PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs

Rong Chen, Jiaxin Shi, Yanzhe Chen, Haibo Chen

Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
{rongchen, jiaxinshi, yanzhechen, haibochen}@sjtu.edu.cn

Abstract

Natural graphs with skewed distribution raise unique challenges to 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 distribution in natural graphs also calls for differentiated processing on high-degree and low-degree vertices. We then introduce PowerLyra, a new graph computation engine that embraces the best of both worlds of existing graph-parallel systems, by dynamically applying different computation and partitioning strategies for different vertices. PowerLyra further provides an efficient hybrid graph partitioning algorithm (hybrid-cut) that combines edge-cut and vertex-cut with heuristics. Based on PowerLyra, we design locality-conscious data layout optimization to improve cache locality of graph accesses during communication. PowerLyra is implemented as a separate computation engine of PowerGraph, and can seamlessly support various graph algorithms. A detailed evaluation on two clusters using graph-analytics and MLDM (machine learning and data mining) applications show 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.

EuroSys'15, April 21–24, 2015, Bordeaux, France.

http://dx.doi.org/10.1145/2741948.2741970

1. Introduction

Graph-structured computation has become increasingly popular, which is evidenced by its emerging adoption in a wide range of areas including social computation, web search, natural language processing and recommendation systems [9, 19, 38, 48, 58, 62]. The strong desire for efficient and expressive programming models for graph-structured computation has recently driven the development of a number of graph-parallel systems such as Pregel [35], GraphLab [33] and PowerGraph [18]. They usually follow the "think as a vertex" philosophy [35] by coding graph computation as vertex-centric programs to process vertices in parallel and communicate across edges.

On the other hand, the distribution of graphs in the real world tends to be diversified and continue evolving [31]. For example, some existing datasets exhibit a skewed power-law distribution [17] where a small number of vertices have a significant number of neighboring vertices, while some other existing datasets (e.g., SNAP [39]) exhibit a more balanced distribution. The diverse properties inside and among graph datasets raise new challenges to efficiently partition and process such graphs [6, 18, 32].

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 [35] and GraphLab [33] centralize their design in making resources locally accessible to hide latency 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 [18] and GraphX [20] 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 some deficiencies for skewed and non-skewed (i.e. regular) graphs. For example, edge-cut [26, 44, 49, 52], which divides a graph by cutting cross-partition edges among subgraphs with the goal of evenly distributing vertices, usu-

	Pregel-like [2, 3, 35, 43]	**GraphLab** [33]	**PowerGraph** [18]	**GraphX** [20]	**PowerLyra**
Graph Placement	edge-cuts [44, 49, 52]	edge-cuts	vertex-cuts [12, 18, 24]	vertex-cuts	hybrid-cuts
Vertex Comp. Pattern	local	local	distributed	distributed	L: local H: distributed
Comm. Cost	$\leq$ #*edge-cuts*	$\leq 2 \times$ #*mirrors*	$\leq 5 \times$ #*mirrors*	$\leq 4 \times$ #*mirrors*	L: $\leq$ #*mirrors* H: $\leq 4 \times$ #*mirrors*
Dynamic Comp.	no	yes	yes	yes	yes
Load Balance	no	no	yes	yes	yes

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

ally results in replication of edges as well as imbalanced messages with high contention. In contrast, vertex-cut [12, 18, 24], 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 skewed vertex distribution demand differentiated computation and partitioning on low-degree and high-degree vertices. Based on our analysis, we introduce PowerLyra, a new graph-parallel system that embraces the best of both worlds of existing systems. The key idea of PowerLyra is to adaptively processing different vertices according to their degrees.

PowerLyra follows the GAS (Gather, Apply and Scatter) model [18] and can seamlessly support existing graph algorithms under such a model. Internally, PowerLyra distinguishes the processing of low-degree and high-degree vertices: it uses centralized computation for low-degree vertices to avoid frequent communication and only distributes the computation for high-degree vertices.

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 hybrid-cut algorithm 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.

Finally, PowerLyra mitigates the poor locality and high interference among threads during the communication phase by a locality-conscious graph layout optimization based on hybrid-cut. It trades a small growth of graph ingress time for a notable speedup during graph computation.

We have implemented PowerLyra[1] as a separate engine of PowerGraph, which comprises about 2500 lines of C++ code. Our evaluation on two 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, 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 locality-conscious data layout optimization (§5).
- A detailed evaluation that demonstrates the performance benefit of PowerLyra (§6).

2. Background and Motivation

Many graph-parallel systems, including PowerLyra, abstract computation as 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 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 they are representatives of existing systems.

2.1 Skewed Graphs

Natural graphs, such as social networks (follower, citation, and co-authorship), email and instant messaging graphs, or web graphs (hubs and authorities), usually exhibit skewed distribution, such as power-law degree distribution [17]. This implies that a major fraction of vertices have relatively few neighbors (i.e., low-degree vertex), while a small fraction of vertices has a significant large number of neighbors (i.e., high-degree vertex). Given a positive power-law constant α, the probability that the vertex has degree d under power-law distribution is

