PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs (2012)

J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin

The next 25 minutes

1. Key Takeaway (1 min)
    - If you remember one thing from today…
2. Background (5 min)
    - What is the distributed graph-parallel paradigm?
    - Why does this paradigm exist?
3. Context (5 min)
    - What does existing work here look like? Pregel, GraphLab
    - What are the issues with these existing ideas?
4. PowerGraph (10 min)
    - What is this paper’s main argument?
    - Do the experimental results presented prove the argument?
5. Opinion + Discussion (4 min)

Key Takeaway

Vertex-cuts >> Edge-cuts

PowerGraph argues:

If your distributed graph contains very high-degree vertices, vertex-cuts are more performant than edge-cuts

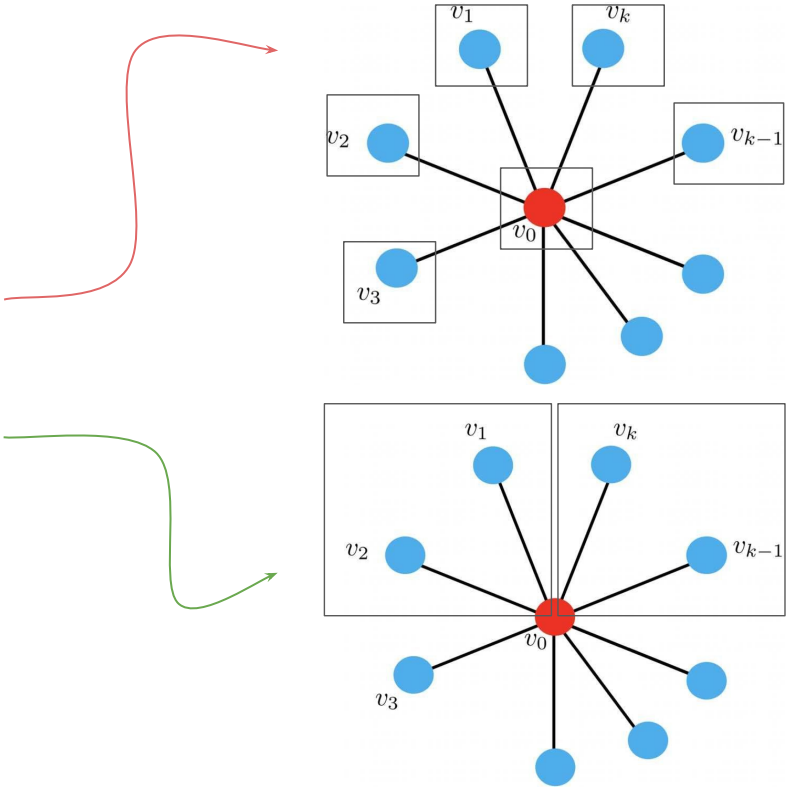


Figure adapted from Figure 1: Turab et al., “Existence of solutions for a class of non-boundary…”, Advances in Differential Equations, 2021

Background: Breaking down the paper title

Distributed Graph-Parallel Computation on Natural Graphs

Distributed Graph-Parallel Computation on Natural Graphs

Distributed?

Distributed Graph-Parallel Computation on Natural Graphs

Distributed? Around 2010, graphs got too big to fit on one machine – billions of vertices, trillions of edges – so needed to represent graphs across a cluster

With any distributed representation, you inevitably raise new questions

- **Storage**: how do you store adjacency information on each node?
- **Communication**: how do you manage state updates across multiple nodes?
- **Computation:** can you parallelise program execution across the cluster?

Distributed Graph-Parallel Computation on Natural Graphs

Graph-Parallel?

