I/O-Efficient Big Graph Computation

Ge Yu
Northeastern University, China
yuge@cse.neu.edu.cn
Outline

1. Background & Challenges
2. Push & Pull Style Message Optimization
3. Open source Hybrid-Graph System
4. Work in Progress
1. Background

• Big Graph
  – Social network, scientific computations, ...
  – Billion vertices & trillion edges, still growing
  – Iterative algorithms: PageRank, Shortest Path
Background

• How to handle “Big Graph”?
  – Graph data
    “There are 10.4 billion daily active users on Facebook” (Dec, 2015)
  – Message data

<table>
<thead>
<tr>
<th>Solution</th>
<th>Benefits</th>
<th>Problems</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-machine+Disk</td>
<td>Ease of Management</td>
<td>Poor scalability</td>
<td>GraphChi, TurboGraph</td>
</tr>
<tr>
<td>Cluster + memory</td>
<td>High efficiency</td>
<td>Scalability vs. Expense</td>
<td>Pregel, GPS, GraphLab</td>
</tr>
<tr>
<td>Cluster + disk</td>
<td>Scalability</td>
<td>I/O-inefficiency</td>
<td>Gbase, Giraph, MOCgraph, Pregelix</td>
</tr>
</tbody>
</table>
Background

- **BSP based Distributed Graph Processing Systems**
  - Pregel[1], Hama[2], Giraph[3], GPS[4], ...
  - Spark[5], Bagel[5], GraphX[6]
  - GraphLab[7], PowerGraph[8]
**Background**

- **Example of Computation Expense**
  - Twitter-2010 ($|V| = 41.7$ million, $|E| = 1.4$ billion)
    - Graph data: 13GB
    - Message data: $\geq 13$GB
  - Amazon EC2

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>Unit Price (per month)</th>
<th>Twitter-2010 (13+13GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>$9.36/GB</td>
<td>$244</td>
</tr>
<tr>
<td>SSD</td>
<td>$0.10/GB</td>
<td>$2.6</td>
</tr>
<tr>
<td>HDD</td>
<td>$0.05/GB</td>
<td>$1.3</td>
</tr>
</tbody>
</table>
Background

• Optimizations for distributed memory systems
  – **Partition:**
    METIS[9], LDG[10], vertex-cut[8], load balance[11,12]
  – **Message Management:**
    combiner[1], LALP[4]
  – **Convergence:**
    asynchronous[7,8]/block-centric update[13,14]
  – **Fault-tolerance:**
    checkpoint[1], vertex replication[15]

All these optimizations are NOT designed for I/O efficient computation in disk-resident environments!
Challenges

• **I/O-inefficiency for Disk-based systems**
  – Graph data IO Cost
    read & write vertices
    read edges
  – Message data IO Cost
    read & write

• **Other problems**
  – Graph partitioning
  – Convergence
  – Fault-tolerance
  – Data update and incremental maintenance
2. Push vs. Pull

- Two Existing data operations: Push and Pull
- **PUSH Approach** *(e.g. PageRank on Giraph)*
  Messages are generated when updating vertices at the i-th iteration, but used in the (i+1)-th iteration
  - I/O of graph data: seq. reads & writes
  - I/O of message data: random writes & seq. reads

Favorite scenario:
More Messages in memory
Push vs. Pull

- **PULL Approach** *(e.g. PageRank on modified PowerGraph)*
  - Messages are generated on demand of vertex updates and then consumed immediately at the \((i+1)\)-th iteration
  - I/O of graph data: random reads & seq. writes
  - I/O of message data: none

Favorite scenario:
More Graph data in memory

<table>
<thead>
<tr>
<th>scenarios</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>livej</td>
</tr>
<tr>
<td>original</td>
<td>3.0</td>
</tr>
<tr>
<td>ext-mem</td>
<td>3.1</td>
</tr>
<tr>
<td>ext-edge</td>
<td>3.9</td>
</tr>
<tr>
<td>ext-edge-v3</td>
<td>4.5</td>
</tr>
<tr>
<td>ext-edge-v2.5</td>
<td>654.7</td>
</tr>
</tbody>
</table>
Existing systems

