PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

Authors: Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin
Power-Law Graphs

- **Degree** of vertex is number of edges attached to vertex
- Real-world graphs have **highly-skewed, power-law degree distribution**
  - Most vertices have few edges
  - **Some vertices have many edges** (long-tailed distribution)
Understanding Power-Law Graphs

- Probability that a vertex has degree $d$: $P(d) \propto 1/d^a$
  - $a$ is a constant, $a > 0$, typical value is 2
  - $a \downarrow \Rightarrow$ skew (vertices with high degree) $\uparrow$
    - density (#edges/#vertices) $\uparrow$

- Intuition
  - Say $d = 100$, and it goes up by 1
  - $P(101)/P(100) = 100^2/101^2 = 0.98$ (close to 1)
    - So, significant probability of high degree vertices
  - Compare with exponential distribution: $P(d) \propto 1/e^d$
    - $P(101)/P(100) = 1/e = 0.37$, low probability of high degree vertices
Power-Law Degree Distribution

More than $10^8$ vertices have one neighbor

Over 10 million vertices have one neighbor

Top 1% of vertices are adjacent to 50% of edges!

AltaVista WebGraph
1.4B Vertices, 6.6B Edges

High-degree vertices

Star-like Motif

Obama

Followers
Why PowerGraph?

- Power-law graphs are hard to partition well
- Distributed graph computation systems perform poorly on such graphs
  - Hard to balance computation and storage load
  - Significant communication
The Graph-Parallel Abstraction

• A user-defined vertex program runs on each vertex
• Graph constrains interaction along edges
  • Using messages, e.g., Pregel
  • Using shared memory, e.g., GraphLab
• Parallelism: run multiple vertex programs concurrently
The Pregel Abstraction

Vertex programs interact by sending messages

```
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] = 0.15 + total
  // Send new messages to neighbors
  foreach (j in out_neighbors[i]) :
    Send msg(R[i] * w_{ij}) to vertex j
```
The GraphLab Abstraction

Vertex programs directly **read** the neighbor’s state

```
GraphLab_PageRank(i)
// Compute sum over neighbors
  total = 0
  foreach (j in in_neighbors(i)):
    total = total + R[j] * w_{ji}
// Update the PageRank
  R[i] = 0.15 + total
// Trigger neighbors to run again
  if R[i] not converged then
    foreach (j in out_neighbors(i)):
      signal vertex-program on j
```
Challenges of High-Degree Vertices

- Sequentially process edges
- Sends many messages (Pregel)
- Touches a large fraction of graph (GraphLab)
- Edge metadata too large for single machine
- Synchronous execution prone to stragglers (Pregel)
- Asynchronous execution needs heavy locking (GraphLab)
Key Idea in PowerGraph

- Split high-degree vertices
  - Parallelize processing of high-degree vertices
  - Guarantee split and non-split vertices operate equivalently

Program on This

Run on This

Machine 1

Machine 2
How to Split Vertex Processing?

Insight: each vertex program consists of three steps

```javascript
GraphLab_PageRank(i)

// Compute sum over neighbors
total = 0
foreach (j in in_neighbors(i)):
  total = total + R[j] * w_{ji}

// Update the PageRank
R[i] = 0.15 + total

// Trigger neighbors to run again
if R[i] not converged then
  foreach (j in out_neighbors(i)):
    signal vertex-program on j
```

Gather information from neighbors

Update vertex

Signal neighbors & modify edge data
How to Split Vertex Processing?

Work is proportional to vertex degree in **first, third** steps

```java
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] = 0.15 + total

// Trigger neighbors to run again
if R[i] not converged then
    foreach (j in out_neighbors(i)):
        signal vertex-program on j
```

Gather information from neighbors

Update vertex

Signal neighbors & update edge data

Work is proportional to vertex degree in **first, third** steps
PowerGraph GAS Abstraction

- PowerGraph splits the 3 steps of vertex processing into:
  - Gather (gather information from neighbors)
  - Apply (update vertex)
  - Scatter (signal neighbors, update edge data)
- Enables parallelizing Gather and Scatter phases by moving computation to the data
GAS Abstraction

- Split a high-degree vertex across multiple machines
- Mark one a master, rest are mirrors
GAS Abstraction

