PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs

J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin
OSDI 2012
Background

1. **Why Graph?**
   Machine Learning and Data Mining (MLDM) problems represented as graphs ----> Graph structured computation is important

2. Current distributed graph computation systems
   - Pregel
   - GraphLab

3. **Problems:** Existing distributed graph computation systems perform poorly on Natural Graphs.
Challenges of Natural Graphs

Many Graphs have skewed degree distribution: a small fraction of the vertices are adjacent to a large fraction of the edges.

How do we *program* graph computation?

“Think like a Vertex.”

-Malewicz et al. [SIGMOD’10]
The **Graph-Parallel** Abstraction

1. A user-defined **Vertex-Program** runs on each vertex
2. **Graph** constrains **interaction** along edges
   - Using **messages** (e.g. Pregel [PODC’09, SIGMOD’10])
   - Through **shared state** (e.g., GraphLab [UAI’10, VLDB’12])
3. **Parallelism**: run multiple vertex programs simultaneously
PageRank Algorithm

\[ R[i] = 0.15 + \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j] \]

- Update ranks in parallel
- Iterate until convergence
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] * wij) to vertex j
```

Malewicz et al. [PODC’09, SIGMOD’10]
The GraphLab Abstraction

Vertex-Programs directly **read** the neighbors state

```plaintext
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
```

Low et al. [UAI’10, VLDB’12]
Problem: Challenges of High-Degree Vertices

GraphLab and Pregel on Natural Graphs:
1. Work Imbalance
2. Partitioning
3. Communication
4. Storage
5. Computation

Source: Joseph Gonzalez. (2012)
Problem: Random Partitioning

- Both GraphLab and Pregel resort to random (hashed) partitioning on natural graphs

\[ E \left[ \frac{|\text{Edges Cut}|}{|E|} \right] = 1 - \frac{1}{p} \]

10 Machines \(\rightarrow\) 90% of edges cut
100 Machines \(\rightarrow\) 99% of edges cut!

Source: Joseph Gonzalez. (2012)
In Summary

**GraphLab** and **Pregel** are not well suited for **natural graphs**

- Challenges of **high-degree vertices**
- Low quality **partitioning**
PowerGraph

• **GAS Decomposition:** distribute vertex-programs
  – Move computation to data
  – Parallelize high-degree vertices

• **Vertex Partitioning:**
  – Effectively distribute large power-law graphs
A Common Pattern for Vertex-Programs

```python
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] = 0.1 + 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 About Neighborhood
Update Vertex
Signal Neighbors & Modify Edge Data

GAS Decomposition

**Gather (Reduce)**
Accumulate information about neighborhood

*User Defined:*
- \( \text{Gather}(V) \rightarrow \Sigma \)
- \( \Sigma_1 + \Sigma_2 \rightarrow \Sigma_3 \)

**Apply**
Apply the accumulated value to center vertex

*User Defined:*
- \( \text{Apply}(Y, \Sigma) \rightarrow Y' \)

**Scatter**
Update adjacent edges and vertices.

*User Defined:*
- \( \text{Scatter}(V) \rightarrow Y' \)

Parallel Sum
\[ V + V + ... + V \rightarrow \Sigma \]

Update Edge Data & Activate Neighbors

Source: Joseph Gonzalez. (2012)
PageRank in PowerGraph

\[ R[i] = 0.15 + \sum_{j \in \text{Nbrs}(i)} w_{ji} R[j] \]

**PowerGraph\_PageRank(i)**

- **Gather\(( j \rightarrow i )\)**: return \( w_{ji} \times R[j] \)
- **sum(a, b)**: return \( a + b \)

**Apply\((i, \Sigma)\)**: \( R[i] = 0.15 + \Sigma \)

**Scatter\((i \rightarrow j)\)**:
  - if \( R[i] \) changed then trigger \( j \) to be **recomputed**

Edge Cut vs Vertex Cut

Two methods for partitioning the graph in a distributed environment:
- Edge Cut (Used by Pregel and GraphLab abstractions)
- Vertex Cut (Used by PowerGraph)

Figure 4: (a) An edge-cut and (b) vertex-cut of a graph into three parts. Shaded vertices are ghosts and mirrors respectively.
New Approach to Partitioning

• Rather than cut edges:

• We cut vertices

Source: Joseph Gonzalez, (2012)
New Approach to Partitioning

New Theorem:

For any edge-cut we can directly construct a vertex-cut which requires strictly less communication and storage.

