PowerGraph:
Distributed Graph-Parallel Computation on Natural Graphs

Presented By Louise He

Background

USENIX OSDI（Operating Systems Design and Implementation）

ARTICLE

PowerGraph: distributed graph-parallel computation on natural graphs

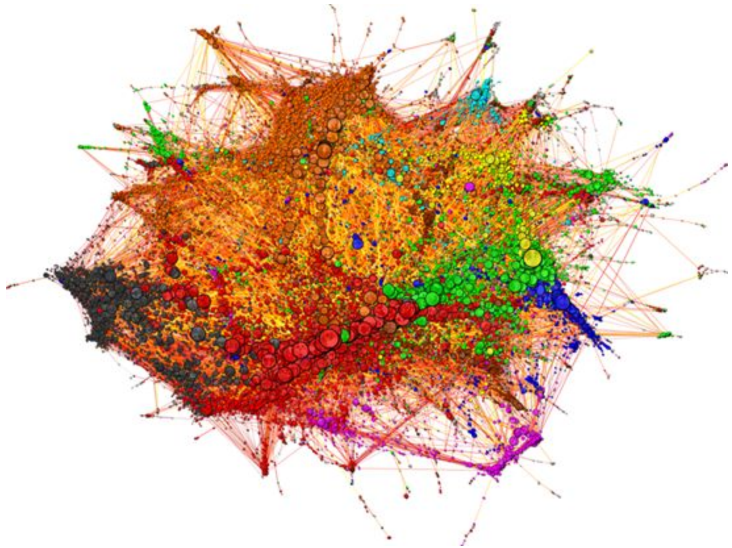
Authors: Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin | Authors Info & Claims

OSDI'12: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation • Pages 17 - 30

Published: 08 October 2012 Publication History

Problem Trying to Solve

Large-scale graph-structured computation plays a central role in tasks such as targeted advertising and natural language processing, and has driven the development of various graph-parallel abstraction models, such as Pregel and GraphLab.

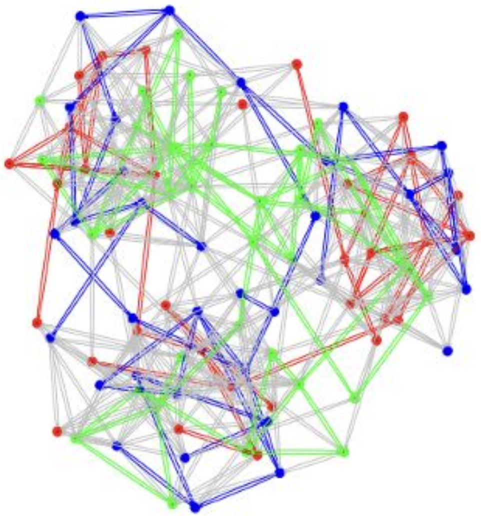


How Pregel and GraphLab Work

Pregel (vertex-centric distributed graph computing model)

- In each round, **only active vertices execute**: they read messages that arrived in the previous round, update local state, and send new messages that become visible in the next round.
- A global synchronization barrier at the end of each superstep keeps all machines in lockstep, which simplifies reasoning about correctness, convergence, and reproducibility

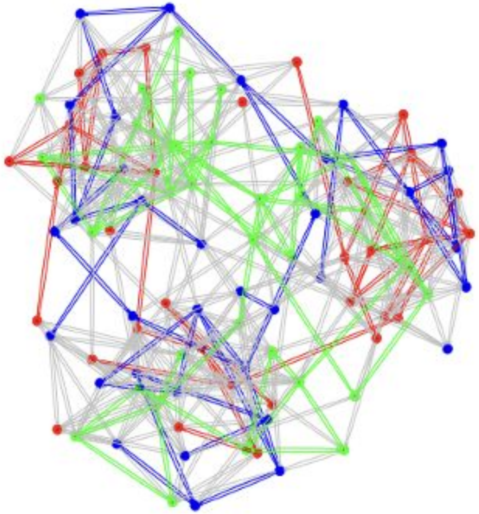
```
Message combiner(Message m1, Message m2) :
  return Message(m1.value() + m2.value());
void PregelPageRank(Message msg) :
  float total = msg.value();
  vertex.val = 0.15 + 0.85*total;
  foreach(nbr in out_neighbors) :
    SendMsg(nbr, vertex.val/num_out_nbrs);
```



How Pregel and GraphLab Work

GraphLab (dependency-driven graph-parallel model)

