–165. https://doi.org/10.1145/2612669.2612673

[35] U. Kang, Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. 2011. HADI: Mining Radii of Large Graphs. *ACM Trans. Knowl. Discov. Data* 5, 2, Article 8 (Feb. 2011), 24 pages. https://doi.org/10.1145/1921632.1921634

[36] George Karypis and Vipin Kumar. 1999. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review* 41, 2 (1999), 278–300.

[37] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. ACM, New York, NY, USA, 169–182. https://doi.org/10.1145/2465351.2465369

[38] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[39] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884

[40] Michael LeBeane, Shuang Song, Reena Panda, Jee Ho Ryoo, and Lizy K John. 2015. Data partitioning strategies for graph workloads on heterogeneous clusters. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. IEEE, 1–12.

[41] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph Evolution: Densification and Shrinking Diameters. *ACM Trans. Knowl. Discov. Data* 1, 1, Article 2 (March 2007). https://doi.org/10.1145/1217299.1217301

[42] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[43] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8

(April 2012), 716–727. https://doi.org/10.14778/2212351.2212354

[44] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[45] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: efficient GPU-accelerated graph processing on a single machine with balanced replication. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. USENIX Association, 195–207.

[46] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*. ACM, New York, NY, USA, 527–543. https://doi.org/10.1145/3064176.3064191

[47] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[48] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. USENIX Association, 631–643.

[49] Daniel Margo and Margo Seltzer. 2015. A Scalable Distributed Graph Partitioner. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1478–1489. https://doi.org/10.14778/2824032.2824046

[50] Kameshwar Munagala and Abhiram Ranade. 1999. I/O-complexity of Graph Algorithms. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 687–694. http://dl.acm.org/citation.cfm?id=314500.314891

[51] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[52] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[53] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[54] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. 2009. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1426–1437. https://doi.org/10.14778/1687553.1687569

[55] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/2465351.2465353

[56] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, USA, 410–424. https://doi.org/10.1145/2815400.2815408

[57] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, USA, 472–488. https://doi.org/10.1145/2517349.2522740

[58] Alessandra Sala, Lili Cao, Christo Wilson, Robert Zablit, Haitao Zheng, and Ben Y. Zhao. 2010. Measurement-calibrated Graph Models for Social Network Experiments. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. ACM, New York, NY, USA, 861–870. https://doi.org/10.1145/1772690.1772778

[59] Alessandra Sala, Xiaohan Zhao, Christo Wilson, Haitao Zheng, and Ben Y. Zhao. 2011. Sharing Graphs Using Differentially Private Graph Models. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC'11)*. ACM, New York, NY, USA, 81–98. https://doi.org/10.1145/2068816.2068825

[60] Semih Salihoglu and Jennifer Widom. 2013. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM'13)*. ACM, New York, NY, USA, Article 22, 12 pages. https://doi.org/10.1145/2484838.2484843

[61] Kirk Schloegel, George Karypis, and Vipin Kumar. 2000. Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning (Distinguished Paper). In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing (Euro-Par'00)*. Springer-Verlag, London, UK, UK, 296–310. http://dl.acm.org/citation.cfm?id=646665.698944

[62] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed Socialite: A Datalog-based Language for Large-scale Graph Analysis. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1906–1917. https://doi.org/10.14778/2556549.2556572

[63] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, New York, NY, USA, 505–516. https://doi.org/10.1145/2463676.2467799

[64] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-based Distributed Graph Exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 317–332. http://dl.acm.org/citation.cfm?id=3026877.3026902

[65] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[66] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference*. IEEE, 403–412.

[67] Alexander Smola and Shravan Narayanamurthy. 2010. An Architecture for Parallel Topic Models. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 703–710. https://doi.org/10.14778/1920841.1920931

[68] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'12)*. ACM, New York, NY, USA, 1222–1230. https://doi.org/10.1145/2339530.2339722

[69] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. 2009. Scalable Collaborative Filtering Approaches for Large Recommender Systems. *J. Mach. Learn. Res.* 10 (June 2009), 623–656. http://dl.acm.org/citation.cfm?id=1577069.1577091

[70] Tencent. 2018. Design and Practice of the Anomaly Detection Framework for Billions of User in WeChat (in Chinese). https://cloud.tencent.com/developer/article/1028442

[71] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.* 7, 3 (Nov. 2013), 193–204. http://dl.acm.org/citation.cfm?id=2732232.2732238

[72] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM'14)*. ACM, New York, NY, USA, 333–342. https://doi.org/10.1145/2556195.2556213

[73] Alexander Ulanov, Manish Marwah, Mijung Kim, Roshan Dathathri, Carlos Zubieta, and Jun Li. 2017. Sandpiper: Scaling probabilistic inferencing to large scale graphical models. In *Proceedings of 2017 IEEE International Conference on Big Data*. IEEE, 383–388.

