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
Graph-Based Parallel Computing

William Cohen
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
Problems we’ve seen so far

• Operations on sets of sparse feature vectors:
  – Classification
  – Topic modeling
  – Similarity joins

• Graph operations:
  – PageRank, personalized PageRank
  – Semi-supervised learning on graphs
Architectures we’ve seen so far

• Stream-and-sort: limited-memory, serial, simple workflows
• + parallelism: Map-reduce (Hadoop)
• + abstract operators like join, group: PIG, Hive, GuineaPig, ...
• + caching in memory and efficient iteration: Spark, Flink, ...
• + parameter servers (Petuum, ...)

+ .....?

one candidate: architectures for graph processing
Architectures we’ve seen so far

- 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
  - today: large mutable graphs
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
    • which kinds of brings things back to Map-Reduce systems
GRAPH ABSTRACTIONS: PREGEL
(SIGMOD 2010*)

*Used internally at least 1-2 years before
Many ML algorithms tend to have

- Sparse data dependencies
- Local computations
- Iterative updates

- Typical example: Gibbs sampling
Example: Gibbs Sampling [Guestrin UAI 2010]

1) Sparse Data Dependencies

2) Local Computations

3) Iterative Updates

For LDA: $Z_{d,m}$ for $X_{d,m}$ depends on others $Z$'s in doc $d$, and topic assignments to copies of word $X_{d,m}$
Pregel (Google, Sigmod 2010)

- Primary data structure is a graph
- Computations are sequence of supersteps, in each of which
  - user-defined function is invoked (in parallel) at each vertex \( v \), can get/set value
  - UDF can also issue requests to get/set edges
  - UDF can read messages sent to \( v \) in the last superstep and schedule messages to send to in the next superstep
  - Halt when every vertex votes to halt
- Output is directed graph
- Also: aggregators (like ALLREDUCE)
- Bulk synchronous processing (BSP) model: all vertex operations happen simultaneously
Pregel (Google, Sigmod 2010)

• One master: partitions the graph among workers
• Workers keep graph “shard” in memory
• Messages to other partitions are buffered

• Communication across partitions is expensive, within partitions is cheap
  – quality of partition makes a difference!
template<typename VertexValue, 
    typename EdgeValue, 
    typename MessageValue>

class Vertex {
    public:
        virtual void Compute(MessageIterator* msgs) = 0;

        const string& vertex_id() const;
        int64 superstep() const;

        const VertexValue& GetValue();
        VertexValue* MutableValue();
        OutEdgeIterator GetOutEdgeIterator();

        void SendMessageTo(const string& dest_vertex, 
                           const MessageValue& message);
        void VoteToHalt();
};

Figure 3: The Vertex API foundations.
Streaming PageRank:
with some long rows

• Repeat until converged:
  
  \[ v^{t+1} = cu + (1-c)Wv^t \]

• Store \( A \) as a list of edges: each line is: “i d(i) j”
• Store \( v’ \) and \( v \) in memory: \( v’ \) starts out as \( cu \)
• For each line “i d j”
  
  \[ v'[j] += (1-c)v[i]/d \]

We need to get the degree of \( i \) and store it locally

note we need to scan through the graph each time
class PageRankVertex
    : public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *MutableValue() =
                0.15 / NumVertices() + 0.85 * sum;
        }

        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
    }
};
Another task: single source shortest path

class ShortestPathVertex
  : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                       mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
edge weight
Figure 7: SSSP—1 billion vertex binary tree: varying number of worker tasks scheduled on 300 multicore machines
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
Low-Rank Matrix Factorization:

\[ f[i] = \arg \min_{w \in \mathbb{R}^d} \sum_{j \in \text{Nbrs}(i)} (r_{ij} - w^T f[j])^2 + \lambda ||w||_2^2 \]
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
GRAPH ABSTRACTIONS: SIGNAL/COLLECT (SEMANTIC WEB CONFERENCE, 2010)

Stutz, Strebel, Bernstein, Univ Zurich
class PageRankVertex
 : public Vertex<double, void, double> {
 public:

  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() =
      0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
};
class ShortestPathVertex
    : public Vertex<int, int, int> {
    
    void Compute(MessageIterator* msgs) {
        int mindist = IsSource(vertex_id()) ? 0 : INF;
        for (; !msgs->Done(); msgs->Next())
            mindist = min(mindist, msgs->Value());
        if (mindist < GetValue()) {
            *MutableValue() = mindist;
            OutEdgeIterator iter = GetOutEdgeIterator();
            for (; !iter.Done(); iter.Next())
                SendMessageTo(iter.Target(),
                    mindist + iter.GetValue());
        }
    VoteToHalt();
    }
};