- runs update functions over a vertex/edge–scoped neighborhood while enforcing a chosen consistency model to control concurrent reads/writes.
- supports asynchronous or quasi-synchronous execution and priority scheduling so that regions with recent changes are updated first, reducing global synchronization overhead and improving efficiency on sparse, non-uniformly active workloads.



```
void GraphLabPageRank(Scope scope) :
  float accum = 0;
  foreach (nbr in scope.in_nbrs) :
     accum += nbr.val / nbr.nout_nbrs();
  vertex.val = 0.15 + 0.85 * accum;
```

Challenges of Existing Frameworks

Pregel (vertex-centric distributed graph computing model)

- **wait-for-the-slowes delays**, tends to suffer from load imbalance and message storms on power-law graphs

- less convenient for complex subgraph operations or highly dynamic graphs

```
Message combiner(Message m1, Message m2) :
  return Message(m1.value() + m2.value());
void PregelPageRank(Message msg) :
  float total = msg.value();
  vertex.val = 0.15 + 0.85*total;
  foreach(nbr in out_neighbors) :
     SendMsg(nbr, vertex.val/num_out_nbrs);
```

GraphLab (dependency-driven graph-parallel model)

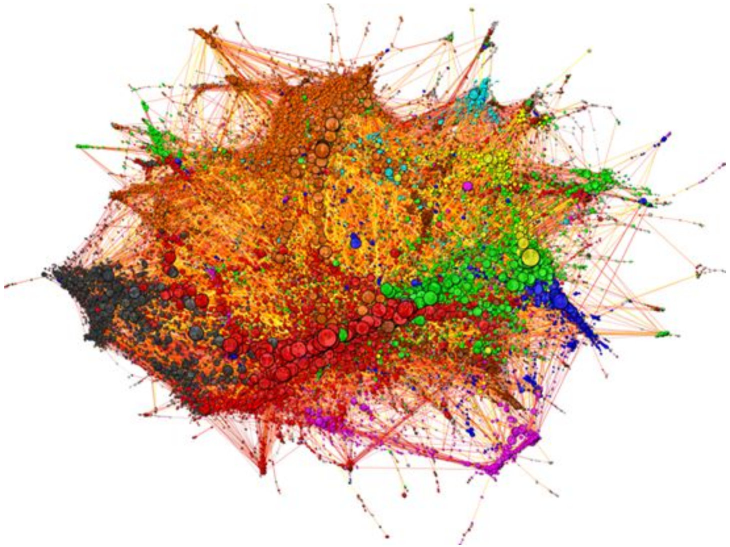
- higher implementation and debugging complexity
- need to manage locks and ghost/mirror state with associated communication and consistency costs
- potential non-deterministic execution orders, and contention around hotspot

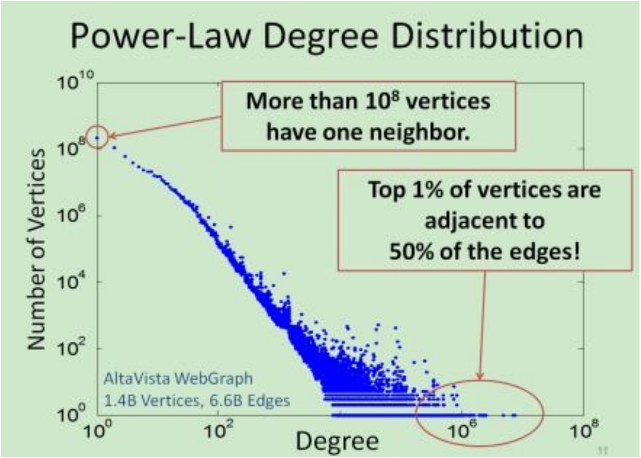
```
void GraphLabPageRank(Scope scope) :
  float accum = 0;
  foreach (nbr in scope.in_nbrs) :
     accum += nbr.val / nbr.nout_nbrs();
  vertex.val = 0.15 + 0.85 * accum;
```

Problem Trying to Solve

Large-scale graph-structured computation plays a central role in tasks such as targeted advertising and natural language processing, and has driven the development of various graph-parallel abstraction models, such as Pregel and GraphLab.

Real-world natural graphs often exhibit **highly skewed power-law degree distributions**, challenging the fundamental assumptions of existing abstractions and limiting system performance and scalability.



