Graph-Based Parallel Computing

William Cohen
Announcements

• Next Tuesday 12/8:
  – Presentations for 10-805 projects.
  – 15 minutes per project.
  – Final written reports due Tues 12/15

• For exam:
  – Spectral clustering will not be covered
  – It’s ok to bring in two pages of notes
  – We’ll give a solution sheet for HW7 out on Wednesday noon

  • but you get no credit on questions HW7 5-7 if you turn in answers after that point
Outline

• Motivation/where it fits
• Sample systems (c. 2010)
  – Pregel: and some sample programs
    • Bulk synchronous processing
  – Signal/Collect and GraphLab
    • Asynchronous processing
• GraphLab descendants
  – PowerGraph: partitioning
  – GraphChi: graphs w/o parallelism
  – GraphX: graphs over Spark
Many Graph-Parallel Algorithms

• Collaborative Filtering
  – Alternating Least Squares
  – Stochastic Gradient Descent
  – Tensor Factorization
• Structured Prediction
  – Loopy Belief Propagation
  – Max-Product Linear Programs
  – Gibbs Sampling
• Semi-supervised ML
  – Graph SSL
  – CoEM
• Community Detection
  – Triangle-Counting
  – K-core Decomposition
  – K-Truss
• Graph Analytics
  – PageRank
  – Personalized PageRank
  – Shortest Path
  – Graph Coloring
• Classification
  – Neural Networks
Signal/collect model

```java
v.doSignal()
    lastSignalState := state
for all (e ∈ outgoingEdges) do
    e.target.uncollectedSignals.append(e.signal())
    e.target.signalMap.put(e.sourceId, e.signal())
end for

v.doCollect()
    state := collect()
    uncollectedSignals := Nil
```

- Signals are made available in a list and a map
- Next state for a vertex is output of the collect() operation

**Algorithm 1 Synchronous execution**

```java
for i ← 1..num_iterations do
    for all v ∈ V parallel do
        v.doSignal()
    end for
    for all v ∈ V parallel do
        v.doCollect()
    end for
end for
```
<table>
<thead>
<tr>
<th>initialState</th>
<th>if (isTrainingData) trainingData else avgProbDist</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>if (isTrainingData)</td>
</tr>
<tr>
<td></td>
<td>return oldState</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>return signals.sum.normalise</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>initialState</th>
<th>Set(id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>return union(oldState, union(signals))</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state</td>
</tr>
</tbody>
</table>

Fig. 8. Transitive closure (data-graph/data-flow).

<table>
<thead>
<tr>
<th>initialState</th>
<th>randomColour</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>if (contains(signals, oldState))</td>
</tr>
<tr>
<td></td>
<td>return randomColorExcept(oldState)</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>return oldState</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state</td>
</tr>
</tbody>
</table>

Fig. 9. Vertex colouring (data-graph).
CoEM (Rosie Jones, 2005)

<table>
<thead>
<tr>
<th>System</th>
<th>Cores</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>95</td>
<td>7.5 hrs</td>
</tr>
<tr>
<td>GraphLab</td>
<td>16</td>
<td>30 min</td>
</tr>
</tbody>
</table>

6x fewer CPUs! 15x Faster!
GRAPH ABSTRACTIONS: GRAPHLAB CONTINUED....
Outline

• Motivation/where it fits
• Sample systems (c. 2010)
  – Pregel: and some sample programs
    • Bulk synchronous processing
  – Signal/Collect and GraphLab
    • Asynchronous processing
• GraphLab descendants
  – PowerGraph: partitioning
  – GraphChi: graphs w/o parallelism
  – GraphX: graphs over Spark
GraphLab’s descendents

• PowerGraph
• GraphChi
• GraphX

On multicore architecture: shared memory for workers

On cluster architecture (like Pregel): different memory spaces

What are the challenges moving away from shared-memory?
Top 1% of vertices is adjacent to 53% of the edges!
Problem:
High Degree Vertices Limit Parallelism

- Edge information too large for single machine
- Touches a large fraction of graph (GraphLab 1)
- Produces many messages (Pregel, Signal/Collect)

Asynchronous consistency requires heavy locking (GraphLab 1)

Synchronous consistency is prone to stragglers (Pregel)
PowerGraph

- Problem: GraphLab’s localities can be large
  - “all neighbors of a node” can be large for hubs, high indegree nodes