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

[1] The source code and a brief instruction of PowerLyra are available at http://ipads.se.sjtu.edu.cn/projects/powerlyra.html

```
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

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

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

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

(b) PowrGraph/PowerLyra

Figure 1. *The sample code of PageRank on various systems.*

The lower exponent α 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 [18]. Though there are also other models [31, 41, 42, 55] for skewed graphs, we restrict the discussion to power-law distribution due to space constraints. However, PowerLyra is not bound to such a distribution and should benefit other models with skewed distribution as well. (having high-degree and low-degree vertices).

2.2 Existing Systems on Skewed Graphs

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. Tab. 1 provides a comparative study of typical graph-parallel systems.

Pregel and its open-source relatives [2, 3, 43] use the BSP (*Bulk Synchronous Parallel*) model [53] with explicit messages to fetch all resources for the vertex computation. Fig. 1(a) illustrates an example implementation of PageRank [9] 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 a global convergence estimated by a distributed aggregator is reached. As shown in Fig. 2, Pregel adopts random (hash-based) edge-cut evenly assigning two vertices in the sample graph to two machines, 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 [33].

GraphLab replicates vertices for all edges spanning machines and leverages an additional vertex activation message to support dynamic computation. In Fig. 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 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 from mirrors to the master. PageRank implemented in GraphLab is similar to that of Pregel, except

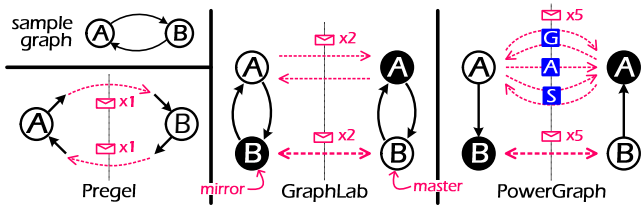


Figure 2. *A comparison of graph-parallel models.*

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 to split a vertex into multiple replicas in different machines to parallelize the computation for a single vertex. Fig. 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 Fig. 2, the edges of a single vertex are assigned to multiple machines to evenly distribute workloads, and the replicas of vertex are placed in machines with its edges.

GraphX [20] extends the general dataflow framework in Spark [60], 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 workload for skewed graphs.

2.2.1 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 [23], 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 Fig. 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

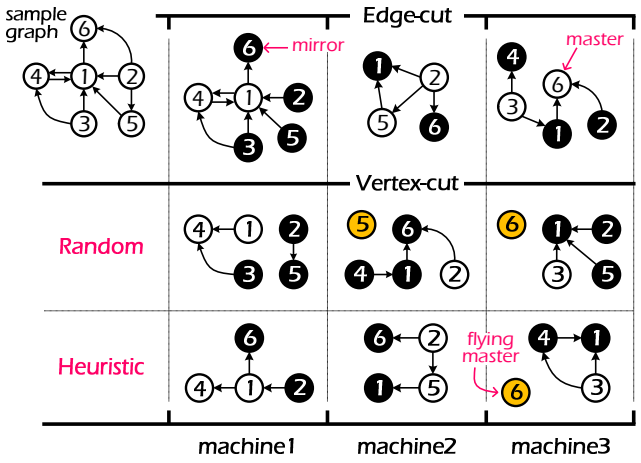


Figure 3. *A comparison of graph partitioning algorithms.*

model, which split a vertex into multiple 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 (Fig. 2). However, as the major number of vertices are only with small degrees 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). Yet, both PowerGraph and GraphX still require redundant communication and data movement. The workload is always distributed to all replicas even without such edges. Under the Random vertex-cut in Fig. 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.

2.2.2 Issues with Graph Partitioning

Graph partitioning plays a vital role in reducing communication and ensuring load balance. Traditional balanced p-way **edge-cut** [44, 49, 52], evenly assigns *vertices* of a graph to p machines to minimize the number of *edges* spanning machines. Under edge-cut in Fig. 3, six vertices are 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 distribution are difficult to partition using edge-cut [6, 30], since skewed vertices will cause a burst of communication cost and work imbalance. Vertex 1 in Fig. 3 contributes about half of 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** [18] 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 (λ) (i.e., the average number of replicas for a vertex), since it incurs poor placement of low-degree vertices. In Fig. 3, Random vertex-cut creates a mirror for vertex 3 even if it has only two edges[2].

Algorithm & Graph	Vertex-cut	λ	Time (Sec)	
			Ingress	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 rec.	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**

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. λ means replication factor. Ingress means loading graph into memory and building local graph structures.*

To reduce replication factor, PowerGraph uses a *greedy* heuristic [18] to accumulate adjacent edges on the same machine. In practice, applying the greedy heuristic to all edges (i.e., Coordinated) incurs a significant penalty during graph partitioning [23], mainly caused by exchanging vertex information among machines. Yet, using greedy heuristic independently on each machine (i.e., Oblivious) will notably increase the replication factor.

The *constrained* vertex-cut [24] (e.g., Grid) is proposed to strike a balance between ingress and execution time. It follows the classic 2D partitioning [11, 59] 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 independently calculated on each machine by hashing, constrained vertex-cut can significantly reduce the ingress time[3]. 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). Further, constrained vertex-cut necessitates the number of partitions (N) close to be a square number to get a reasonably balanced graph partitioning.

Some work (e.g., [23]) argues that intelligent graph placement schemes may dominate and hurt the total execution time, whereas such argument just partially[4] works for *greedy heuristic partitioning and simple graph algorithms*. First,

[2] 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.