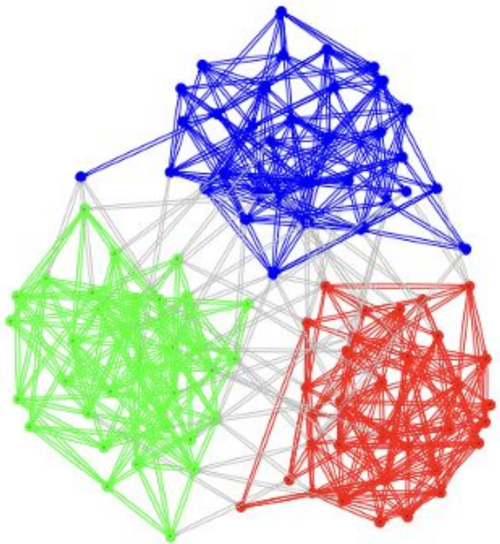


Challenges of Existing Frameworks

- **Work Balance:** Power-law distributions result in a few vertices having extremely high degrees (i.e., connecting to many edges), while most vertices have low degrees.
- **Communication:** High-degree vertices cause communication asymmetry
- **Storage:** Each machine must locally store adjacency information for its responsible vertices, with memory consumption linear in vertex degree.
- **Computation**: Existing abstractions treat each vertex program as an atomic unit, preventing further internal parallelization

PowerGraph

- Instead of binding computation to vertices, PowerGraph uses the **GAS (Gather–Apply–Scatter)** model to factor the vertex program **along edges**. Gather/Scatter execute in parallel across machines on edge partitions, while a master performs Apply, so the work of **high-degree vertices** is distributed and parallelized



PowerGraph - GAS

Gather-Apply-Scatter (GAS)

- Gather: accumulate information from neighborhood.
- Apply: apply the accumulated value to center vertex.
- Scatter: update adjacent edges and vertices.

`interface` *GASVertexProgram*(u) {
 // Run on gather_nbrs(u)
 gather(D_u, $D_{(u,v)}$, D_v) $\rightarrow$ *Accum*
 sum(*Accum* left, *Accum* right) $\rightarrow$ *Accum*
 apply(D_u, *Accum*) $\rightarrow$ D_u^{new}
 // Run on scatter_nbrs(u)
 scatter(D_u^{new}, $D_{(u,v)}$, D_v) $\rightarrow$ ($D_{(u,v)}^{\text{new}}$, *Accum*)
}

Figure 2: All PowerGraph programs must implement the stateless gather, sum, apply, and scatter functions.

Algorithm 1: Vertex-Program Execution Semantics

Input: Center vertex u
if *cached accumulator* a_u *is empty* **then**
  foreach *neighbor v in gather_nbrs(u)* **do**
    $a_u \leftarrow \mathrm{sum}(a_u, \mathrm{gather}(D_u, D_{(u,v)}, D_v))$
  end
end
$D_u \leftarrow \mathrm{apply}(D_u, a_u)$
foreach *neighbor v scatter_nbrs(u)* **do**
  $(D_{(u,v)}, \Delta a) \leftarrow \mathrm{scatter}(D_u, D_{(u,v)}, D_v)$
  if a_v *and* Δa *are not Empty* **then** $a_v \leftarrow \mathrm{sum}(a_v, \Delta a)$
  else $a_v \leftarrow$ Empty
end

PowerGraph - GAS

```
Pregel_PageRank(i, messages):
  // receive all the messages
  total = 0
  foreach(msg in messages):
    total = total + msg

  // update the rank of this vertex
  R[i] = total

  // send new messages to neighbors
  foreach(j in out_neighbors[i]):
    sendmsg(R[i] * wij) to vertex j
```

```
GraphLab_PageRank(i)
  // compute sum over neighbors
  total = 0
  foreach(j in in_neighbors(i)):
    total = total + R[j] * wji

  // update the PageRank
  R[i] = total

  // trigger neighbors to run again
  foreach(j in out_neighbors(i)):
    signal vertex-program on j
```

```
PowerGraph_PageRank(i):
  Gather(j -> i):
    return wji * R[j]

  sum(a, b):
    return a + b

  // total: Gather and sum
  Apply(i, total):
    R[i] = total

  Scatter(i -> j):
    if R[i] changed then activate(j)
```

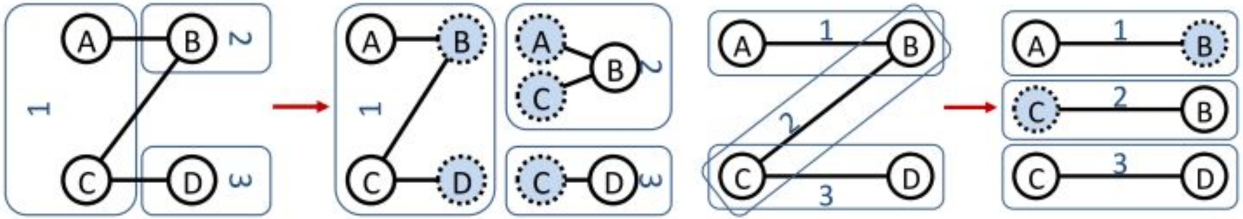
PowerGraph - Vertex-cut

Traditional Approach: Edge-Cut

- The graph is partitioned by edges into several subgraphs, with each machine storing a subset of edges.
- Each vertex can only belong to a single machine (the master).
- If an edge connects vertices on different machines, data transfer across the network is required.
- For supernodes (high-degree vertices), their numerous edges may be spread across many machines, leading to heavy remote access overhead