- **Distributed systems**
  - **Memory-based**
    PUSH/PULL
  - **Disk-based**
    PUSH

<table>
<thead>
<tr>
<th>Name</th>
<th>PUSH</th>
<th>PULL</th>
<th>DISK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Giraph++</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blogel</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GiraphX</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPS</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GRE</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mizan</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naiad</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pregel</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trinity</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faunus</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PEGASUS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Gbase</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Giraph</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GraphX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MOCgraph</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hama</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Pregelix</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Surfer</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Chronos</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Kineograph</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Pregel+</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Kylin</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Seraph</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>GraphLab</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PowerGraph</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>LFGraph</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Our Hybrid Solution

• Switching between push and pull adaptively.

• Challenges
  – Existing pull-based approaches are I/O-inefficient.
  – How to efficiently combine push and pull?

• Contributions
  – **b-pull**: pull messages in block-centric, not vertex-centric
  – **VE-BLOCK**: data structure for disk-based graph data
  – **Hybrid engine**: a seamless switching mechanism an effective performance prediction model

Block-centric Pull (b-pull)

- Strength
  - avoid message data read & write
  - reduce source vertex access costs
  - reduce pull request communication costs
Graph Storage: VE-BLOCK

- **VE-BLOCK**
  - VBlocks
  - Ebblocks
  - Metadata
Block-centric Pull (b-pull)

- **Pulling messages in VBlocks**
  - 1) Receiver side:
    - sending pull requests for one local Vblock $b_i$
  - 2) Sender side:
    - producing and sending messages for $b_i$ on demand by reading Vblocks $b_j$ and Eblocks $g_{ji}$
  - 3) Receiver side:
    - consuming messages immediately and updating values of vertices in $b_i$

One iteration = Repeat 1)-3) for every Vblock $b_i$
Hybrid-Decomposition

• Decomposing push and b-pull
  – push: load() ⇆ update() ⇆ pushRes()
  – b-pull: pullRes() ⇆ update()
    shared update()

• Graph storage
  – push: Vblocks + edges (in adjacency list)
  – b-pull: Vblocks + edges (in Eblocks)
    shared Vblocks and two replicas of edges
Hybrid-Switching Operations

- Switching between b-pull and push
  - b-pull $\Rightarrow$ push: pulling & pushing messages
  - push $\Rightarrow$ b-pull: no messages are generated
Hybrid-Switching Timing

- Performance metric: $Q = Q_{push} - Q_{b-pull}$

$$Q^t = \frac{M_{coByte_m}}{s_{net}} + \frac{IO(M_{disk})}{s_{rw}} - \frac{IO(V_{rr}^t)}{s_{rr}} + \frac{IO(E^t) + IO(M_{disk}) - IO(\mathcal{E}^t) - IO(F^t)}{s_{sr}}$$

$Q^t > 0$, b-pull, otherwise, push

- How to predict $Q$?

  Metrics collected at the $t$-th superstep are regarded as the predicted values on superstep $(t+x)$. (based on the methods in Ref [12] Z. Shang et al., ICDE2013)

- Default Prediction interval $x=2$
  - Prediction accuracy $\propto 1/x$, $x \in N^+$
  - $x \geq 2$, to balance the cost and gain
4. HybridGraph System

- Architecture

https://github.com/HybridGraph/HybridGraph
HybridGraph System

- Console of Running one job
Evaluation

• Compared solutions
  – push: Giraph
  – pushM: MOCgraph (message online computing)
  – pull: GraphLab PowerGraph (vertex-cut optimizations)
  – b-pull: our block-centric pull
  – hybrid: our hybrid solution on top of push and b-pull