[3] Coordinated greedy vertex-cut has been deprecated due to its excessive graph ingress time and buggy, meanwhile both PowerGraph and GraphX have adopted Grid-like vertex-cut as the preferential partitioning algorithm.

[4] GraphLab has been highly optimized in the 2.2 release, especially for ingress time with parallel loading.

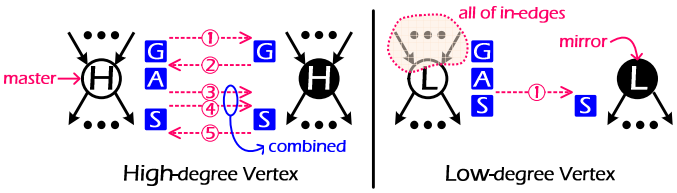


Figure 4. *The computation model on high-degree and low-degree vertex for algorithms gathering along in-edges and scattering along out-edges.*

naive random partitioning does not necessarily imply efficiency in ingress 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 ingress time will become relatively small to the overall computation.

Tab. 2 illustrates a comparison of various state-of-the-art vertex-cuts of PowerGraph for 48 partitions. *Random* vertex-cut performs worst in both ingress 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 ingress time. *Oblivious* vertex-cut decreases ingress time (but still slower than Random) while doubling replication factor and execution time. *Grid* vertex-cut outperforms coordinated vertex-cuts in ingress time by 2.8X but decreasing runtime performance. Besides, the percent of ingress time for PageRank on the Twitter follower graph with 10 iterations[5] of graph computation ranges from 24.2% to 56.8%, while for ALS on Netflix motive recommendation graph only ranges from 3.6% to 22.8%. The random *Hybrid*-cut of PowerLyra (see §4) provides optimal performance by significantly decreasing execution time while only slightly increasing ingress time (compared to Grid).

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 low-degree and high-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 Abstraction and Engine

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 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.

PowerLyra employs a simple loop to express iterative computation of graph algorithms. The graph engine sequentially executes the Gather, Apply and Scatter phases in each iteration, but processes vertices differently according to the vertex degrees.

3.2 Differentiated Vertex Computation

Processing high-degree vertex: To exploit **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 (the left part of Fig. 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 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 Fig. 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 help of its mirrors. The message to activate mirrors (that further scatter their neighbors along out-edges) are combined with the message for updating vertex data (sent from master to mirrors).

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 [14].

[5] Increasing iterations, like [23, 35], will further reduce the percent.

Type	Gather	Scatter	Example Algo.
Natural	`in` or `none` `out` or `none`	`out` or `none` `in` or `none`	PR, SSSP DIA[25]
Other	`any`	`any`	CC, ALS[5]

Table 3. *A classification of various graph algorithms.*

3.3 Generalization

PowerLyra applies a simplified model for low-degree vertices to minimize communication overhead, but 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 way to handle 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 of PowerGraph accordingly. Hence, it can be checked at runtime without any changes to applications.

Tab. 3 summarizes the classification of graph algorithms. PowerLyra naturally 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) [25]. For such algorithms, PowerLyra needs up to one message per mirror for low-degree vertices in each iteration.

For *Other* algorithms that gather and scatter data via any edges, PowerLyra asks mirrors to do gathering or scattering operations like those of high-degree vertices, but only *on demand*. For example, *Connected Components* (CC) 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.

4. Distributed Graph Partitioning

This section describes a new hybrid vertex-cut algorithm that uses differentiated partitioning for low-degree and high-degree vertices. Based this, a new heuristic, called *Ginger*, is provided to further optimize partitioning for PowerLyra.

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 (λ). Hence, existing vertex-cuts mostly aim at reducing the overall λ of all vertices. However, we observe that the key is instead reducing λ of low-degree vertices, since high-degree vertices inevitably need to be replicated on most of machines. Nevertheless, many current heuristics for vertex-cuts have a bias towards

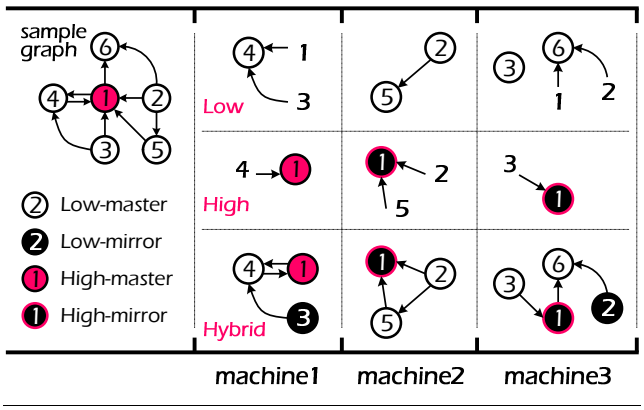


Figure 5. *The hybrid-cut on sample graph.*

high-degree vertices, while paying little attention to low-degree vertices.

We propose a balanced p-way **hybrid-cut** that focuses on reducing λ of low-degree vertices. It uses differentiated partitioning to low-degree and high-degree vertices. To avoid replication of edges, each edge is exclusively assigned to its target vertex (the destination of an edge)[6]. 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 in-edges by hashing their *source* vertices. After that, hybrid-cut creates replicas and constructs local graphs, as done in typical vertex-cuts.

As shown in Figure 5, all vertices along with their in-edges are assigned as low-degree vertices except vertex 1, whose in-edges are assigned as high-degree vertex. For example, the edge $(1,4)$ and $(3,4)$ are placed in machine 1 with the master of low-degree vertex 4, while the edge $(2,1)$ and $(5,1)$ are placed in machine 2 with the mirror of high-degree vertex 1. The partition constructed by hybrid-cut only yields four mirrors and achieves good load balance.

Hybrid-cut addresses the major issues in edge-cut and vertex-cut on skewed graphs. First, hybrid-cut can provide 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 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. Second, hybrid-cut provides unidirectional access locality for low-degree vertices, which can used by hybrid computation model (§3) to reduce communication cost at runtime. Third, hybrid-cut is very efficient in graph ingress, since it is a hash-based partitioning for both low-degree and high-degree vertices. Finally, the partition constructed by hybrid-cut is naturally balanced on vertices and edges. Randomized placement of low-degree vertices leads to bal-