PowerGraph's Proposal: Vertex-Cut

- Partitioning is edge-centric
- A vertex is allowed to exist on multiple machines, where: One machine holds the **master** ; Other machines hold **mirrors**



(a) Edge-Cut

(b) Vertex-Cut

PowerGraph - Vertex-cut

Theorem 5.1. *If vertices are randomly assigned to p machines then the expected fraction of edges cut is:*

$$\mathbb{E}\left[\frac{|Edges\ Cut|}{|E|}\right] = 1 - \frac{1}{p}. \tag{5.1}$$

For a power-law graph with exponent α, the expected number of edges cut per-vertex is:

$$\mathbb{E}\left[\frac{|Edges\ Cut|}{|V|}\right] = \left(1 - \frac{1}{p}\right)\mathbb{E}[\mathbf{D}[v]] = \left(1 - \frac{1}{p}\right)\frac{\mathbf{h}_{|V|}(\alpha - 1)}{\mathbf{h}_{|V|}(\alpha)}, \tag{5.2}$$

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

Theorem 5.2 (Randomized Vertex Cuts)**.** *A random vertex-cut on p machines has an expected replication:*

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v\in V}|A(v)|\right] = \frac{p}{|V|}\sum_{v\in V}\left(1 - \left(1 - \frac{1}{p}\right)^{\mathbf{D}[v]}\right). \tag{5.5}$$

where $\mathbf{D}[v]$ denotes the degree of vertex v. For a power-law graph the expected replication (Fig. 6a) is determined entirely by the power-law constant α:

$$\mathbb{E}\left[\frac{1}{|V|}\sum_{v\in V}|A(v)|\right] = p - \frac{p}{\mathbf{h}_{|V|}(\alpha)}\sum_{d=1}^{|V|-1}\left(\frac{p-1}{p}\right)^{d} d^{-\alpha}, \tag{5.6}$$

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

Theorem 5.3. *For a given an edge-cut with g ghosts,* ***any*** *vertex cut along the same partition boundary has strictly fewer than g mirrors.*

Greedy Vertex-Cut

- Place edges one by one, choosing the machine that yields the lowest expected replication cost.
- Use heuristic rules to decide which machine an edge should be assigned to:
  - If both endpoints already have replicas on the same machine → place the edge there.
  - Otherwise, prefer the machine that hosts the endpoint with more unassigned edges remaining.

PowerGraph - Delta Caching

Core Ideas

- During the Scatter phase, return only the “delta” (Δa)
- The receiving vertex updates its local accumulator with Δa, instead of waiting for a full Gather.
- In the next Apply step, the vertex can directly use the already-maintained accumulator, thus skipping redundant Gather operations.

PageRank

```
// gather_nbrs: IN_NBRS
gather(D_u, D_(u,v), D_v):
  return D_v.rank / #outNbrs(v)
sum(a, b): return a + b
apply(D_u, acc):
  rnew = 0.15 + 0.85 * acc
  D_u.delta = (rnew - D_u.rank)/
              #outNbrs(u)
  D_u.rank = rnew
// scatter_nbrs: OUT_NBRS
scatter(D_u,D_(u,v),D_v):
  if(|D_u.delta|>ε) Activate(v)
  return delta
```

Greedy Graph Coloring

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return set(D_v)
sum(a, b): return union(a, b)
apply(D_u, S):
  D_u = min c where c ∉ S
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // Nbr changed since gather
  if(D_u == D_v)
    Activate(v)
  // Invalidate cached accum
  return NULL
```

Single Source Shortest Path (SSSP)

