PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

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

OSDI'12

Presenter: Grant Wilkins (gfw27)

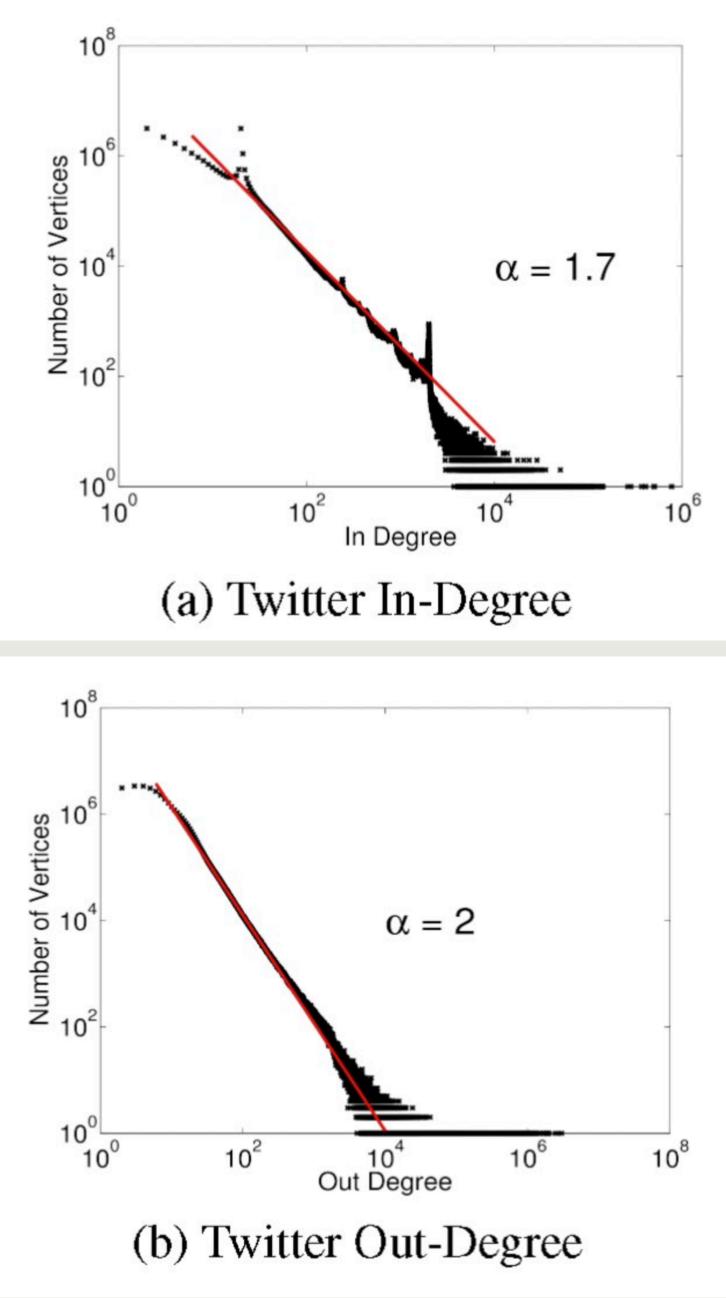
Situating this Study

- Large graph processing becoming more pressing due to growing social media networks, NLP,
- Pregel and GraphLab existing software for large-scale graph processing
- The problem(s): ***Power-law degree distribution***

Power-Law Distribution

Definition: Probability that vertex has degree d is $P(d) = d^{-\alpha}$ where α is skewness factor to control distribution.

Problem: When a few nodes have a lot of connections, they bottleneck typical systems.



(a) Twitter In-Degree

(b) Twitter Out-Degree

What this study aims to address?

- Work Balance
  - *Power-law throws off symmetric graph computation*
- Partitioning
  - *Hard to split up a natural graph*
- Communication
  - *Difficult to update skewed graphs*
- Storage
  - *High-degree vertices carry lots of memory*
- Computation
  - *Individual vertex computation doesn't scale*

Design of Powergraph

Gather, Apply, Scatter (GAS)

- D_u, D_v: vertex data (e.g. metadata & computation state)
- $D_{(u,v)}$: edge data between u, v
- Roughly same as GraphLab's implementation, but with parallel gather
- Very similar to Map-Reduce

```
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)
}
```

