Graph-parallel Computation

Rong Chen

Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University
http://ipads.se.sjtu.edu.cn/

Credits: some slides from Joseph Gonzalez (CMU) and Yucheng Low (CMU)
Big Data Everywhere

- 1.11 Billion Tweets/year
- 6 Billion Photos
- 100 Hrs of Video every minute
- 400 Million Tweets/day

How do we understand and use Big Data?
Big Data → Big Learning

1.11 Billion Tweets/day

6 Billion Photos

100 Hrs of Video every minute

400 Million Tweets/day

Big Learning: machine learning and data mining on Big Data
Label a Face and Propagate

From GraphLab and Carlos Guestrin
Pairwise Similarity not Enough

From GraphLab and Carlos Guestrin
Probabilistic Graphical Models

Similarity edges

c-o-occurring faces

further evidence

Propagate Similarities & Co-occurrences for Accurate Predictions

From GraphLab and Carlos Guestrin
Collaborative Filtering

Latent Factor Models
Non-negative Matrix Factorization

What do I recommend???

Red sorghum
The story of Qiujin
The flowers of war
Coming home
Example: Collaborative Filtering

Alternating Least Squares (ALS)

\[ u_i = \text{arg min}_w \sum_{j \in N[i]} (r_{ij} - m_j \cdot w)^2 \]

\[ m_j = \text{arg min}_w \sum_{i \in N[j]} (r_{ij} - u_i \cdot w)^2 \]

Alternatively iterate until convergence

http://dl.acm.org/citation.cfm?id=1424269
Example: PageRank

What's the rank of this webpage?

PowerLyra

GraphLab

Imitator

Wikipedia: graph

Depends on rank of who links them ...

Loops in graph → Must iterate!

Depends on rank of who links it
Example: **PageRank**

Iterate until convergence:

“**My rank is weighted average of my neighbors’ ranks**”

\[ R[i] = \alpha + (1 - \alpha) \sum_{(j, i) \in E} w_{ji} R[j] \]

- \( \alpha \) is the random reset probability
- \( W_{ij} \) is the probability of transitioning (similarity) from j to i
Example: PageRank

PageRank (Centrality Measures)

\[ R[i] = \alpha + (1 - \alpha) \sum_{(j,i) \in E} \frac{1}{L[j]} R[j] \]

\( \alpha \) is the random reset probability
\( L[j] \) is the number of links on page \( j \)

iterate until convergence

example:

\[ R[1] = 0.15 + 0.85 \left( R[3] + \frac{1}{3} R[4] + \frac{1}{2} R[5] \right) \]
Example Algorithms

Collaborative Filtering
- Alternating Least Squares
- Stochastic Gradient Descent
- Tensor Factorization

Graph Analytics
- PageRank
- SSSP
- Triangle-Counting
- Graph Coloring
- K-core Decomposition

Classification
- Neural Networks
- Lasso

Semi-supervised ML
- Graph SSL
- CoEM

Structured Prediction
- Loopy Belief Propagation
- Max-Product Linear Programs
- Gibbs Sampling
It’s all about the graphs
It is cumbersome to write graph algorithms in data-parallel models (e.g., MapReduce)

1. difficult to express dependent data (stateless)
2. hard to support iterative computation
3. don’t allow point to point communication
...
Graph Algorithms

Dependent Data

Local Accesses

Iterative Computation

“Think as a Vertex”
Think as a **Vertex**

**Algorithm** ➔ Impl. `compute()` for vertex

1. **aggregate** value of neighbors
2. **update** itself value
3. **activate** neighbors

Example: **PageRank**

\[
R[i] = \alpha + (1 - \alpha) \sum_{(j,i) \in E} \frac{1}{L[j]} R[j]
\]

```
compute():
    double sum = 0;
    double value, last = get();
    foreach (n in nbrs)
        sum += n.value / n.nedges;
    value = 0.15 + 0.85 * sum;
    set(value);
    activate(nbrs);
```
Think as a **Vertex**

**Algorithm** ➔ Impl. `compute()` for vertex

**Computation** ➔ Compute all Vertex in parallel

1. **synchronous** scheduling

2. **asynchronous** scheduling
Think as a **Vertex**

**Algorithm** → Impl. `compute()` for vertex

**Computation** → Compute all **Vertex** in parallel

**Communication** → Access data through **edge**

1. **message passing**  
2. **shared memory**

![Diagram](image)
Pioneer Systems

Pregel
Google (2009)
- synchronous
- message passing

GraphLab
CMU (2010)
- asynchronous
- shared memory
Natural Graphs
Natural Graphs
Graphs derived from natural phenomena
Problem:

Existing *distributed* graph computation systems perform poorly on **Natural Graphs**.
PageRank on Twitter Follower Graph

Natural Graph with 40M Users, 1.4 Billion Links

Runtime Per Iteration

- Hadoop
- GraphLab
- Twister
- Piccolo

Order of magnitude by exploiting properties of Natural Graphs

Hadoop results from [Kang et al. ’11]
Twister (in-memory MapReduce) [Ekanayake et al. ‘10]
Properties of Natural Graphs

Power-Law Degree Distribution
Power-Law Degree Distribution

More than $10^8$ vertices have one neighbor.