- Approach:
  - new graph partitioning algorithm
    - can replicate data
  - gather-apply-scatter API: finer-grained parallelism
    - gather ~ combiner
    - apply ~ vertex UDF (for all replicates)
    - scatter ~ messages from vertex to edges
**Signal/collect examples**

**Single-source shortest path**

<table>
<thead>
<tr>
<th>initialState</th>
<th>if (isSource) 0 else infinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>return min(oldState, min(signals))</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state + edge.weight</td>
</tr>
</tbody>
</table>

**Initial State:**
- 0 → ∞
- ∞ ← ∞

**Step 1:**
- 0 → 1
- 1 ← ∞

**Step 2:**
- 0 → 1
- 1 ← 2
## Signal/collect examples

### Life

<table>
<thead>
<tr>
<th>initState</th>
<th>if (isInitiallyAlive) 1 else 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>switch (sum(signals))</td>
</tr>
<tr>
<td></td>
<td>case 0: return 0 // dies of loneliness</td>
</tr>
<tr>
<td></td>
<td>case 1: return 0 // dies of loneliness</td>
</tr>
<tr>
<td></td>
<td>case 2: return oldState // same as before</td>
</tr>
<tr>
<td></td>
<td>case 3: return 1 // becomes alive if dead</td>
</tr>
<tr>
<td></td>
<td>other: return 0 // dies of overcrowding</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state</td>
</tr>
</tbody>
</table>

## PageRank

<table>
<thead>
<tr>
<th>initState</th>
<th>baseRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>return baseRank + dampingFactor * sum(signals)</td>
</tr>
<tr>
<td>signal()</td>
<td>return source.state * edge.weight / sum(edgeWeights(source))</td>
</tr>
</tbody>
</table>
## Signal/collect examples

**Co-EM/wvRN/Harmonic fields**

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialState</td>
<td><code>if (isTrainingData) trainingData else avgProbDist</code></td>
</tr>
<tr>
<td>collect()</td>
<td><code>if (isTrainingData)</code></td>
</tr>
<tr>
<td></td>
<td><code>return oldState</code></td>
</tr>
<tr>
<td></td>
<td><code>else</code></td>
</tr>
<tr>
<td></td>
<td><code>return signals.sum.normalise</code></td>
</tr>
<tr>
<td>signal()</td>
<td><code>return source.state</code></td>
</tr>
</tbody>
</table>
PageRank in PowerGraph

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

- **Gather** \(( j \rightarrow i )\): return \( w_{ji} \times R[j] \)
- **sum** \((a, b)\): return \(a + b\)
- **Apply** \((i, \Sigma)\): \( R[i] = \beta + (1 - \beta) \times \Sigma \)
- **Scatter** \((i \rightarrow j)\):
  - if \((R[i] \text{ changes})\) then **activate**\((j)\)

Gather/sum like a *collect* or a *group by ... reduce* (with combiner)

**PageRankProgram**(\(i\))

scatter is like a *signal*
Distributed Execution of a PowerGraph Vertex-Program

Gather

Apply

Scatter

Machine 1

Machine 2

Machine 3

Machine 4
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]
Partitioning Performance

Twitter Graph: 41M vertices, 1.4B edges

Oblivious balances partition quality and partitioning time.
Partitioning matters...

![Bar chart showing reduction in runtime for different algorithms](image)

- **PageRank**
  - Random: 1.0
  - Oblivious: 0.8
  - Greedy: 0.5

- **Collaborative Filtering**
  - Random: 0.9
  - Oblivious: 0.7
  - Greedy: 0.5

- **Shortest Path**
  - Random: 1.0
  - Oblivious: 0.8
  - Greedy: 0.5
Outline

• Motivation/where it fits
• Sample systems (c. 2010)
  – Pregel: and some sample programs
    • Bulk synchronous processing
  – Signal/Collect and GraphLab
    • Asynchronous processing
• GraphLab descendants
  – PowerGraph: partitioning
  – **GraphChi**: graphs w/o parallelism
  – GraphX: graphs over Spark
GraphLab’s descendents

• PowerGraph
• GraphChi
• GraphX
GraphLab con’t

• PowerGraph

• GraphChi
  – Goal: use graph abstraction on-disk, not in-memory, on a conventional workstation
GraphLab con’t

• GraphChi
  – Key insight:
    • some algorithms on graph are streamable (i.e., PageRank-Nibble)
    • in general we can’t easily stream the graph because neighbors will be scattered
    • but maybe we can limit the degree to which they’re scattered … enough to make streaming possible?
      – “almost-streaming”: keep P cursors in a file instead of one
• Vertices are numbered from 1 to n
  – P intervals, each associated with a **shard** on disk.
  – **sub-graph** = interval of vertices
PSW: Layout