Distributed Graph-Parallel Computation on Natural Graphs

Graph-Parallel? Paradigm that supports parallelism across a distributed graph

Write code from the perspective of a vertex; receive updates from your neighbours -> do something -> update neighbours with new state

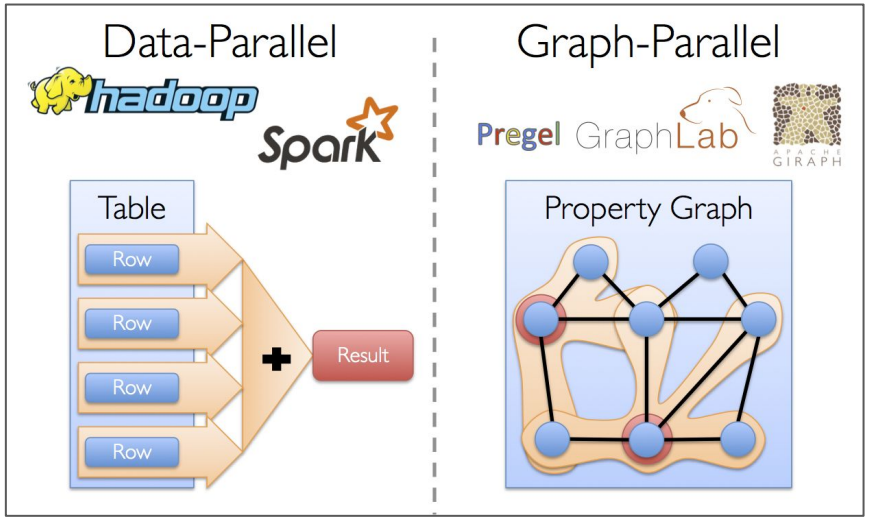


Figure from Apache Spark GraphX Programming Guide (v1.0.1)
https://downloads.apache.org/spark/docs/1.0.1/graphx-programming-guide.html

Why are graph-parallel paradigms necessary?

Data-parallel engines do not provide a natural idiom to express iterative workflows

For example, how would you express PageRank using Apache Spark?

- Shuffle-join every vertice with its neighbours
- Map a function to run over every vertex
- Reduce for every vertex over neighbours
- Materialize results
- Repeat…

Inefficient; ignores inherent structure of the data; unnatural programming model

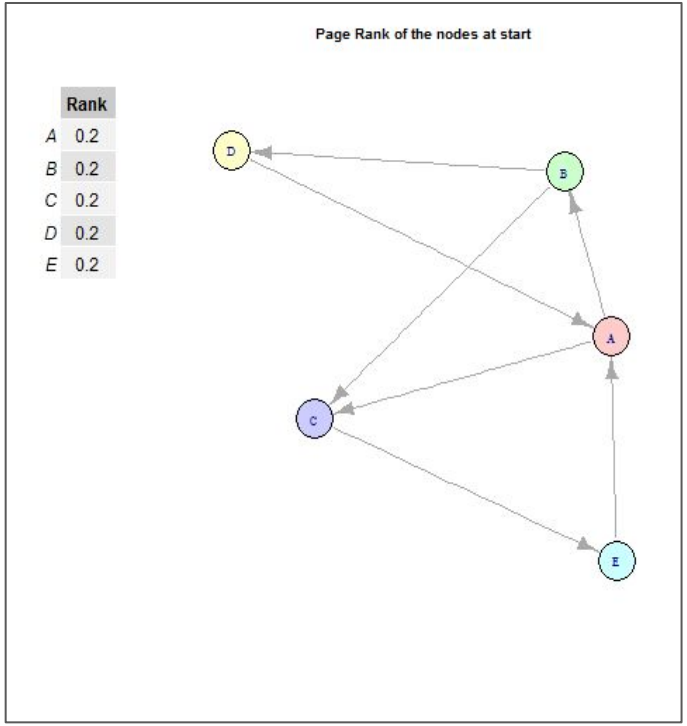


Figure from Dey, S. "Google Page Rank and the impact of the Second EigenValue..." 2017
https://sandipanweb.wordpress.com/2017/01/02/page-rank-and-power-iteration/

Why are graph-parallel paradigms necessary?

PageRank	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [22]	198s	–	1.1B	50x8
Spark [37]	97.4s	40M	1.5B	50x2
Twister [15]	36s	50M	1.4B	64x4
PowerGraph (Sync)	3.6s	40M	1.5B	64x8

Triangle Count	Runtime	$\|V\|$	$\|E\|$	System
Hadoop [36]	423m	40M	1.4B	1636x?
PowerGraph (Sync)	1.5m	40M	1.4B	64x16

LDA	Tok/sec	Topics	System
Smola et al. [34]	150M	1000	100x8
PowerGraph (Async)	110M	1000	64x16

Table 2: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

Distributed Graph-Parallel Computation on Natural Graphs

Natural Graphs?