[74] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

[75] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. 2017. An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 493–504. https://doi.org/10.14778/3055540.3055543

[76] Huaimin Wang, Peichang Shi, and Yiming Zhang. 2017. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *Proceedings of the 37th International Conference on Distributed Computing Systems (ICDCS '17)*. IEEE, 1846–1855.

[77] Hao Wang, Jing Zhang, Da Zhang, Sarunya Pumma, and Wu-chun Feng. 2017. PaPar: A Parallel Data Partitioning Framework for Big Data Applications. In *Proceedings of 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. IEEE, 605–614.

[78] Lei Wang, Liangji Zhuang, Junhang Chen, Huimin Cui, Fang Lv, Ying Liu, and Xiaobing Feng. 2018. Lazygraph: Lazy Data Coherency for Replicas in Distributed Graph-parallel Computation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 276–289. https://doi.org/10.1145/3178487.3178508

[79] Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. 2014. Replication-Based Fault-Tolerance for Large-Scale Graph Processing. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*. IEEE Computer Society, Washington, DC, USA, 562–573. https://doi.org/10.1109/DSN.2014.58

[80] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. 2018. Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration. In *Proceedings of 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 651–664.

[81] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. ACM, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

[82] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. 2009. User Interactions in Social Networks and Their Implications. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*. ACM, New York, NY, USA, 205–218. https://doi.org/10.1145/1519065.1519089

[83] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: Scaling Graph Computation to the Trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 408–421. https://doi.org/10.1145/2806777.2806849

[84] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. Tux2: Distributed Graph Computation for Machine Learning. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. 669–682.

[85] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. ACM, New York, NY, USA, 194–204. https://doi.org/10.1145/2688500.2688508

[86] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis. In *Proceedings of the 28th Annual Conference on Neural Information Processing Systems (NIPS'14)*. 1673–1681.

[87] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[88] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. 2009. Stochastic Gradient Boosted Distributed Decision Trees. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM'09)*. ACM, New York, NY, USA, 2061–2064. https://doi.org/10.1145/1645953.1646301

[89] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. 2005. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC'05)*. IEEE Computer Society, Washington, DC, USA, 25–. https://doi.org/10.1109/SC.2005.4

[90] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[91] Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Santa Clara, CA, 167–180. https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-heng

[92] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware Graph-structured Analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. ACM, New York, NY, USA, 183–193. https://doi.org/10.1145/2688500.2688507

[93] Mingxing Zhang, Yongwei Wu, Kang Chen, Teng Ma, and Weimin Zheng. 2016. Measuring and Optimizing Distributed Array Programs. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 912–923. https://doi.org/10.14778/2994509.2994511

[94] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the Hidden Dimension in Graph Processing. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 285–300.

[95] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *Proceedings of 2018 IEEE International Symposium onHigh Performance Computer Architecture (HPCA'18)*. IEEE, 544–557.

[96] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond Stateful Stream Querying over Fast-evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 614–630. https://doi.org/10.1145/3132747.3132777

[97] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Guang Tan, and Bing Bing Zhou. 2017. HotGraph: Efficient Asynchronous Processing for Real-World Graphs. *IEEE Trans. Comput.* 66, 5 (May 2017), 799–809. https://doi.org/10.1109/TC.2016.2624289

[98] Xiaohan Zhao, Adelbert Chang, Atish Das Sarma, Haitao Zheng, and Ben Y. Zhao. 2013. On the Embeddability of Random Walk Distances. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1690–1701. https://doi.org/10.14778/2556549.2556554

[99] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management (AAIM'08)*. Springer-Verlag, Berlin, Heidelberg, 337–348. https://doi.org/10.1007/978-3-540-68880-8_32

[100] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 301–316.

[101] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Conference on Annual Technical Conference (USENIX ATC'15)*.

Received August 2015; revised October 2016; accepted April 2018