Another task: single source shortest path
Signal/collect model vs Pregel

• Integrated with RDF/SPARQL
• Vertices can be non-uniform types
• Vertex:
  – *id*, mutable *state*, outgoing *edges*, most recent *received signals* (map: neighbor id ➔ signal), *uncollected signals*
  – user-defined *collect* function
• Edge: *id*, *source*, *dest*
  – user-defined *signal* function
• Allows *asynchronous* computations....via v.scoreSignal, v.scoreCollect

For “data-flow” operations

On multicore architecture: shared memory for workers
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

Algorithm 1 Synchronous execution
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

Signals are made available in a list and a map.
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>initialState</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>initialState</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>
### PageRank + Preprocessing and Graph Building

#### Algorithm

```scala
class Document(id: Any) extends Vertex(id, 0.15) {
  def collect = 0.15 + 0.85 * signals[Double].foldLeft(0.0)(_ + _)
  override def processResult = if (state > 5) println(id + "::" + state)
  override def scoreSignal = (state - lastSignalState.getOrElse(0)).abs
}
```

```scala
class Citation(citer: Any, cited: Any) extends Edge(citer, cited) {
  override type SourceVertexType = Document
  def signal = source.state * weight / source.sumOfOutWeights
}
```

#### Initialization

```scala
object Algorithm {
  def executeCitationRank(db: SparqlAccessor) {
    val computeGraph = new ComputeGraph(ScoreGuidedSynchronous)
    val citations = new SparqlTuples(db, "select ?source ?target where {
    + "?source <http://lsdis.cs.uga.edu/projects/semdis/opus#cites> ?target}"
    citations foreach {
      case (citer, cited) =>
        computeGraph.addVertex[Document](citer)
        computeGraph.addVertex[Document](cited)
        computeGraph.addEdge[Citation](citer, cited)
    }
    computeGraph.execute(signalThreshold = 0)
  }
}
```
# Signal/collect examples

## Co-EM/wvRN/Harmonic fields

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>initialState</code></td>
<td><code>if (isTrainingData) trainingData else avgProbDist</code></td>
</tr>
<tr>
<td><code>collect()</code></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><code>signal()</code></td>
<td><code>return source.state</code></td>
</tr>
<tr>
<td>initialState</td>
<td>Set(id)</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td>collect()</td>
<td><code>return union(oldState, union(signals))</code></td>
</tr>
<tr>
<td>signal()</td>
<td><code>return source.state</code></td>
</tr>
</tbody>
</table>

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

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

Fig. 9. Vertex colouring (data-graph).
<table>
<thead>
<tr>
<th>initialState</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td><code>return 1 / (1 + e^{- signals.sum})</code></td>
</tr>
<tr>
<td>signal()</td>
<td><code>return source.state * edge.weight</code></td>
</tr>
</tbody>
</table>

Fig. 15. Artificial neural networks (data-graph).
Signal/collection examples

Matching path queries:
 dept(X) -[member]→ postdoc(Y) -[recieved]→ grant(Z)

<table>
<thead>
<tr>
<th>initialState</th>
<th>emptySet</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>matched = successfulMatchesWithVertex(signals)</td>
</tr>
<tr>
<td></td>
<td>(fullyMatched, partiallyMatched) = partition(matched)</td>
</tr>
<tr>
<td></td>
<td>reportResults(fullyMatched)</td>
</tr>
<tr>
<td></td>
<td>return union(oldState - lastSignalState, partiallyMatched)</td>
</tr>
<tr>
<td>signal()</td>
<td>return successfulMatchesWithEdge(source.state)</td>
</tr>
</tbody>
</table>

dep(X) -[member]→ postdoc(Y) -[recieved]→ grant(Z)
Signal/collect examples: data flow

Matching path queries:
depth(X) -[member]→ postdoc(Y) -[recieved]→ grant(Z)

<table>
<thead>
<tr>
<th>initialState</th>
<th>emptySet</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>matched = successfulMatchesWithVertex(signals)</td>
</tr>
<tr>
<td></td>
<td>(fullyMatched, partiallyMatched) = partition(matched)</td>
</tr>
<tr>
<td></td>
<td>reportResults(fullyMatched)</td>
</tr>
<tr>
<td></td>
<td>return union(oldState - lastSignalState, partiallyMatched)</td>
</tr>
<tr>
<td>signal()</td>
<td>return successfulMatchesWithEdge(source.state)</td>
</tr>
</tbody>
</table>

note: can be multiple input signals
## Signal/collect examples

### Matching path queries:

$$\text{dept}(X) -[\text{member}] \rightarrow \text{postdoc}(Y) -[\text{received}] \rightarrow \text{grant}(Z)$$