Delta Caching

Algorithm 1: Vertex-Program Execution Semantics

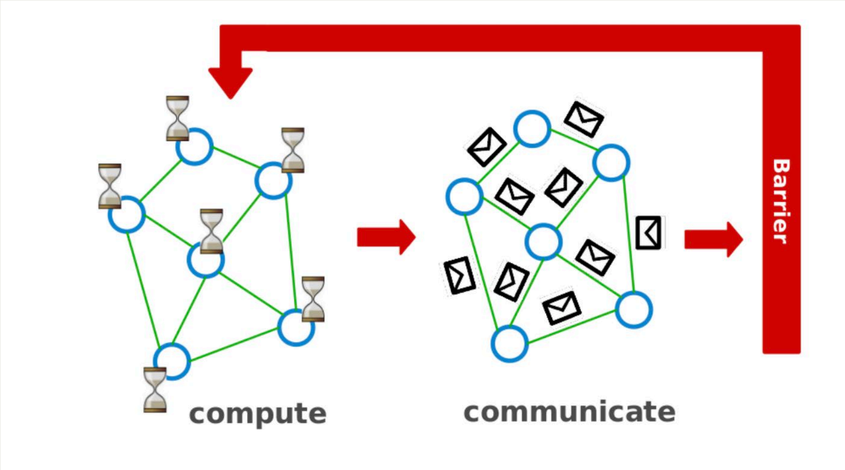
Input: Center vertex u

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

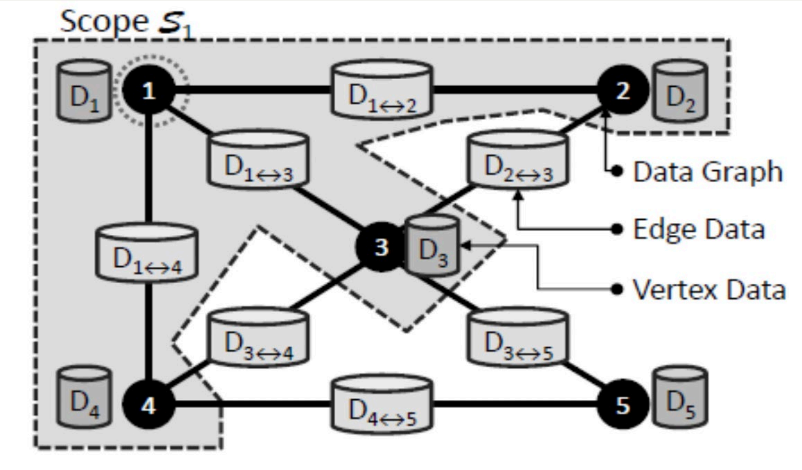
- Maintains cached accumulator at each vertex to avoid redundant gather operations.
  - Later results will show the advantage of keeping this, significant speedup
- The scatter phase can return Δa which gets added to the neighbor's accumulator, incrementally updating it.

Synchronous and Asynchronous Execution Model

- Synchronous schedules like Pregel. Executes GAS and commits at end.
- Asynchronous schedules like GraphLab. Changes occur instantaneously during *apply* and *scatter*.



Pregel: Synchronous Model



GraphLab: Asynchronous Model

Example Implementation

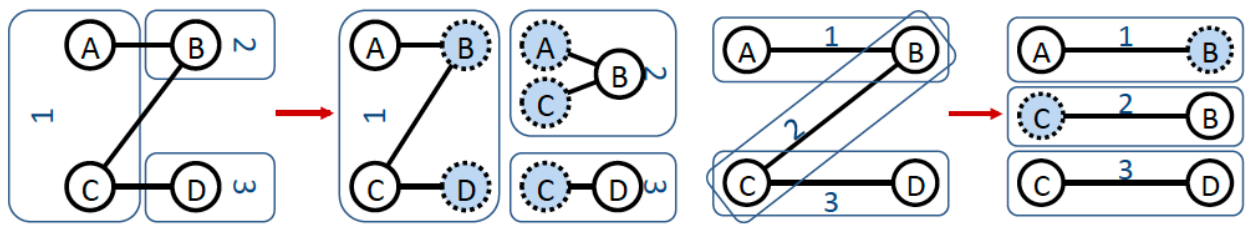
PageRank

```
// 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
```