Distributed Graph-Parallel Computation on Natural Graphs

Natural Graphs? PowerGraph argues: Real graphs often have ‘power-law’ degree distributions

A small number of vertices have very, very high degree: $\mathbf{P}(d) \propto d^{-\alpha}$

A small number of vertices are adjacent to a large fraction of edges, resulting in a *star-like* motif

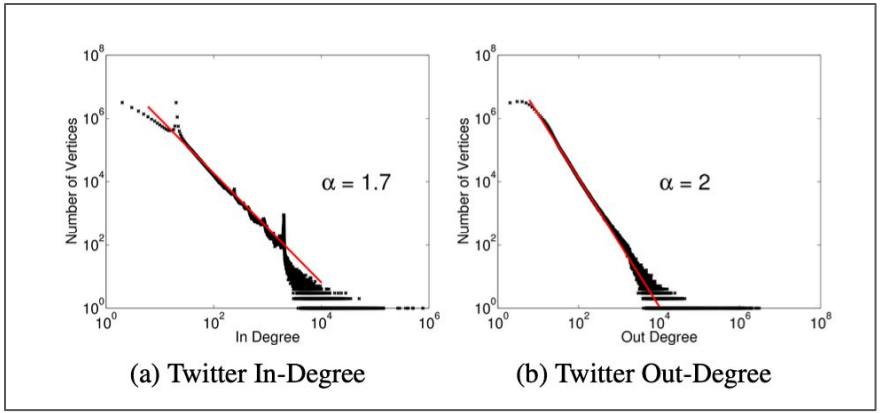


Figure 1: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

Context: Existing edge-cut ideas and their problems

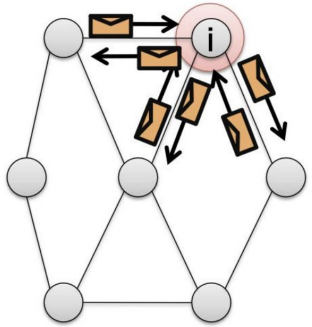
Pregel: BSP model with Message Passing (2010)

Bulk: all vertices process messages at the same time

Message Passing: neighbours communicate state updates by sending user-defined messages

Synchronous: every vertex waits for everyone else to send messages before progressing (*superstep*); global barriers ensures serializability

```
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);
```



Section 2.1: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012;
Figure from: Gonzalez et al., "PowerGraph" presentation slides, OSDI 2012
https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

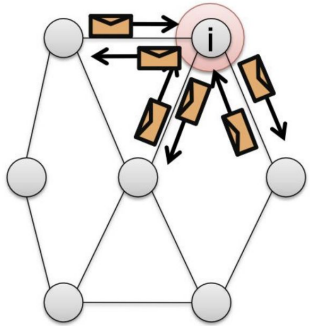
Pregel: BSP model with Message Passing (2010)

Bulk: all vertices process messages at the same time; wastes available resources

Message Passing: neighbours communicate state updates by sending user-defined messages; potentially lots of unnecessary messages; places onus on user to manage state updates

Synchronous: every vertex waits for everyone else to send messages before progressing; global barriers ensures serializability; everyone waits for slowest

```
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);
```



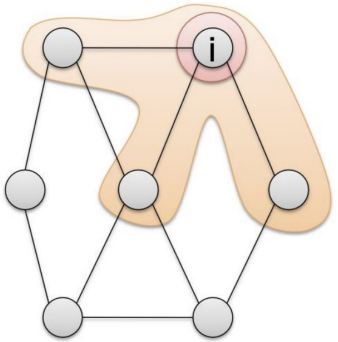
Section 2.1: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012;
Figure from: Gonzalez et al., "PowerGraph" presentation slides, OSDI 2012
https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

GraphLab: Asynchronous shared-memory model (2012)