<table>
<thead>
<tr>
<th>initialState</th>
<th>emptySet</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect()</td>
<td>matched = successfulMatchesWithVertex(signals)</td>
</tr>
<tr>
<td></td>
<td>(fullyMatched, partiallyMatched) = partition(matched)</td>
</tr>
<tr>
<td></td>
<td>reportResults(fullyMatched)</td>
</tr>
<tr>
<td></td>
<td>return union(oldState - lastSignalState, partiallyMatched)</td>
</tr>
<tr>
<td>signal()</td>
<td>return successfulMatchesWithEdge(source.state)</td>
</tr>
</tbody>
</table>

![Diagram](image.png)

$$\text{dept}(X=\text{MLD}) -[\text{member}] \rightarrow \text{postdoc}(Y=\text{partha}) -[\text{received}] \rightarrow \text{grant}(Z)$$
Signal/collection model vs Pregel

• Integrated with RDF/SPARQL
• Vertices can be non-uniform types
• **Vertex:**
  – *id*, mutable *state*, outgoing *edges*, most recent *received signals* (map: neighbor id ➔ signal), uncollected signals
  – user-defined *collect* function
• **Edge:** *id*, *source*, *dest*
  – user-defined *signal* function
• Allows *asynchronous* computations....via v.scoreSignal, v.scoreCollect
Asynchronous Parallel Computation

- **Bulk-Synchronous**: All vertices update in parallel
  - need to keep copy of “old” and “new” vertex values
- **Asynchronous**:  
  - Reason 1: if two vertices are not connected, can update them in any order  
    - more flexibility, less storage  
  - Reason 2: not all updates are equally important  
    - parts of the graph converge quickly, parts slowly
Algorithm 2 Score-guided synchronous execution

done := false
iter := 0
while iter < max_iter and !done do
    done := true
    iter := iter + 1
    for all v ∈ V parallel do
        if (v.scoreSignal() > s_threshold) then
            done := false
            v.doSignal()
        end if
    end for
    for all v ∈ V parallel do
        if (v.scoreCollect() > c_threshold) then
            done := false
            v.doCollect()
        end if
    end for
end while

using:
- v.scoreSignal
- v.scoreCollect
Algorithm 3: Score-guided asynchronous execution

ops := 0
while
    ops < max_ops and ∃v ∈ V(
        v.scoreSignal() > s_threshold or
        v.scoreCollect() > c_threshold)
    do
        S := choose subset of V
        for all v ∈ S parallel do
            Randomly call either v.doSignal() or v.doCollect() iff respective threshold is reached; increment ops if an operation was executed.
        end for
    end while
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
GRAPH ABSTRACTIONS: GRAPHLAB (UAI, 2010)

Guestrin, Gonzalez, Bikel, etc.

Many slides below pilfered from Carlos or Joey....
GraphLab

• Data in graph, UDF vertex function
• Differences:
  – some control over scheduling
    • vertex function can insert new tasks in a queue
  – messages must follow graph edges: can access adjacent vertices only
  – “shared data table” for global data
  – library algorithms for matrix factorization, coEM, SVM, Gibbs, ...
  – GraphLab ➔ Now Dato
Graphical Model Learning

![Graphical Model Learning Diagram]

15.5x speedup on 16 cpus

On multicore architecture: shared memory for workers
Gibbs Sampling

- **Protein-protein interaction networks** [Elidan et al. 2006]
  - Pair-wise MRF
  - 14K Vertices
  - 100K Edges

- 10x Speedup
- Scheduling reduces locking overhead
CoEM (Rosie Jones, 2005)

Named Entity Recognition Task

Is “Dog” an animal?
Is “Catalina” a place?

<table>
<thead>
<tr>
<th></th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>0.2M</td>
<td>20M</td>
</tr>
<tr>
<td>Large</td>
<td>2M</td>
<td>200M</td>
</tr>
</tbody>
</table>

the dog → <X> ran quickly

Australia → <X> travelled to <X>

Catalina Island → <X> is pleasant

<table>
<thead>
<tr>
<th>Hadoop</th>
<th>95 Cores</th>
<th>7.5 hrs</th>
</tr>
</thead>
</table>
CoEM (Rosie Jones, 2005)

<table>
<thead>
<tr>
<th></th>
<th>Number of CPUs</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadoop</td>
<td>95 Cores</td>
<td>7.5 hrs</td>
</tr>
<tr>
<td>GraphLab</td>
<td>16 Cores</td>
<td>30 min</td>
</tr>
</tbody>
</table>

**6x fewer CPUs!  15x Faster!**
GRAPH ABSTRATIONS: 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?
Natural Graphs $\rightarrow$ Power Law

Top 1% of vertices is adjacent to 53% of the edges!