[6] The edge could also exclusively belong to its source vertex, which depends on the direction of locality preferred by the graph algorithm.

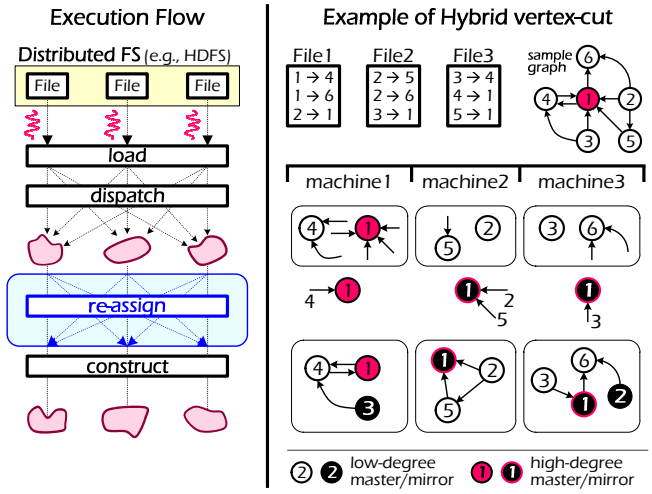


Figure 6. *The execution flow of hybrid-cut.*

ance of vertices, which is almost equivalent to the balance of edges for low-edge vertices. For high-degree vertices, all in-edges are evenly assigned since they are hosted by the owner machine of source vertices, which are also evenly assigned.

Constructing hybrid-cut: A general approach to constructing a hybrid-cut is adding an extra re-assignment phase for high-degree vertices to the original streaming graph partitioning. The left part of Fig. 6 illustrates the execution flow of graph ingress using hybrid-cut, and the right part shows the sample graph in each stage. First, the worker on each machine loads the raw graph data from underlying distributed file systems in parallel, and dispatches a vertex and its in-edges according to the hash of a vertex. Each worker counts the in-degree of vertices and compares it with a user-defined threshold (θ) to identify high-degree vertices. After that, in-edges of high-degree vertex are re-assigned by hashing source vertices. Finally, each worker creates mirrors to construct a local graph as normal vertex-cut.

This approach is compatible with existing formats of raw graph data, but incurs a few communication overhead due to re-assigning in-edges of high-degree vertices. For some graph file format (e.g., adjacent list), the worker can directly identify high-degree vertices and distributes edges in loading stage to avoid extra communication, 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 heuristic algorithm, namely **Ginger**, inspired by Fennel [52], 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.

Graph	$\|V\|$	$\|E\|$
Twitter [28]	42M	1.47B
UK-2005 [8]	40M	936M
Wiki [22]	5.7M	130M
LJournal [16]	5.4M	79M
GoogleWeb [32]	0.9M	5.1M

α	#Edges
1.8	673,275,560
1.9	249,459,718
2.0	105,156,471
2.1	53,824,704
2.2	38,993,568

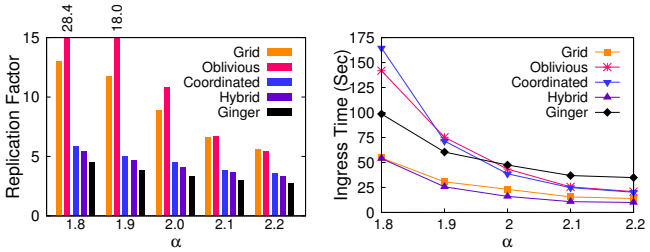
Table 4. *A collection of real-world graphs and randomly constructed power-law graphs with varying α and fixed 10 million vertices. Smaller α produces denser graphs.*

Considering the special requirements of hybrid-cut, we differ from Fennel in several aspects to improve performance and balance. First, Fennel is inefficient to partition skewed graphs due to high-degree vertices. Hence, we just use 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 to determine the host machine. By contrast, Ginger only estimates edges in one direction to decrease ingress time. Finally, Fennel just focuses on the balance of vertices, by only using the number of vertices $|S_i|^V$ as parameter of the balance formula ($\delta c(x)$). Hence, it usually causes a significant imbalance of edges even for regular graphs. Hence, to improve balance of edges, we add the normalized number of edges $\mu|S_i|^E$ into the parameter of the balance formula, where μ is the ratio of vertices to edges, and $|S_i|^E i$ is the number of edges in S_i. The composite balance parameter becomes $(|S_i|^V + \mu|S_i|^E)/2$.

4.3 Graph Partitioning Comparison

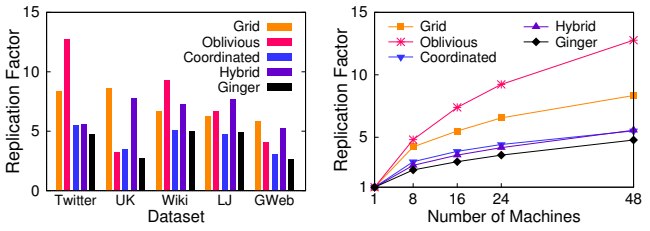
We use a collection of real-world and synthetic power-law graphs to compare various graph partitioning algorithms, as shown in Tab. 4. Most real-world graphs were from the Laboratory for Web Algorithmics [4] and Stanford Large Network Dataset Collection [39]. Each synthetic graph has 10 million vertices and a power-law constant (α) ranging from 1.8 to 2.2. Smaller α produces denser graphs. They were generated by tools in PowerGraph, which randomly sample the in-degree of each vertex from a Zipf distribution [7] and then add in-edges such that the out-degrees of each vertex are nearly identical. In all cases, hybrid-cut retains balanced load for both edges and vertices.

In Fig. 7, we compare the replication factor and graph ingress time of hybrid-cut against various vertex-cuts for the Power-law graphs with different constant (α). Random hybrid-cut notably outperforms Grid vertex-cut with slightly less ingress time, and the gap increases with the growing of skewness of the graph, reaching up to 2.4X (α=1.8). Oblivious vertex-cut has poor replication factor and ingress time



(a) Replication Factor *(b) Ingress Time*

Figure 7. *The replication factor and ingress time for the Power-law graphs with different constants on 48 machines.*



(a) Real-world Graphs *(b) Twitter Follower Graph*

Figure 8. *The replication factor for the real-world graphs on 48 machines and replication factor of the Twitter follower graph with increasing machines.*

for the Power-law graphs. Though Coordinated vertex-cut provides comparable replication factor to Random hybrid-cut (10% higher), it triples the ingress time. Ginger can further reduce the replication factor by more than 20% (Fig. 7(a)), but also increases ingress time like Coordinated vertex-cut.

For real-world graphs (Fig. 8(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. However, Ginger still performs much better in such cases, up to 3.11X improvement over Grid on UK. Fig. 8(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% ingress time, and outperforms Grid and Oblivious vertex-cut by 1.74X and 2.67X respectively.

5. Locality-conscious Graph Layout

Graph computation usually exhibits poor data access locality [34], due to irregular traversal of neighboring vertices along edges as well as frequent message exchanges between masters and mirrors. 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 separate arrays and sequentially assigns a unified local ID in each machine to vertex for indexing, which is mapped to global vertex ID. As shown in the bottom of Fig. 9, in each phase, the worker thread sequentially