```
// gather_nbrs: ALL_NBRS
gather(D_u, D_(u,v), D_v):
  return D_v + D_(v,u)
sum(a, b): return min(a, b)
apply(D_u, new_dist):
  D_u = new_dist
// scatter_nbrs: ALL_NBRS
scatter(D_u,D_(u,v),D_v):
  // If changed activate neighbor
  if(changed(D_u)) Activate(v)
  if(increased(D_u))
    return NULL
  else return D_u + D_(u,v)
```

Results - Implementation Variants

- **Synchronous (BSP):**

  All machines proceed in lockstep. Results are deterministic and easy to debug, but performance is easily dragged down by slow machines.

- **Asynchronous:**

  Vertices can execute at any time, and updates propagate immediately. This yields faster speed and higher resource utilization, but results may differ depending on execution order (non-deterministic).

- **Asynchronous + Serializable (Async+S):**

  Builds on asynchronous execution by introducing a consistency protocol, ensuring the final outcome is equivalent to some sequential order of execution. This retains the efficiency of asynchronous execution while guaranteeing consistent results.

Results - Implementation Results

Setup

Deployed on a **64-node Amazon EC2 cluster** (cc1.4xlarge instances, dual quad-core Xeon, 23 GB RAM, 10 GbE network).

Implementations Compared

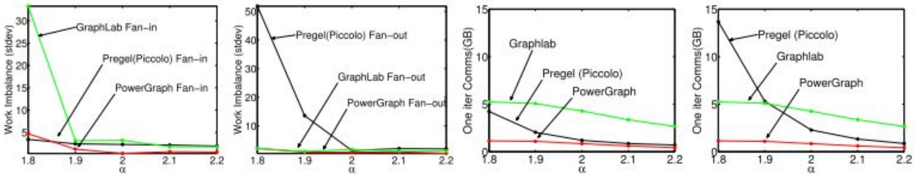
Three PowerGraph variants implemented: **Sync, Async, Async+Serializable**, to compare consistency-performance trade-offs.

Key Findings

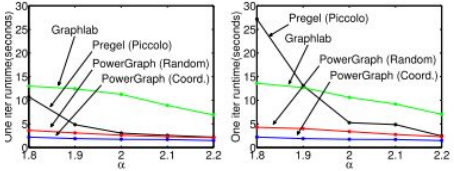
- On large workloads (PageRank, collaborative filtering, statistical inference), PowerGraph **substantially reduces runtime** compared to Pregel and GraphLab.
- Cuts communication and scales better, maintaining low network and storage overhead even in real deployment scenarios.
- Gains are **especially strong on power-law graphs**, with up to **~10× speedups**.

Results - Synthetic Graph Evaluation

- Metrics:
  - Work imbalance → std dev of worker runtimes per iteration
  - Communication volume → Bytes exchanged per iteration
  - Per-iteration runtime → Execution time per superstep
- Findings:
  - Pregel / GraphLab: Work and communication imbalance increase with skew (α ↑).
  - PowerGraph:
    - Much lower communication volume
    - Balanced work distribution
    - Lower per-iteration time



(a) Power-law Fan-In Balance (b) Power-law Fan-Out Balance (c) Power-law Fan-In Comm. (d) Power-law Fan-Out Comm.



(a) Power-law Fan-In Runtime (b) Power-law Fan-Out Runtime

Strengths

- **Significant performance gains**
  On real, large-scale datasets (e.g., social networks, web graphs), PowerGraph runs about **one order of magnitude faster**—roughly **10× speedup**—compared to traditional systems like Pregel and GraphLab.

- **Lower communication volume**
  dramatically reduces cross-machine communication on power-law graphs, leading to substantially lower overall communication cost than Pregel and GraphLab.

- **Better parallelism**
  splits the work of high-degree vertices across multiple machines, fully exploiting cluster parallelism.

- **Flexible execution**
  supports **synchronous**, **asynchronous**, and **asynchronous-serializable** modes, allowing users to trade off speed and consistency as needed.

- **Broad applicability**
  Across tasks such as **PageRank**, **collaborative filtering**, and **graph inference**, PowerGraph shows faster convergence and better scalability.

Weaknesses

- **Memory & storage overhead from vertex replication**
  Vertex-cut enables high parallelism but increases the replication factor, which raises both storage cost and synchronization overhead.

- **Restricted applicability of Delta Caching**
  Delta Caching requires accumulator operations to be commutative, associative, and preferably invertible; it cannot be applied to algorithms with non-linear or conditional updates.

- **Coordination cost in the parallel locking protocol**
  The Async+Serializable mode introduces a parallel locking protocol to ensure serializability, but this adds synchronization and coordination overhead.

- **Lack of native support for dynamic graphs**
  The system assumes static graphs and does not directly support dynamic graph updates (e.g., edge insertions or deletions).

Potential Next Step? / Discussions

- Could an adaptive runtime that switches between synchronous and asynchronous modes help balance performance and correctness more effectively than current fixed approaches?
- Should evaluation metrics extend beyond runtime and communication to include energy efficiency, peak memory, and latency for more realistic performance assessment?
- How can PowerGraph be extended to support dynamic graphs, and what techniques are most promising?