Shard: in-edges for interval of vertices; sorted by source-id

Vertices 1..100
Shard 1

Vertices 101..700
Shard 2

Vertices 701..1000
Shard 3

Vertices 1001..10000
Shard 4

Shards small enough to fit in memory; balance size of shards

1. Load
2. Compute
3. Write
PSW: Loading Sub-graph

Load subgraph for vertices 1..100

1. Load
2. Compute
3. Write

What about out-edges?
Arranged in sequence in other shards
PSW: Loading Sub-graph

Load subgraph for vertices 101..700

1. Load
2. Compute
3. Write
PSW Load-Phase

Only $P$ large reads for each interval.

$P^2$ reads on one full pass.
PSW: Execute updates

• Update-function is executed on interval’s vertices
• Edges have **pointers** to the loaded data blocks
  – Changes take effect immediately → **asynchronous.**
PSW: Commit to Disk

- In write phase, the blocks are written back to disk
  - Next load-phase sees the preceding writes asynchronous.

In total:
- \( P^2 \) reads and writes / full pass on the graph.
- Performs well on both SSD and hard drive.

To make this work: the size of a vertex state can’t change when it’s updated (at last, as stored on disk).
Experiment Setting

- Mac Mini (Apple Inc.)
  - 8 GB RAM
  - 256 GB SSD, 1TB hard drive
  - Intel Core i5, 2.5 GHz
- Experiment graphs:

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>P (shards)</th>
<th>Preprocessing</th>
</tr>
</thead>
<tbody>
<tr>
<td>live-journal</td>
<td>4.8M</td>
<td>69M</td>
<td>3</td>
<td>0.5 min</td>
</tr>
<tr>
<td>netflix</td>
<td>0.5M</td>
<td>99M</td>
<td>20</td>
<td>1 min</td>
</tr>
<tr>
<td>twitter-2010</td>
<td>42M</td>
<td>1.5B</td>
<td>20</td>
<td>2 min</td>
</tr>
<tr>
<td>uk-2007-05</td>
<td>106M</td>
<td>3.7B</td>
<td>40</td>
<td>31 min</td>
</tr>
<tr>
<td>uk-union</td>
<td>133M</td>
<td>5.4B</td>
<td>50</td>
<td>33 min</td>
</tr>
<tr>
<td>yahoo-web</td>
<td>1.4B</td>
<td>6.6B</td>
<td>50</td>
<td>37 min</td>
</tr>
</tbody>
</table>
Comparison to Existing Systems

On a Mac Mini:

- ✓ GraphChi can solve as big problems as existing large-scale systems.
- ✓ Comparable performance.

Notes: comparison results do not include time to transfer the data to cluster, preprocessing, or the time to load the graph from disk. GraphChi computes asynchronously, while all but GraphLab synchronously.
Outline

- Motivation/where it fits
- Sample systems (c. 2010)
  - Pregel: and some sample programs
    - Bulk synchronous processing
  - Signal/Collect and GraphLab
    - Asynchronous processing
- GraphLab “descendants”
  - PowerGraph: partitioning
  - GraphChi: graphs w/o parallelism
  - GraphX: graphs over Spark (Gonzalez)
GraphLab’s descendents

• PowerGraph
• GraphChi
• **GraphX**
  – implementation of GraphLabs API on top of Spark
  – Motivations:
    • avoid transfers between subsystems
    • leverage larger community for common infrastructure
  – What’s different:
    • Graphs are now *immutable* and operations transform one graph into another (RDD ➞ RDG, resilient distributed graph)
The GraphX Stack
(Lines of Code)

- PageRank (5)
- Connected Comp. (10)
- Shortest Path (10)
- SVD (40)
- ALS (40)
- K-core (51)
- Triangle Count (45)
- LDA (120)

Pregel (28) + GraphLab (50)

GraphX (3575)

Spark
Idea: Graph as Tables

Property Graph

Under the hood things can be split even more finely: eg a vertex map table + vertex data table. Operators maximize structure sharing and minimize communication.