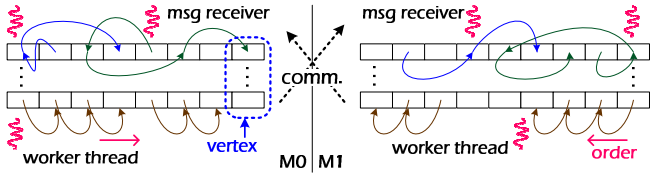


Figure 9. *An example of execution along with comm.*

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 and the order of accessing vertices is only determined by the order in the sender. However, since the order of messages is predefined by the traversal order, accesses to vertices have poor locality due to mismatching of orders between sender and receiver. Worse even, messages from multiple machines are processed in parallel and interfere with each other (shown in the upper part of Fig. 9). Though it appears that both problems could be partially addressed at runtime by sorting or dispatching messages on the fly [14], our experience shows that this will cause notable overhead instead of performance boost, due to non-trivial CPU cycles.

PowerLyra mitigates the above problems by extending hybrid-cut in four steps, as shown in Fig. 10. 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, all masters and mirrors of high-degree and low-degree vertices are mixed and stored in random order. For example, the order of update messages from masters in machine 0 (`M0`) mismatches the order of their mirrors stored in machine 1 (`M1`).

First, hybrid-cut divides vertex space into 4 **zones** to store (`H`)igh-degree masters (`Z0`), (`L`)ow-degree masters (`Z1`), (`h`)igh-degree mirrors (`Z2`) and (`l`)ow-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 helps to skip the unnecessary vertex traversal and avoid interference between the sender and the receiver. 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 reduce the working set and the interference when multiple receiver 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 on messages from the master of low-degree vertices in `M0` (`L0`) and `M2` (`L2`) are restricted to different groups (`l0` and `l2`) on `M1`.

Third, hybrid-cut **sorts** the masters and mirrors within a group according to the global vertex ID and sequentially assigns its local ID. Because the order of messages follows the order of local ID, sorting ensures masters and mirrors

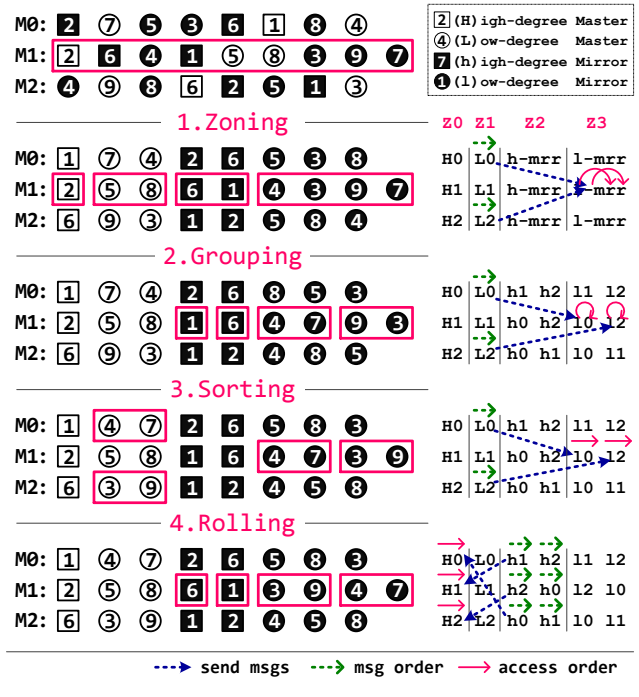
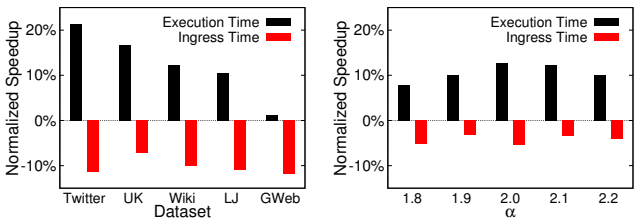


Figure 10. *An example of data placement optimization.*

have the same relative ordering 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 the similar order, then it would lead to high contention on masters. For example, messages from mirrors in `M1` and `M2` (`h0` and `l0`) will be simultaneously updated to the master in `M0` (`H0` and `L0`). Therefore, hybrid-cut places mirror groups in a **rolling** order: the mirror groups in machine n for p partitions start from $(n+1)$ mod p. For example, the mirror groups of high-degree vertex in `M1` start from `h2` then `h0`, where $n = 2$ and $p = 3$.

Though we separately describe above four steps, they are indeed implemented as one step of hybrid-cut 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 graph ingress time due to the above optimization is modest (less than 10% for the Power-law graphs and around 5% for real-world graphs), resulting in usually more than 10% speedup (21% for the Twitter follow graph), as shown in Fig. 11. The speedup for Google Web graph is negligible, as the number of vertices is very small. For graph computation that processes a graph multiple iterations and even multiple times in memory, we believe a small increase of graph ingress time is worthwhile for an often larger speedup in execution time.



(a) Real-world Graphs *(b) Power-law Graphs*

Figure 11. *The effect of locality-conscious optimization.*

6. Evaluation

We have implemented PowerLyra based on the latest GraphLab 2.2 (release in Mar. 2014) as a separate engine, which can seamlessly run all existing graph algorithms in GraphLab and respect the fault tolerance model. PowerLyra currently supports both synchronous and asynchronous execution. Due to the space restriction, we focus on the experiment of synchronous mode, which is the default mode for most graph tools and adopted as the sole execution mode of most graph processing systems. To illustrate the efficiency and generality of PowerLyra, we further port the Random hybrid-cut to GraphX.

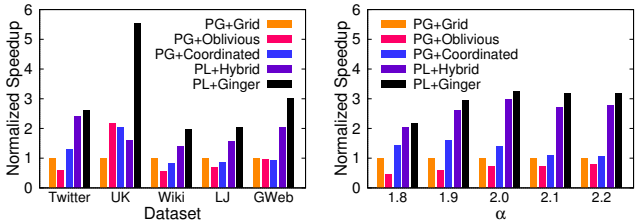
We evaluate PowerLyra with hybrid-cut (Random and Ginger) against PowerGraph with vertex-cut (Grid, Oblivious and Coordinated), and report the average results of five runs for each experiment. Most experiments are performed on a dedicated, VM-based 48-node EC2-like cluster. Each node has 4 AMD Opteron cores and 12GB DRAM. All nodes are connected via 1Gb Ethernet. To avoid exhausting memory for other systems, a 6-node in-house physical cluster with total of 144 cores and 384GB DRAM is used to evaluate the scalability in terms of data size (§6.3) and the comparison with other systems (§6.9). We use the graphs listed in Tab. 4 and set 100 as the default threshold of hybrid-cut during our evaluation.

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:

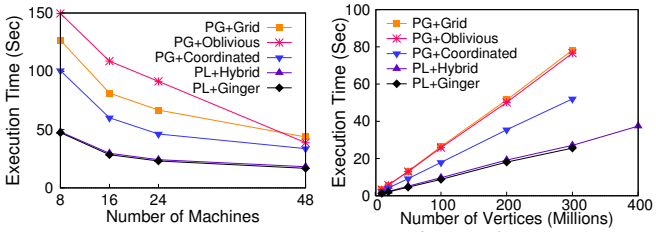
PageRank (PR) computes the rank of each vertex based on the ranks of its neighbors [9], which belongs to *Natural* algorithms that gather data along in-edges and scatter data along out-edges, for which PowerLyra should have significant speedup. Unless specified, the execution time of PageRank is the average of 10 iterations.

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 [25]. DIA belongs to the inverse *Natural* type of algorithms that gather data along out-edges and scatter none. In such a case, PowerLyra is still expected to show notable improvements.



(a) Real-world Graphs *(b) Power-law Graphs*

Figure 12. *Overall performance comparison between PowerLyra and PowerGraph on the real-world and power-law graphs for PageRank using 48 machines.*