Shared-Memory Abstraction: vertices access locally cached state (*ghost*) for scope; cache updates are abstracted from user, so no message passing needed

Asynchronous: no global barrier to vertex-program execution; users opt in to levels of serializability via scope locks

```
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;
```



Section 2.2: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012;
Figure from: Gonzalez et al., "PowerGraph" presentation slides, OSDI 2012
https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

Unsolved problem: High degree in natural graphs

Imagine a vertex with 1MM neighbours,
randomly partitioned across 1000 machines

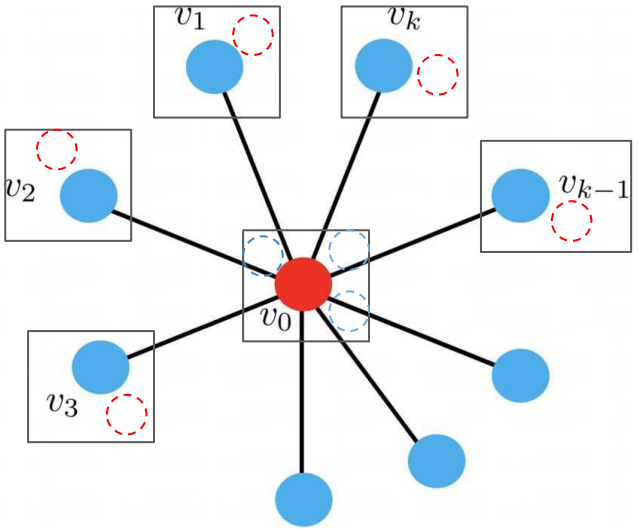


Figure adapted from Figure 1: Turab et al., “Existence of solutions for a class of non-boundary…”, Advances in Difference Equations 2021

Unsolved problem: High degree in natural graphs

Storage: Machine 1 has to store v0 + 1MM other ghosts

Communication: Vertex-program will need to acquire locks + broadcast updates + release locks across 1000 nodes to update v0 ghosts

Computation: No parallelism of vertex-program; sequential execution on single machine

Unsolved problem: High degree in natural graphs

When vertices have very high degree, GraphLab exhibits issues, because

1. edges are very likely to be cut, leading to ghost replication

2. performance drops linearly in number of replicas

For *p*=100 nodes, 99% chance edge cut

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

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

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

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

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

Proof. An edge is cut if both vertices are randomly assigned to different machines. The probability that both vertices are assigned to different machines is $1 - 1/p$. □

Theorem 5.1: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

Unsolved problem: High degree in natural graphs

“In order to address the challenges of natural graphs, the PowerGraph abstraction requires the size of the accumulator and the complexity of the apply function to be sub-linear in the degree. However, directly executing GraphLab and Pregel vertex-programs within the apply function leads the size of the accumulator and the complexity of the apply function to be linear in the degree eliminating many of the benefits on natural graphs.”

Section 4.4: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

PowerGraph: make performance sublinear in degree

Idea 1: Vertex-cut reduces effective replication

Assign edges to machines, allow vertices to span machines, with master <> read-replica pattern

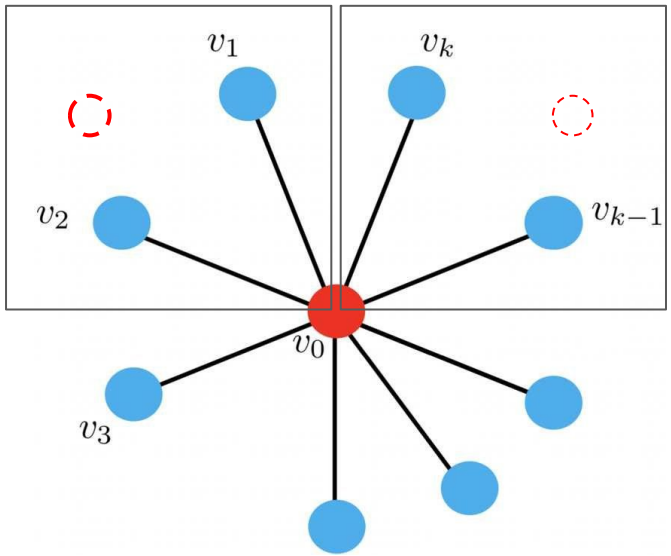


Figure adapted from Figure 1: Turab et al., “Existence of solutions for a class of non-boundary…”, Advances in Difference Equations 2021