Powergraph on Distributed Systems

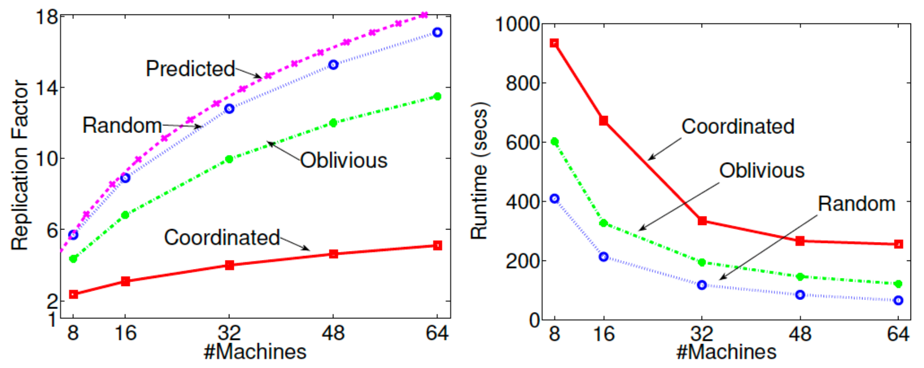
Edge vs. Vertex Partitioning

- PowerGraph uses vertex-cutting!
- Increases replication of vertices, lowers copies of edges.
- Think about power distributed graphs, and how much data replicating edges would cost



(a) Edge-Cut (b) Vertex-Cut

Random vs. Greedy Partitioning



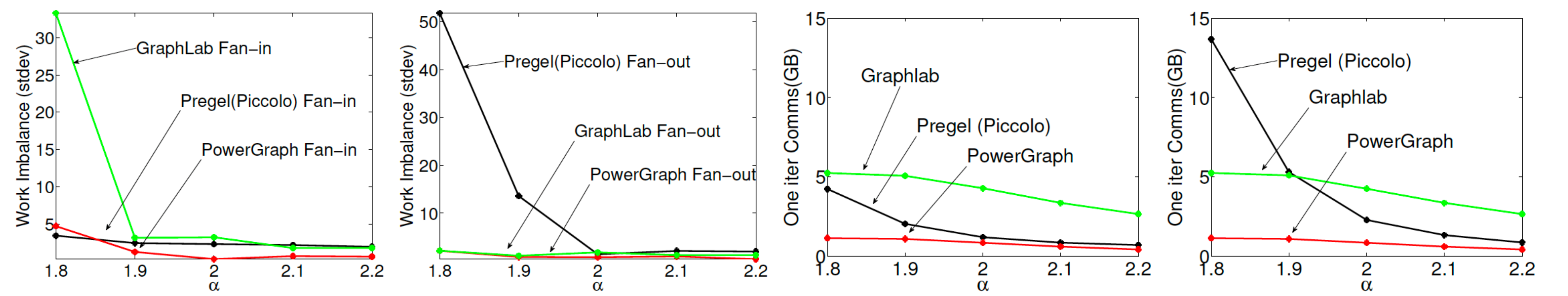
(a) Replication Factor (Twitter) (b) Ingress time (Twitter)

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.

- *Random:* Randomize where you put vertices
- *Greedy:* do a minimization problem of expected number of replications
  - Coordinated: maintains a shared table
  - Oblivious: maintains a local model of data

How does PowerGraph actually perform?

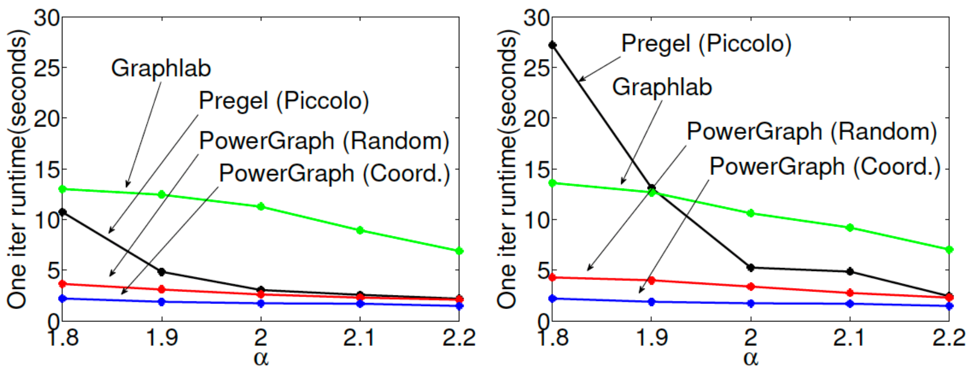
Finding: PowerGraph maintains constant behavior despite skewness factor α