• Cluster (31 nodes)
  – Local cluster: HDDs
  – Amazon cluster: SSDs

<table>
<thead>
<tr>
<th>Cluster</th>
<th>RAM</th>
<th>Disk</th>
<th>$s_{rr}$/$s_{rw}$/$s_{sr}$</th>
<th>$s_{net}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>6.0GB</td>
<td>500GB</td>
<td>1.177/1.182/2.358MB/s</td>
<td>112MB/s</td>
</tr>
<tr>
<td>amazon</td>
<td>7.5GB</td>
<td>30GB</td>
<td>18.177/18.194/18.270MB/s</td>
<td>116MB/s</td>
</tr>
</tbody>
</table>
## Evaluation

### Dataset

Table 4: Real Graph Datasets (M: million)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>Degree</th>
<th>Type</th>
<th>Disk size</th>
</tr>
</thead>
<tbody>
<tr>
<td>livej</td>
<td>4.8M</td>
<td>68M</td>
<td>14.2</td>
<td>Social networks</td>
<td>0.50GB</td>
</tr>
<tr>
<td>wiki</td>
<td>5.7M</td>
<td>130M</td>
<td>22.8</td>
<td>Web graphs</td>
<td>0.98GB</td>
</tr>
<tr>
<td>orkut</td>
<td>3.1M</td>
<td>234M</td>
<td>75.5</td>
<td>Social networks</td>
<td>1.59GB</td>
</tr>
<tr>
<td>twi</td>
<td>41.7M</td>
<td>1,470M</td>
<td>35.3</td>
<td>Social networks</td>
<td>12.90GB</td>
</tr>
<tr>
<td>fri</td>
<td>65.6M</td>
<td>1,810M</td>
<td>27.5</td>
<td>Social networks</td>
<td>17.00GB</td>
</tr>
<tr>
<td>uk</td>
<td>105.9M</td>
<td>3,740M</td>
<td>35.6</td>
<td>Web graphs</td>
<td>33.02GB</td>
</tr>
</tbody>
</table>

![Disk Size (GB)](chart.png)
Evaluation

- **Limited memory (local cluster with HDDs)**
  - b-pull vs. push: up to 35x
  - b-pull vs. pushM: up to 16x
  - b-pull vs. pull: up to 239x
  - hybrid vs. b-pull: up to 1.6x

(a) Runtime of PageRank

(b) Runtime of SSSP
Evaluation

- Limited memory (Amazon cluster with SSDs)
  - b-pull vs. push: up to 56x
  - b-pull vs. pushM: up to 15x
  - b-pull vs. pull: up to 343x
  - hybrid vs. b-pull: up to 1.5x

(a) Runtime of PageRank

(b) Runtime of SSSP
5. Work in progress (1)

- Partitioning graph for I/O-efficient computation
  - Besides reducing # of cut-edges and balancing load, also considering reducing I/O costs of graph data index
  - Building a reverse graph $G$ and Clustering vertices on $G$ based on the similarities of outgoing edges (for I/O costs)
  - Assigning clustered vertex blocks among machines using METIS (for cut-edges and load balance)
Work in progress (2)

- Prioritized block scheduling for b-PULL
  - Asynchronous vertex update to speed up message propagation
  - Prioritized block scheduling instead of vertex scheduling to avoid random access to vertices
  - Estimating block priority based on dynamical block-dependency graph
Work in progress (3)

- Lightweight Fault-tolerance using b-pull
  - Archiving local historical data (vertices, not messages)
  - Writing distributed checkpoint data in parallel with updating vertices (not a blocking way)
  - For failure recovery, re-pulling messages for lost vertices and updating lost vertex values based on b-pull
Work in progress (4)

• More functions for various applications
  – Offering a library implementing the typical iterative algorithms (graph computation, matrix analysis and data mining)
  – Friendly interface and visualization
  – Incremental computation and index maintenance for graph data updates
  – Support for temporal graphs and hypergraphs
References


Thanks!