- Run **Gather** on all edges, in parallel on the machines
GAS Abstraction

- Run **Gather** on all edges, in parallel on the machines
  - Send **partial sum** from mirrors to master
    - Similar to Pregel Combiners
GAS Abstraction

- Run **Gather** on all edges, in parallel on the machines
  - Send partial sum from mirrors to master
- Apply update based on partial sums on master vertex
GAS Abstraction

• Run **Gather** on all edges, in parallel on the machines
  • Send partial sum from mirrors to master
• **Apply** update based on partial sums on master vertex
  • Sent vertex update to mirrors
GAS Abstraction

- Run **Gather** on all edges, in parallel on the machines
  - Send partial sum from mirrors to master
- **Apply** update based on partial sums on master vertex
  - Sent vertex update to mirrors
- Run **Scatter** on all edges, in parallel on the machines
PageRank in PowerGraph

Gather and Scatter operate on single edge, not all edges

```java
PowerGraph_PageRank(i)

// Compute sum over neighbors
Gather(j->i): return R[j] * w_{ji}
sum(a, b) = a + b:

// Update the PageRank
Apply(I, Σ): R[i] = 0.15 + Σ

// Trigger neighbors to run again
Scatter(i->j):
if R[i] not converged then
    signal vertex-program on j
```
Graph Partitioning

• GAS model spreads processing load for high-degree vertices by splitting them across machines

• This approach enables a new method of graph partitioning called vertex cut
  • Assign each edge to a machine
  • A vertex may span machines

• For power-law graph, vertex cuts help:
  • Improve load balancing
  • Reduce communication and storage overhead
Edge Cuts versus Vertex Cuts

**Edge Cut**
- Must synchronize many edges

**Vertex Cut**
- Must synchronize a single vertex

Diagram showing the comparison between edge cuts and vertex cuts.
Constructing Vertex Cuts

• **Goals**
  - Evenly assign edges to machines
  - Minimize number of machines spanned by each vertex
  - Assign each edge as it is loaded, without reassigning it again

• **Three distributed approaches**
  - Random edge placement
  - Coordinated greedy edge placement
    - Place an edge on a machine that already has vertices of that edge
    - Requires coordination to track current vertex->machine assignment
  - Oblivious greedy edge placement
    - Same as above, but use local approximation of vertex->machine assignment, so no coordination required
Synchronization

- PowerGraph supports three execution modes:
  - Synchronous
    - Each of the GAS phases run in bulk-synchronous model
  - Asynchronous
    - The GAS phases run completely asynchronously
  - Asynchronous + Serializable
    - Neighboring vertices do not run simultaneously
    - Similar to Dining Philosopher problem
Comparison with Pregel & GraphLab

PageRank on Synthetic Power-Law Graphs
Partitioning Cost

Twitter Graph: 41M vertices, 1.4B edges
Performance With Different Partitioning Schemes

![Bar Chart]

- **PageRank**
  - Random
  - Oblivious
  - Coordinated

- **Collaborative Filtering**
  - Random
  - Oblivious

- **Shortest Path**
  - Random
  - Oblivious

Runtime Relative to Random
Conclusions

• Real-world graphs are power-law graphs
• Computation on power-law graph is challenging
  • High-degree vertices
  • Low-quality edge-cuts
• PowerGraph proposes 1) GAS decomposition model for splitting and parallelizing vertex programs, 2) vertex cuts for partitioning power-law graphs
• PowerGraph theoretically and experimentally outperforms Pregel and GraphLab
• PowerGraph is available as Apache GraphLab 2.1
Discussion
Q1

- What problems do power-law distributions cause for graph processing?
Q2

• Why do vertex cuts make it easier to perform load balancing compared to edge cuts?
Powergraph splits processing into three steps:

- gather + sum
- apply
- scatter

Assuming a node X has four neighbors A, B, C, D, the **sum** operation is performed as follows:

- \text{sum}(\text{gather}(A), \text{gather}(B), \text{gather}(C), \text{gather}(D))

Why does the **sum** operation need to be commutative and associative?
Q4

- PowerGraph can accelerate the gather phase by caching partial sums for each vertex during the scatter phase.
- Why would this caching be beneficial?