GraphChi: Large-Scale Graph Computation on Just a PC

Authors: Aapo Kyrola, Guy Blelloch, Carlos Guestrin

Presenter: Terryn Brunelle
Motivation
Motivation

- Large graphs require distributed computing
  - Ex: Social networks, web graphs, protein interaction graphs
- Distributed graph algorithm development challenging to non-experts
Existing Work: Vertex-Centric

Algorithm 1: Typical vertex update-function

1. Update(vertex) begin
2. \( x[] \leftarrow \text{read values of in- and out-edges of vertex} \);
3. \( \text{vertex.value} \leftarrow f(x[]) \);
4. \( \text{foreach edge of vertex do} \)
5. \( \quad \text{edge.value} \leftarrow g(\text{vertex.value, edge.value}); \)
6. \( \text{end} \)
7. \( \text{end} \)
Problems

- Can scale to billions of edges by distributing computation...
- But to do so need to partition graph across cluster nodes
- Finding efficient graph cuts is difficult
Goal

Find graph cuts that

- Minimize communication between nodes
- Are balanced
GraphChi
Parallel Sliding Windows (PSW)

- Process very large graphs on disk
- Asynchronous model of computation
PSW Approach

1. Load subgraph from disk
2. Update vertices and edges
3. Write updated values to disk
1. Load Subgraph from Disk
2. Update Vertices and Edges

- Within each interval
  - Execute update-function for each vertex in parallel
- Enforce external determinism
  - Critical vertices updated in sequential order
  - Non-critical vertices updated in parallel
3. Write Updated Values to Disk

- Load edges from disk in large blocks cached in memory
- Write to blocks and load them back to disk to replace old data
  - Completely rewrite memory shard
  - Only rewrite active sliding windows of other shards
- P non-sequential disk writes per interval
PSW Example

(a) Execution interval (vertices 1-2)  (b) Execution interval (vertices 1-2)  (c) Execution interval (vertices 3-4)  (d) Execution interval (vertices 3-4)
I/O Complexity

\[
\frac{2|E|}{B} \leq Q_B(E) \leq \frac{4|E|}{B} + \Theta(P^2)
\]
GraphChi Data Pre-Processing

Compact Shard Format
- Adjacency shard -- edge array for each vertex
- Edge data shard -- flat array of edge values

Sharder
- Count vertex in-degrees
- Compute prefix sum over degree array
- Divide vertices into P intervals
- Write each edge to temporary scratch file of owning shard
- For each file, sort edges and write them in compact format

I/O Cost: 5|E|/B + |V|/B
GraphChi Implementation

- Calculate exact memory needed for execution interval
  - Use multithreading to access needed vertices
  - Degreefile stores in/out degrees for each vertex
- Divide execution into sub-intervals
- Evolving graphs
- Selective Scheduling
Main Execution

Figure 5: Main execution flow. Sequence of operations for processing one execution interval with GraphChi.

1 typedef: VertexType float
2 Update(vertex) begin
3 var sum ← 0
4 for e in vertex.inEdges() do
5     sum += e.weight * neighborRank(e)
6 end
7 vertex.setValue(0.15 + 0.85 * sum)
8 broadcast(vertex)
9 end
Algorithm 5: Type definitions, and implementations of neighborRank() and broadcast() in the standard model.

1  typedef: EdgeType { float weight, neighbor_rank; }
2  neighborRank(edge) begin
3    return edge.weight * edge.neighbor_rank
4  end
5  broadcast(vertex) begin
6    for e in vertex.outEdges() do
7      e.neighbor_rank = vertex.getValue()
8    end
9  end
Algorithm 6: Datatypes and implementations of neighborRank() and broadcast() in the alternative model.

1. `typedef` EdgeType { float weight; }
2. float[] in_mem_vert
3. neighborRank(edge) **begin**
   4. | return edge.weight * in_mem_vert[edge.vertex_id]
5. **end**
6. broadcast(vertex) /* No-op */
Applications

- SpMV Kernels
- Graph Mining
- Collaborative Filtering
- Probabilistic Graphical Models
Experimental Results
# Experimental Results

<table>
<thead>
<tr>
<th>Application &amp; Graph</th>
<th>Iter.</th>
<th>Comparative result</th>
<th>GraphChi (Mac Mini)</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pagerank &amp; domain</td>
<td>3</td>
<td>GraphLab[30] on AMD server (8 CPUs) 87 s</td>
<td>132 s</td>
<td>-</td>
</tr>
<tr>
<td>Pagerank &amp; twitter-2010</td>
<td>5</td>
<td>Spark [45] with 50 nodes (100 CPUs): 486.6 s</td>
<td>790 s</td>
<td>[38]</td>
</tr>
<tr>
<td>Pagerank &amp; V=105M, E=3.7B</td>
<td>100</td>
<td>Stanford GPS, 30 EC2 nodes (60 virt. cores), 144 min</td>
<td>approx. 581 min</td>
<td>[37]</td>
</tr>
<tr>
<td>Pagerank &amp; V=1.0B, E=18.5B</td>
<td>1</td>
<td>Piccolo, 100 EC2 instances (200 cores) 70 s</td>
<td>approx. 26 min</td>
<td>[36]</td>
</tr>
<tr>
<td>Webgraph-BP &amp; yahoo-web</td>
<td>1</td>
<td>Pegasus (Hadoop) on 100 machines: 22 min</td>
<td>27 min</td>
<td>[22]</td>
</tr>
<tr>
<td>ALS &amp; netflix-mm, D=20</td>
<td>10</td>
<td>GraphLab on AMD server: 4.7 min</td>
<td>9.8 min (in-mem)</td>
<td>[30]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40 min (edge-repl.)</td>
<td></td>
</tr>
<tr>
<td>Triangle-count &amp; twitter-2010</td>
<td>-</td>
<td>Hadoop, 1636 nodes: 423 min</td>
<td>60 min</td>
<td>[39]</td>
</tr>
<tr>
<td>Pagerank &amp; twitter-2010</td>
<td>1</td>
<td>PowerGraph, 64 x 8 cores: 3.6 s</td>
<td>158 s</td>
<td>[20]</td>
</tr>
<tr>
<td>Triangle-count &amp; twitter-2010</td>
<td>-</td>
<td>PowerGraph, 64 x 8 cores: 1.5 min</td>
<td>60 min</td>
<td>[20]</td>
</tr>
</tbody>
</table>


Experimental Results

- Within constant factor of other systems
- Uses a fraction of the resources
- Can process 5-20 million edges/second on Mac Mini
Conclusion
Strengths/Weaknesses

Strengths

- Sparse graphs
- Efficient on consumer PC
- Makes large-scale graph computation widely accessible

Weaknesses

- Difficult to benchmark results
- Dynamic ordering and graph traversals
Directions for Future Work

- Evaluating more efficient shard formats
- Testing on additional infrastructures
Discussion Questions

- Even though there were no comparable models to benchmark GraphChi against, do you find the experimental results compelling?
- How would GraphChi perform on dense graphs?
- Could GraphChi be adapted to support graph traversal problems?