(a) #Machines *(b) Graph Size*

Figure 13. *A comparison between PowerLyra and PowerGraph for the Twitter follower graph with increasing machines and for the Power-law (α=2.2) graph on the 6-node cluster with increasing data size.*

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 along 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 layout optimizations.

6.2 Performance

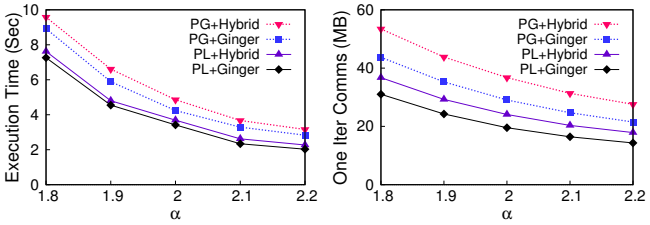
We compare the execution time of different systems and partitioning algorithms as in the PowerGraph paper [18]. Fig. 12(a) shows the speedup of PowerLyra over PowerGraph on real-world graphs. The largest speedup comes from UK graph for the Ginger hybrid-cut due to relatively high reduction of replication factor (from 8.62 to 2.77, Fig. 8). 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 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 better computing efficiency of low-degree vertices.

As shown in Fig. 12(b), PowerLyra performs better for the Power-law graphs using hybrid-cut, especially for high power-law constants (i.e., α) due to higher percent 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 Fig. 12(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 balanced partition with a small replication factor (Fig. 7).

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 graph.

Fig. 13 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 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.

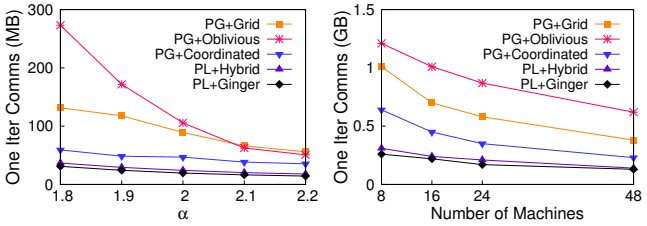


(a) Execution Time *(b) Communication Data Size*

Figure 14. *A comparison between PowerLyra and PowerGraph for the Power-law graphs with different constants on 48 machines using Hybrid and Ginger vertex-cut.*

6.4 Effectiveness of Graph Engine

Hybrid-cut can also be applied to 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 Fig. 14(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.



(a) Power-law Graphs *(b) Twitter Follower Graph*

Figure 15. *One iteration communication data size for the Power-law graphs with different constants on 48 machines and for the Twitter follower graph with increasing machines.*

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 major fraction) only require one message exchange in each iteration. As shown in Fig. 15, 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.

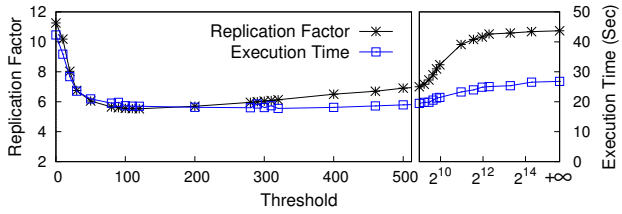
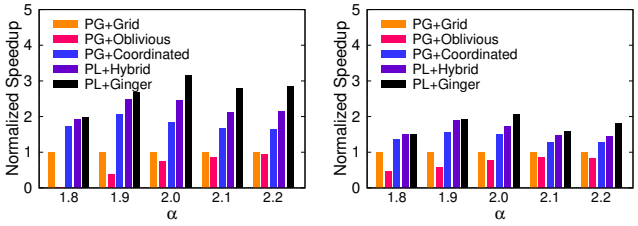


Figure 16. *The impact of threshold in PowerLyra on replication factor and execution time for the Twitter follower graph using PageRank.*

6.6 Threshold

To study the impact of different thresholds, we run PageRank on the Twitter follower graph with different thresholds. As shown in Fig. 16, using high-cut (θ=0) or low-cut (θ=+∞) 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 increasing threshold, the replication factor rapidly decreases and then slowly increases. The best runtime performance usually does not occur simultaneously with the lowest replication factor, since the increase of threshold also decreases the number of high-degree vertices, which benefits the overall performance. The ultimate metric for the optimal threshold is beyond the scope of this paper. Fortunately, the execution time is relatively stable for a large range of thresholds. In Fig. 16, the difference of execution time under threshold from 100 to 500 is lower than 1 sec.



(a) Approximate Diameter *(b) Connected Components*

Figure 17. *A comparison between PowerLyra and PowerGraph on the Power-law graphs for Approximate Diameter and Connected Components algorithms on 48 machines.*

Algorithm & Graph	Vertex-cut	λ	Time (Sec)	
			Ingress	Execution
PageRank & RoadUS[1] |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**

Table 5. *A comparison of various graph partitioning algorithms on 48 machines using PageRank (10 iterations) for the RoadUS graph.*

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 Fig. 17(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 PowerGraph with Grid vertex-cut by up to 1.88X and 2.07X using Random and Ginger hybrid-cut respectively (Fig. 17(b)).

We also investigate the performance of PowerLyra for non-skewed graphs like road networks. Tab. 5 illustrates a performance comparison between PowerLyra and PowerGraph for PageRank with 10 iterations on RoadUS [1], the road network of the United States. The average degree of RoadUS is less than 2.5 (no high-degree vertex). 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.

Netflix Movie Recommendation [63]				Replication Factor	
$\|V\|$	$\|E\|$	Vertex Data	Edge Data	Grid	Hybrid
0.5M	99M	$8d + 13$	16	12.3	2.6

ALS [63]	d=5	d=20	d=50	d=100
PowerGraph	**10** / 33	**11** / 144	16 / 732	Failed
PowerLyra	13 / **23**	13 / **51**	**14 / 177**	**15 / 614**

SGD [50]	d=5	d=20	d=50	d=100
PowerGraph	**15** / 35	**17** / 48	21 / 73	28 / 115
PowerLyra	16 / **26**	19 / **33**	**19 / 43**	**20 / 59**

Table 6. *Performance (ingress / 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.*

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) [63] and Stochastic Gradient Descent (SGD) [50], are used to predict the movie ratings for each user on Netflix movie recommendation dataset [63], 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 higher accuracy of prediction while increasing both memory consumption and computational cost. As shown in Tab. 6, 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 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.

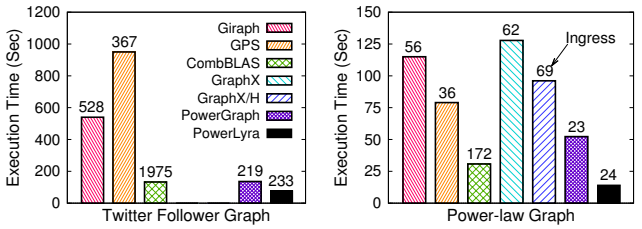


Figure 18. *Performance comparison for various systems on the Twitter follower and Power-law (α=2.0) graphs using PageRank. GraphX/H means GraphX with random hybrid-cut. The labels upon histogram are ingress time.*

6.9 Comparison with Other Systems