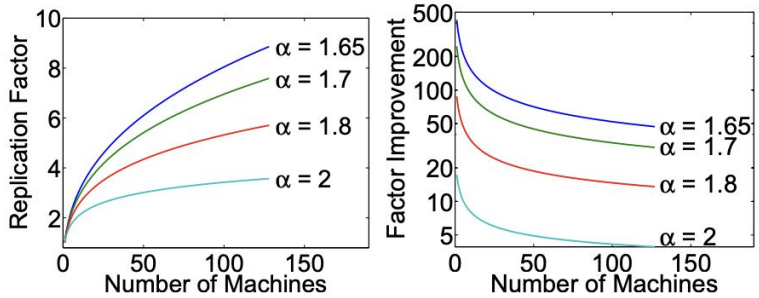
Idea 1: Vertex-cut reduces effective replication

Replication grows **much** slower than with edge-cuts, especially for small α (dense, hub-heavy graphs)

A(v) << |E|, independent of |V|

Storage overhead scales with smaller A(v), not |E|

- Don't need to store |E| ghosts
- Need to maintain A(v) replicas



(a) V-Sep. Bound (b) V-Sep. Improvement

Figure 6: **(a)** Expected replication factor for different power-law constants. **(b)** The ratio of the expected communication and storage cost of random edge cuts to random vertex cuts as a function of the number machines. This graph assumes that edge data and vertex data are the same size.

Figure 6 from Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs" 2012

Proof: Why is A(v) << |E|?

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

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

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

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

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

Proof. An edge is cut if both vertices are randomly assigned to different machines. The probability that both vertices are assigned to different machines is $1 - 1/p$. □

Theorem 5.1: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

For p=100, 99% chance of edge cut
=> all edges need ghosts

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

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

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:

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

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

Theorem 5.2: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

Something much less…

Idea 2: Vertex-cut localises data and compute

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

Figure 3: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

Idea 2: Vertex-cut localises data and compute

Computation is parallelisable

- *Gather* and *Scatter* can be performed locally on local subgraphs
- And in parallel across vertex-replicas containing different subgraphs

Communication overhead scales with smaller A(v), not |E|

- Cross-node communication only during *Apply*, to issue from master to mirrors; scales with A(v)

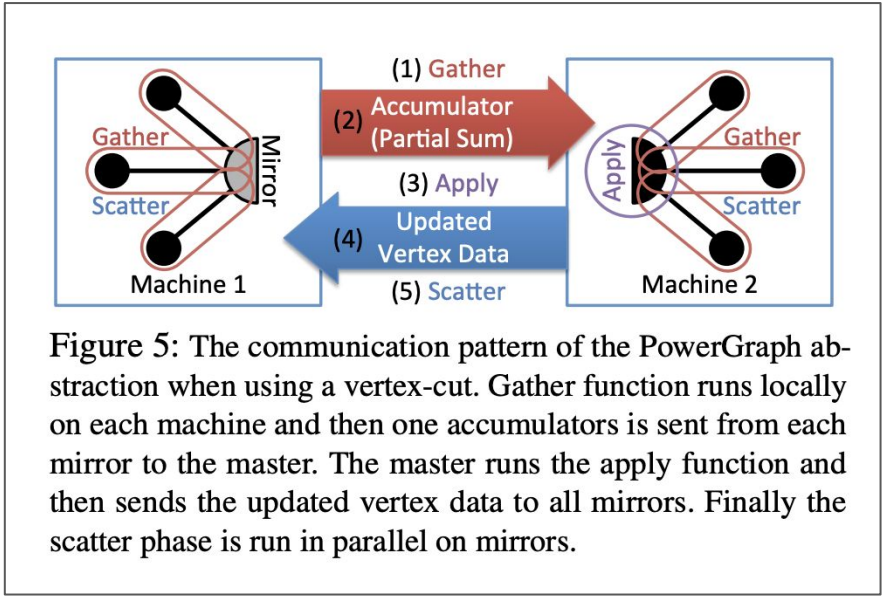


Figure 5: The communication pattern of the PowerGraph abstraction when using a vertex-cut. Gather function runs locally on each machine and then one accumulators is sent from each mirror to the master. The master runs the apply function and then sends the updated vertex data to all mirrors. Finally the scatter phase is run in parallel on mirrors.