(Not shown: partition id’s, carefully assigned....)
Like signal/collect:

- Join vertex and edge tables
- Does map with mapFunc on the edges
- Reduces by destination vertex using reduceFunc

```python
class Graph[V, E]:

def vertex(self):
    VertexDataTable v
    JOIN
    VertexMap vm
    ON (v.id=vm.id)
    RIGHT OUTER JOIN
    EdgeTable e
    ON (e.pid=vm.pid && (e.src=v.id OR e.dst=v.id))
    WITH PARTITIONER edgeTable.partitioner ON pid

def edges(self) -> Graph[V, E2]:
    Id, Id, E => (Id, Id, E2): Graph[V, E2]

def candidateVertices(self, tbl: RDD[(Id, A)],
                      src: (Id, V, A) => (Id, V2)): Graph[V2, E]

def aggregateNeighbors(
    mapFunc: (Id, Edge[V, E]) => A,
    reduceFunc: (A, A) => A): RDD[(Id, A)]

def reverseEdgeDirection(): Graph[V, E] =
    mapEdges(e => (e.dst, e.src, e.data))

def degree(): RDD[(Id, Int)] =
    aggregateNeighbors((id, e) => 1, (a, b) => a + b)
}
Distributed Execution of a PowerGraph Vertex-Program

Gather

Apply

Scatter
def Pregel(graph: Graph[V,E],
    initialMsg: M
    vprogf: ((Id,V), M) => V,
    sendMsggf: Edge[V,E] => Option[M],
    combinef: (M,M) => M,
    numIter: Long): Graph[V,E] = {

    // Initialize the messages to all vertices
    var msgs: RDD[(Vid, A)] =
        graph.vertices.map(v => (v.id, initialMsg))

    // Loop while there are messages
    var i = 0
    while (msgs.count > 0 && i < maxIter) {
        // Receive the message sums on each vertex
        graph = graph.updateVertices(msgs, vprogf)

        // Compute and combine new messages
        msgs = graph.aggregateNeighbors(sendMsggf, combinef)
        i = i + 1
    }
}
// Load and initialize the graph
val graph = Graph.load('hdfs://webgraph.tsv')
var prGraph = graph.updateV(graph.degrees(OutEdges),
  (v, deg) => (v.id, (deg, 1.0)) // Initial rank=1

// Execute PageRank
prGraph = Pregel(prGraph,
  1.0, // Initial message is 1.0
  vprogf = // Update Rank
    (v, msg) => (v.deg, 0.15 + 0.85 * msg),
  sendMsgf = // Compute Msg
    e => e.src.rank/e.src.deg,
  combinef = // Combine msg
    (m1, m2) => m1 + m2,
  10) // Run 10 iterations

// Display the maximum PageRank
print(prGraph.vertices.map(v=>v.rank).max)
Performance Comparisons

Live-Journal: 69 Million Edges

<table>
<thead>
<tr>
<th>Framework</th>
<th>Runtime (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mahout/Hadoop</td>
<td>1340</td>
</tr>
<tr>
<td>Naïve Spark</td>
<td>354</td>
</tr>
<tr>
<td>Giraph</td>
<td>207</td>
</tr>
<tr>
<td>GraphX</td>
<td>68</td>
</tr>
<tr>
<td>GraphLab</td>
<td>22</td>
</tr>
</tbody>
</table>

GraphX is roughly **3x slower** than GraphLab but: integrated with Spark, open-source, resilient.
Summary

• Large immutable data structures on (distributed) disk, processing by sweeping through then and creating new data structures:
  – stream-and-sort, Hadoop, PIG, Hive, ...

• Large immutable data structures in distributed memory:
  – Spark – distributed tables

• Large mutable data structures in distributed memory:
  – parameter server: structure is a hashtable
  – Pregel, GraphLab, GraphChi, GraphX: structure is a graph
Summary

• APIs for the various systems vary in detail but have a similar flavor
  – Typical algorithms iteratively update vertex state
  – Changes in state are communicated with messages which need to be aggregated from neighbors

• Biggest wins are
  – on problems where graph is fixed in each iteration, but vertex data changes
  – on graphs small enough to fit in (distributed) memory
Some things to take away

• Platforms for iterative operations on graphs
  – GraphX: if you want to integrate with Spark
  – GraphChi: if you don’t have a cluster
  – GraphLab/Dato: if you don’t need free software and performance is crucial
  – Pregel: if you work at Google
  – Giraph, Signal/collection, ... ??

• Important differences
  – Intended architecture: shared-memory and threads, distributed cluster memory, graph on disk
  – How graphs are partitioned for clusters
  – If processing is synchronous or asynchronous