Readers might be interested in how the performance of PowerLyra compares to other graph processing systems, even if they adopts different designs such as graph-parallel abstraction [2, 3, 18, 23, 27, 35, 43], dataflow operators [20], sparse matrix operations [10] or declarative programming [45]. We 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[7] on our 6-node cluster[8].

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

Table 7. *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 means PowerLyra running on N machines.*

Fig. 18 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 ingress time is also labeled 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 for data transformation due to the limitation of the programming paradigm (sparse matrix). We further port the Random hybrid-cut to GraphX (i.e., GraphX/H), leading to a 1.33X speedup even without heuristic[9] and differentiated computation engine. 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.

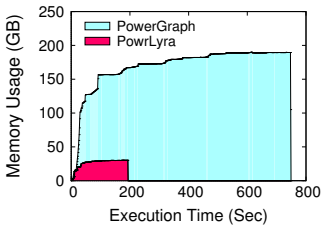
We further change the comparison targets to systems on single-machine platform. One machine of our 6-node cluster (24 cores and 64GB DRAM) is used to run in-memory (Polymer [61] and Galois [37]) and out-of-core (X-Stream[10] [40] and GraphChi [29]) systems for both in-memory and out-of-core graphs using PageRank with 10 iterations. As shown in Tab. 7, 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 in-memory graphs, while distributed solutions are more efficient for out-of-core graphs that can-

[7] The source code of LFGraph [23] is not available, and SociaLite [45] and Mizan [27] have some bugs to run on our clusters, which cannot be fixed in time by their authors.

[8] 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.

[9] We only implement Random hybrid-cut on GraphX for preserving its graph partitioning interface.

[10] 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.



	GX	GX/H
Memory Usage (GB)		
RDD	30.7	25.4
Garbage Collection		
Number	33	18
Time (s)	17.6	10.0

Figure 19. *(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 (α=2.0) for PageRank using the 6-node cluster*

not fit in the memory of a single machine. The current PowerLyra focuses on the distributed platform, resulting the relative 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 [61].

6.10 Memory Consumption

Besides performance improvement, PowerLyra also can 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 Fig. 19, 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 nodes for PageRank with a Power-law graph (α=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 [18, 20, 33, 35], but differs from them in adopting a novel differentiated graph computation and partitioning scheme for vertices with different degrees.

Other graph processing systems: Several prior systems [46] use a distributed in-memory key-value table abstraction to support graph processing. LFGraph [23] proposes a publish-subscribe mechanism to reduce communication cost but restricts graph algorithms just to one direction access. Mizan [27] leverages vertex migration for dynamic load balancing. Imitator [54] reuses computational replication for fault tolerance in large-scale graph processing to provide low-overhead normal execution and fast crash recovery. Giraph++ [51] provides several algorithm-specific optimizations for graph traversal and aggregation applications relying on the graph-centric model with partitioning information. PowerSwitch [57] embraces the best of both synchronous and asynchronous execution modes by adaptively switching graph computation between them. GPS [43] 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. There are also a few systems considering stream processing [15, 36] or temporal graphs [21].

Besides vertex-centric model, various programming paradigms are extended to handle graph processing. SociaLite [45] stores the graph data in tables and abstracts graph algorithms as declarative rules on the tables by Datalog. CombBLAS [10] 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.

None of existing graph processing systems use differentiated computation and partitioning. In addition, PowerLyra is orthogonal to above techniques and can further improve the performance of these systems on skewed graphs.

There are also several efforts aiming at leveraging multicore platforms for graph processing [29, 37, 40, 47, 61], which focus on such as improving out-of-core accesses [29], selecting appropriate execution modes [47], supporting sophisticated task scheduler [37], reducing random operations on edges [40], and adopting NUMA-aware dat layout and access strategy [61]. Such techniques should be useful to improve the performance of PowerLyra on each machine in the cluster.

Graph replication and partitioning: Generally, prior online graph partitioning approaches can be categorized into vertex-cut [18, 24] and edge-cut [44, 49, 52] according to their partition mechanism. Several greedy heuristics [18] and 2D mechanisms [11, 24, 59] are proposed to reduce communication cost and partitioning time on skewed graphs. Surfer [13] exploits the underlying heterogeneity of a public cloud for graph partitioning to reduce communication cost. However, most of them are degree-oblivious but focus on using a general propose for all vertices or edges. Degree-based hashing [56], to our knowledge, is the only other partitioning algorithm for skewed graphs considering the vertex degrees. However, it adopts a uniform partitioning strategy yet and requires long ingress time due to counting the degree of each vertex in advance. 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.