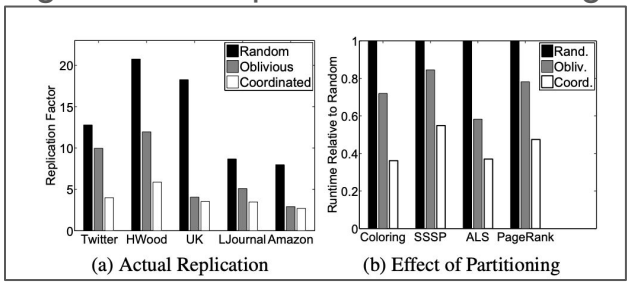
Figure 5: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

Implementation details

Programming model

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

Algorithms to optimise vertex cutting



(a) Actual Replication (b) Effect of Partitioning

Serializability vs concurrency modes

Bulk Synchronous (Sync): A fully synchronous implementation of PowerGraph as described in Sec. 4.3.1. [600 lines]

Asynchronous (Async): An asynchronous implementation of PowerGraph which allows arbitrary interleaving of vertex-programs Sec. 4.3.2. [900 lines]

Asynchronous Serializable (Async+S): An asynchronous implementation of PowerGraph which guarantees serializability of *all* vertex-programs (equivalent to "edge consistency" in GraphLab). [1600 lines]

Algorithms to minimise communication

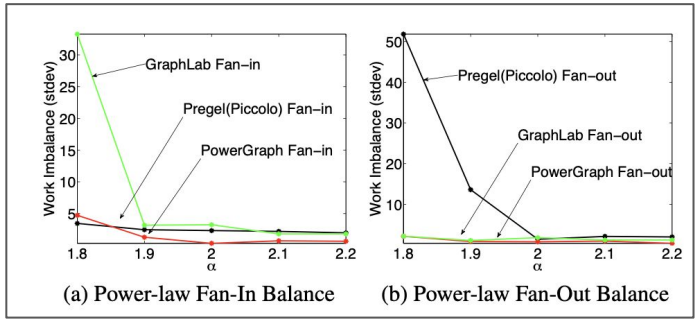
4.2 Delta Caching

In many cases a vertex-program will be triggered in response to a change in a *few* of its neighbors. The gather operation is then repeatedly invoked on *all* neighbors, many of which remain unchanged, thereby wasting computation cycles. For many algorithms [2] it is possible to dynamically maintain the result of the gather phase a_u and skip the gather on subsequent iterations.

Clockwise: Section 2.2; Figure 7; Section 7; Section 4.2: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

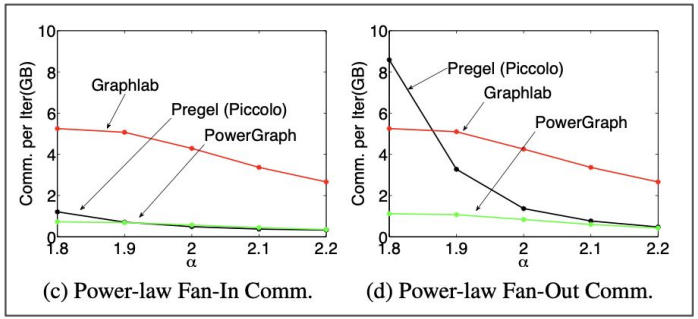
Comparison with graph-parallel systems: Synthetic data