High-Degree Vertices

Altavista WebGraph
1.4B Vertices, 6.6B Edges
Power-Law Degree Distribution

“Star Like” Motif

President Obama

Followers
Power-Law Graphs are Difficult to Partition

- Power-Law graphs do not have low-cost balanced cuts [Leskovec et al. 08, Lang 04]
- Traditional graph-partitioning algorithms perform poorly on Power-Law Graphs. [Abou-Rjeili et al. 06]
Properties of Natural Graphs

High-degree Vertices

Low Quality Partition

Power-Law Degree Distribution
• Split High-Degree vertices
• New Abstraction $\rightarrow$ **Equivalence** on Split Vertices
How do we *program* graph computation?

“Think like a Vertex.”

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

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

What’s the popularity of this user?

Popular?

Depends on the popularity of her followers

Depends on the popularity of their followers
PageRank Algorithm

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

- Update ranks in parallel
- Iterate until convergence
The Pregel Abstraction

Vertex-Programs interact by sending 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
```

Malewicz et al. [PODC’09, SIGMOD’10]
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

- Asynchronous Execution requires heavy locking (GraphLab)
- Synchronous Execution prone to stragglers (Pregel)

Sequentially process edges

Sends many messages (Pregel)

Touches a large fraction of graph (GraphLab)

Edge meta-data too large for single machine
Communication Overhead for High-Degree Vertices

Fan-In vs. Fan-Out
• User defined **commutative associative** (+) message operation:
Pregel Struggles with **Fan-Out**

- **Broadcast** sends many copies of the same message to the same machine!
Fan-In and Fan-Out Performance

- PageRank on synthetic Power-Law Graphs
  - Piccolo was used to simulate Pregel with combiners

![Graph showing total communication vs. power-law constant α]

More high-degree vertices
GraphLab Ghosting

• Changes to master are synced to ghosts
GraphLab Ghosting

- Changes to neighbors of high degree vertices creates substantial network traffic
Fan-In and Fan-Out Performance

- PageRank on synthetic Power-Law Graphs
- GraphLab is **undirected**

![Graph showing performance comparison between Pregel and GraphLab](image)
Graph Partitioning

• Graph parallel abstractions rely on partitioning:
  – Minimize communication
  – Balance computation and storage

Machine 1

Data transmitted across network $O(\# \text{ cut edges})$

Machine 2
Random Partitioning

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

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

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

**GraphLab_PageRank(i)**

```plaintext
// 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:*
- $\text{Gather}(Y) \rightarrow \Sigma$
- $\Sigma_1 + \Sigma_2 \rightarrow \Sigma_3$

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

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

**Scatter**
Update adjacent edges and vertices.

*User Defined:*
- $\text{Scatter}(Y')$
PageRank in PowerGraph

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

**PowerGraph/PageRank(i)**

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

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

- **Scatter**( i → j ) :
  
  if \( R[i] \) changed then trigger \( j \) to be **recomputed**
Distributed Execution of a PowerGraph Vertex-Program

**Gather**

**Apply**

**Scatter**
Minimizing Communication in PowerGraph

Communication is linear in the number of machines each vertex spans.

A vertex-cut minimizes machines each vertex spans.

Percolation theory suggests that power law graphs have good vertex cuts. [Albert et al. 2000]
New Approach to Partitioning

- Rather than cut edges:

  **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

Machine 1
Machine 2
Machine 3

Balanced Vertex-Cut

Y Spans 3 Machines
Z Spans 2 Machines
Not cut!
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
Greedy Vertex-Cuts

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

Machine 1

Machine 2

B

E
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

Oblivious balances cost and partitioning time.
Greedy Vertex-Cuts Improve Performance

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 Evaluation
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:

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

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

PowerGraph is robust to high-degree vertices.
PageRank on the Twitter Follower Graph

Natural Graph with 40M Users, 1.4 Billion Links

Communication

Total Network (GB)

GraphLab  Pregel (Piccolo)  PowerGraph

Reduces Communication

Runtime

Seconds

GraphLab  Pregel (Piccolo)  PowerGraph

Runs Faster

32 Nodes x 8 Cores (EC2 HPC cc1.4x)
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.