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 systems. 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. Performance 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 our shepherd Amitabha Roy and the anonymous reviewers for their insightful comments, Kaiyuan Zhang for evaluating graph-parallel systems on single machine platform and Di Xiao for porting Random hybrid-cut to GraphX. This work is supported in part by the National Natural Science Foundation of China (No. 61402284), the Doctoral Fund of Ministry of Education of China (No. 20130073120040), the Program for New Century Excellent Talents in University, Ministry of Education of China (No. ZXZY037003), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. TS0220103006), the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2014A05), the Shanghai Science and Technology Development Fund for high-tech achievement translation (No. 14511100902), a research grant from Intel and the Singapore NRF (CREATE E2S2).

References

[1] The 9th dimacs implementation challenge - shortest paths. http://www.dis.uniroma1.it/challenge9/.

[2] The Apache Giraph Project. http://giraph.apache.org/.

[3] The Apache Hama Project. http://hama.apache.org/.

[4] Laboratory for web algorithmics. http://law.di.unimi.it/.

[5] Large-scale parallel collaborative filtering for the netflix prize. In *AAIM*, pages 337–348, 2008.

[6] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, 2006.

[7] L. A. Adamic and B. A. Huberman. Zipfs law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[8] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, pages 107–117, 1998.

[10] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *IJHPCA*, 2011.

[11] Ü. i. t. V. Çatalyürek, C. Aykanat, and B. Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SISC*, 32(2):656–683, 2010.

[12] Ü. V. Çatalyürek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In *IRREGULAR*, pages 75–86. 1996.

[13] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *SoCC*, 2012.

[14] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan. Computation and communication efficient graph processing with distributed immutable view. In *HPDC*, 2014.

[15] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[16] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, pages 219–228, 2009.

[17] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.

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

[19] J. E. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *UAI*, pages 203–212, 2009.

[20] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[21] W. Hant, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *EuroSys*, 2014.

[22] H. Haselgrove. Wikipedia page-to-page link database. http://haselgrove.id.au/wikipedia.htm, 2010.

[23] I. Hoque and I. Gupta. Lfgraph: Simple and fast distributed graph analytics. In *TRIOS*, 2013.

[24] N. Jain, G. Liao, and T. L. Willke. Graphbuilder: scalable graph etl framework. In *GRADES*, 2013.

[25] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2): 8, 2011.

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

[27] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, 2013.

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

[29] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[30] K. Lang. Finding good nearly balanced cuts in power law graphs. Technical report, Yahoo Research Lab, 2004.

[31] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1):2, 2007.

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

[33] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB*, 5(8): 716–727, 2012.

[34] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *PPL*, 17(01):5–20, 2007.

[35] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[36] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, 2013.

[37] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471, 2013.

[38] B. Panda, J. Herbach, S. Basu, and R. Bayardo. PLANET: massively parallel learning of tree ensembles with MapReduce. *VLDB*, 2(2):1426–1437, 2009.

[39] S. N. A. Project. Stanford large network dataset collection. http://snap.stanford.edu/data/.

[40] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, 2013.

[41] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao. Measurement-calibrated graph models for social network experiments. In *WWW*, pages 861–870, 2010.

[42] A. Sala, X. Zhao, C. Wilson, H. Zheng, and B. Y. Zhao. Sharing graphs using differentially private graph models. In *IMC*, pages 81–98, 2011.

[43] S. Salihoglu and J. Widom. Gps: A graph processing system. In *SSDBM*, page 22. ACM, 2013.

[44] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper). In *Euro-Par*, pages 296–310, 2000.

[45] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: a datalog-based language for large-scale graph analysis. *VLDB*, 6(14):1906–1917, 2013.

[46] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[47] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

[48] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *VLDB*, 3(1-2):703–710, 2010.

[49] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *SIGKDD*, pages 1222–1230, 2012.

[50] G. Takács, I. Pilászy, B. Németh, and D. Tikk. Scalable collaborative filtering approaches for large recommender systems. *JMLR*, 10:623–656, 2009.

[51] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *VLDB*, 7(3), 2013.

[52] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*, pages 333–342, 2014.

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

[54] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. Replication-based fault-tolerance for large-scale graph processing. In *DSN*, 2014.

[55] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *EuroSys*, pages 205–218, 2009.

[56] C. Xie, L. Yan, W.-J. Li, and Z. Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In *NIPS*, pages 1673–1681, 2014.

[57] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *PPoPP*, 2015.

[58] J. Ye, J. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In *CIKM*, 2009.

[59] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *SC*, 2005.

[60] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[61] K. Zhang, R. Chen, and H. Chen. Numa-aware graph-structured analytics. In *PPoPP*, 2015.

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

[63] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM*, pages 337–348, 2008.