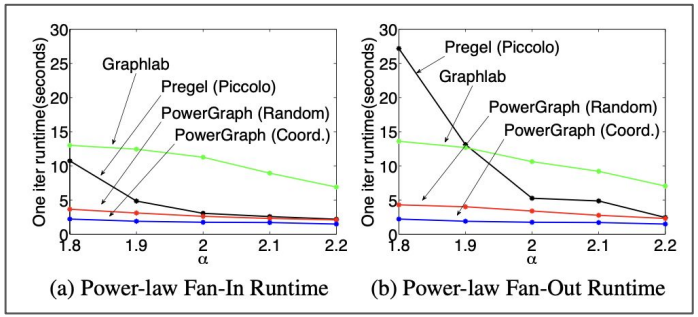
Worker Imbalance: PowerGraph worker performance is stable independent of degree



On synthetic data, the results follow the theory!
PowerGraph performs much better for low-α graphs than other graph-parallel systems.

Communication volume and Runtime: 5-6x improvements; note the similarity in the patterns





Clockwise: Figure 9a; Figure 9b; Figure 10: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

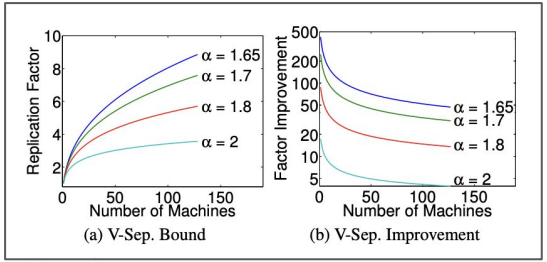
Opinion + Discussion

Elegant solution, but should you always use it?

PowerGraph proposes an elegant solution to the power-law problem it poses; the tradeoffs (e.g. vertex read-replicas) appear inevitable for a vertex-cut

However, as a user, you may have to decide:

1. Do you truly have a power-law graph?
   - Diminishing returns if not
2. Is the startup of an optimal vertex-cut worth it?
3. Is your graph dynamic?
   - New vertex-replicas for new edges?
4. Is storage a problem today?
   - Vertical scaling is cheap



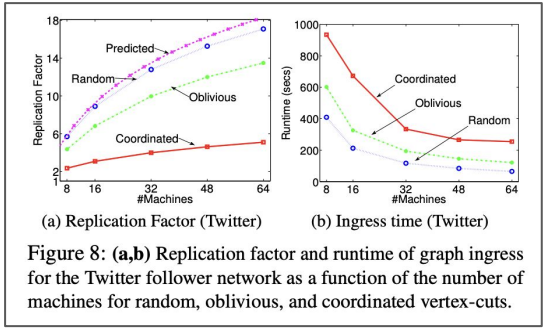


Figure 8: **(a,b)** Replication factor and runtime of graph ingress for the Twitter follower network as a function of the number of machines for random, oblivious, and coordinated vertex-cuts.

Figure 6; Figure 8: Gonzalez et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

References

[1] A. Turab, Z. D. Mitrović, and A. Savić, "Existence of solutions for a class of nonlinear boundary value problems on the hexasilinane graph," Adv. Differ. Equ., vol. 2021, no. 494, Nov. 2021, doi: 10.1186/s13662-021-03653-w.

[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in Proc. 10th USENIX Conf. Operating Systems Design and Implementation (OSDI), Hollywood, CA, USA, 2012, pp. 17–30.

[3] Apache Software Foundation, "GraphX programming guide," Apache Spark Documentation, Version 1.0.1, 2014. [Online]. Available: https://downloads.apache.org/spark/docs/1.0.1/graphx-programming-guide.html. [Accessed: Nov. 5, 2025].

[4] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," Proc. VLDB Endowment, vol. 5, no. 8, pp. 716–727, Apr. 2012, doi: 10.14778/2212351.2212354.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," presented at the 10th USENIX Conf. Operating Systems Design and Implementation (OSDI), Hollywood, CA, USA, Oct. 2012. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez. [Accessed: Nov. 5, 2025].

[6] S. Dey, "Page rank and power iteration," Sandipan's Machine Learning Blog, Jan. 2, 2017. [Online]. Available: https://sandipanweb.wordpress.com/2017/01/02/page-rank-and-power-iteration/. [Accessed: Nov. 5, 2025].