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

Std. dev. of worker computation time

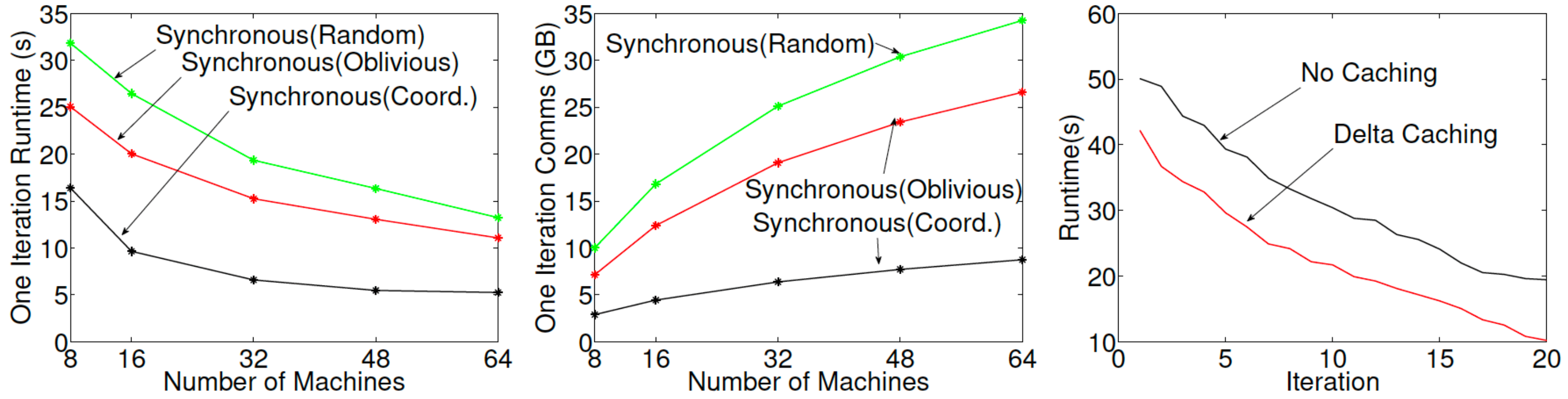
Average info communicated



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

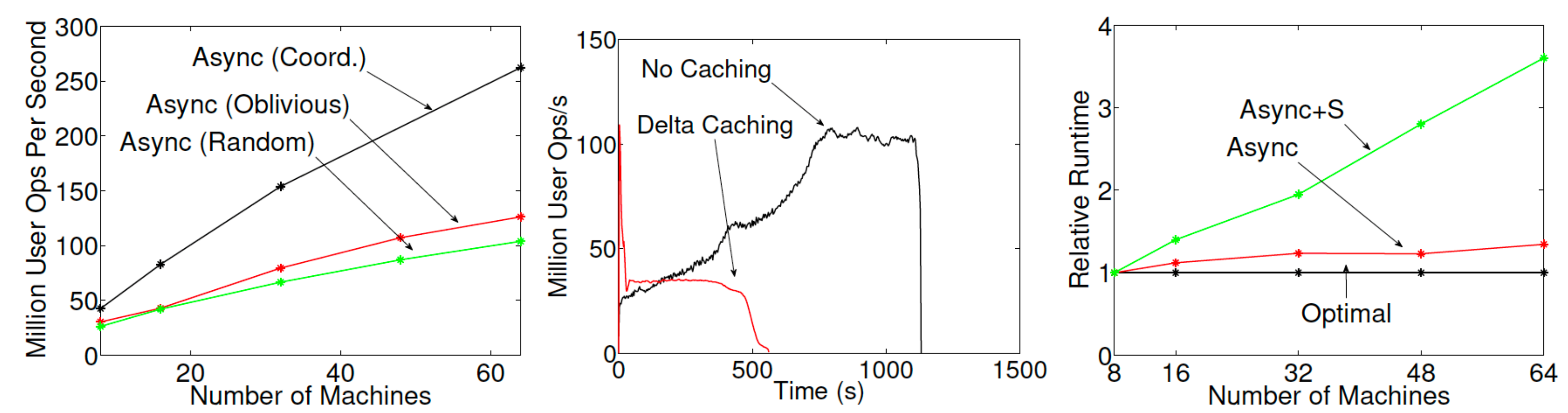
Average Runtime

Finding: PowerGraph's synchronous engine exhibits
(a) good strong scalability
(b) reduces memory overhead with greedy partitioning
(c) saves time using delta caching



(a) Twitter PageRank Runtime (b) Twitter PageRank Comms (c) Twitter PageRank Delta Cache

Finding: PowerGraph's asynchronous engine exhibits
(a) nearly linear throughput increase with machine
(b) reduces operations with caching
(c) nearly linear weak-scaling



(a) Twitter PageRank Throughput (b) Twitter PageRank Delta Cache (c) Coloring Weak Scaling

“Performance” of PowerGraph against competing software

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

My critique

Cons:

- Comparison against other work could be better
- Use of consistent metrics in evaluation
- Consistent comparison between sync and async and async+serialization
- More careful mathematical text

Pros:

- Great motivating concept
- Very good theoretical basis for the results
- Melds two existing models together and then extends to create
- Was successful enough to get acquired by Apple

Questions?