Constructing Vertex-Cuts

- **Evenly** assign *edges* to machines
  - Minimize machines spanned by each vertex

- Assign each edge **as it is loaded**
  - Touch each edge only once

- Propose three **distributed** approaches:
  - *Random* Edge Placement
  - *Coordinated Greedy* Edge Placement
  - *Oblivious Greedy* Edge Placement
**Random Edge-Placement**

- Randomly assign edges to machines

Source: [Joseph Gonzalez (2012)](Joseph Gonzalez (2012))
Analysis Random Edge-Placement

- Expected number of machines spanned by a vertex:

Twitter Follower Graph
41 Million Vertices
1.4 Billion Edges

Accurately Estimate Memory and Comm. Overhead

Source: Joseph Gonzalez, (2012)
Random Vertex-Cuts vs. Edge-Cuts

- Expected improvement from vertex-cuts:

![Graph showing reduction in communication and storage with increasing number of machines.](source: Joseph Gonzalez, (2012))
Greedy Vertex-Cuts

- Place edges on machines which already have the vertices in that edge.
Greedy Vertex-Cuts

- **De-randomization** → greedily minimizes the expected number of machines spanned

- **Coordinated** Edge Placement
  - Requires coordination to place each edge
  - Slower: higher quality cuts

- **Oblivious** Edge Placement
  - Approx. greedy objective without coordination
  - Faster: lower quality cuts
Partitioning Performance

**Twitter Graph:** 41M vertices, 1.4B edges

- **Cost**
  - Random
  - Oblivious
  - Coordinated

- **Construction Time**
  - Oblivious
  - Coordinated

*Oblivious* balances cost and partitioning time.

Source: Joseph Gonzalez, (2012)
Greedy Vertex-Cuts Improve Performance

![Bar chart showing runtime relative to random for PageRank, Collaborative Filtering, and Shortest Path algorithms. The chart compares Random, Oblivious, and Coordinated methods.]

Greedy partitioning improves computation performance.

Other Features (See Paper)

- Supports three execution modes:
  - **Synchronous**: Bulk-Synchronous GAS Phases
  - **Asynchronous**: Interleave GAS Phases
  - **Asynchronous + Serializable**: Neighboring vertices do not run simultaneously

- Delta Caching
  - Accelerate gather phase by **caching** partial sums for each vertex
System Design

- Implemented as C++ API
- Uses HDFS for Graph Input and Output
- Fault-tolerance is achieved by check-pointing
  - Snapshot time < 5 seconds for twitter network

Implemented Many Algorithms

- Collaborative Filtering
  - Alternating Least Squares
  - Stochastic Gradient Descent
  - SVD
  - Non-negative MF

- Statistical Inference
  - Loopy Belief Propagation
  - Max-Product Linear Programs
  - Gibbs Sampling

- Graph Analytics
  - PageRank
  - Triangle Counting
  - Shortest Path
  - Graph Coloring
  - K-core Decomposition

- Computer Vision
  - Image stitching

- Language Modeling
  - LDA
Comparison with GraphLab & Pregel

- PageRank on Synthetic Power-Law Graphs:

  ![Graph showing communication and runtime comparison between GraphLab, Pregel, and PowerGraph across different power-law constants.](image)

  - **Communication:**
    - Pregel (Piccolo)
    - GraphLab
    - PowerGraph
  - **Runtime:**
    - Pregel (Piccolo)
    - GraphLab
    - PowerGraph

  High-degree vertices

PowerGraph is robust to **high-degree** vertices.

PageRank on the Twitter Follower Graph

Natural Graph with 40M Users, 1.4 Billion Links

Communication

Runtime

Reduces Communication

Runs Faster

PowerGraph is Scalable

Yahoo Altavista Web Graph (2002):
One of the largest publicly available web graphs
1.4 Billion Webpages, 6.6 Billion Links

7 Seconds per Iter.
1B links processed per second
30 lines of user code
Summary

- **Problem**: Computation on **Natural Graphs** is challenging
  - High-degree vertices
  - Low-quality edge-cuts

- **Solution**: **PowerGraph System**
  - **GAS Decomposition**: split vertex programs
  - **Vertex-partitioning**: distribute natural graphs

- PowerGraph **theoretically** and **experimentally** outperforms existing graph-parallel systems.
Criticism - pros/cons

- PowerGraph, uses intelligent partitioning of vertices across servers. While this pre-processing reduces per iteration runtime, it is an expensive step by itself. [1]
- High Memory: store in, edges, mirror values
- Out-of-core storage: Support graphs that don’t fit in memory (GraphChi)
- Maintenance of vertex replicas, communication-bound apply phase in the GAS abstraction [2]
<table>
<thead>
<tr>
<th>Goal</th>
<th>PowerGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>2 passes</td>
</tr>
<tr>
<td>Communication</td>
<td>$\propto #\text{vertex mirrors}$</td>
</tr>
<tr>
<td>Pre-Processing</td>
<td>Expensive (Intelligent)</td>
</tr>
<tr>
<td>Memory</td>
<td>High (store in- and out-edges + mirror values)</td>
</tr>
<tr>
<td>Scalability</td>
<td>Good but needs a min $#\text{servers}$</td>
</tr>
</tbody>
</table>
References