Altavista Web Graph: 1.4B Vertices, 6.7B 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
Factorized Vertex Updates

Split update into 3 phases
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>

initialState

<table>
<thead>
<tr>
<th>initial</th>
<th>step 1</th>
<th>step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 → ∞</td>
<td>0 → 1</td>
<td>0 → 1</td>
</tr>
<tr>
<td>∞ ← ∞</td>
<td>∞ ← ∞</td>
<td>1 ← 2</td>
</tr>
</tbody>
</table>

Initial state

Step 1

Step 2
Signal/collect examples

initialState | \textbf{if} (isInitiallyAlive) 1 \textbf{else} 0
\hline
\textbf{collect()} | \textbf{switch} (\text{sum(signals)})
\hspace{1em} \textbf{case} 0: \text{return} 0 \quad \text{// dies of loneliness}
\hspace{1em} \textbf{case} 1: \text{return} 0 \quad \text{// dies of loneliness}
\hspace{1em} \textbf{case} 2: \text{return} \text{oldState} \quad \text{// same as before}
\hspace{1em} \textbf{case} 3: \text{return} 1 \quad \text{// becomes alive if dead}
\hspace{1em} \textbf{other: return} 0 \quad \text{// dies of overcrowding}
\hline
\textbf{signal()} | \text{return} \text{source.state}

PageRank

initialState | baseRank
\hline
\textbf{collect()} | \text{return} \text{baseRank} + \text{dampingFactor} \times \text{sum(signals)}
\hline
\textbf{signal()} | \text{return} \text{source.state} \times \text{edge.weight} / \text{sum(edgeWeights(source))}
PageRank + Preprocessing and Graph Building

```scala
class Document(id: Any) extends Vertex(id, 0.15) {
  def collect = 0.15 + 0.85 * signals[Double].foldLeft(0.0)(_ + _)
  override def processResult = if (state > 5) println(id + ":" + state)
  override def scoreSignal = (state - lastSignalState.getOrElse(0)).abs
}

class Citation(citer: Any, cited: Any) extends Edge(citer, cited) {
  override type SourceVertexType = Document
  def signal = source.state * weight / source.sumOfOutWeights
}

object Algorithm {
  def executeCitationRank(db: SparqlAccessor) {
    val computeGraph = new ComputeGraph(ScoreGuidedSynchronous)
    val citations = new SparqlTuples(db, "select ?source ?target where {
      + "?source <http://lsdis.cs.uga.edu/projects/semdis/opus#cites> ?target}"
    citations foreach {
      case (citer, cited) =>
        computeGraph.addVertex[Document](citer)
        computeGraph.addVertex[Document](cited)
        computeGraph.addEdge[Citation](citer, cited)
    }
    computeGraph.execute(signalThreshold = 0)
  }
}
```
Signal/collect examples

Co-EM/wvRN/Harmonic fields

<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>
PageRank in PowerGraph

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

PageRankProgram(i)

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

gather/sum like a group by ... reduce or collect

j edge
i vertex

scatter is like a signal
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]
Partitioning Performance

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

---

**Cost**

**Construction Time**

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

Reduction in Runtime

- PageRank
- Collaborative Filtering
- Shortest Path

- Random
- Oblivious
- Greedy
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
• 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 \textbf{shard} on disk.
  – \textbf{sub-graph} = interval of vertices
PSW: Layout

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

Vertices 1..100  Vertices 101..700  Vertices 701..1000  Vertices 1001..10000

Shard 1  Shard 2  Shard 3  Shard 4

in-edges for vertices 1..100 sorted by source_id

Shards small enough to fit in memory; balance size of shards
PSW: Loading Sub-graph

Load subgraph for vertices 1..100

Vertices 1..100

Vertices 101..700

Vertices 701..1000

Vertices 1001..10000

What about out-edges?
Arranged in sequence in other shards

1. Load
2. Compute
3. Write

Load all in-edges in memory
PSW: Loading Sub-graph

Load subgraph for vertices 101..700

1. Load
2. Compute
3. Write

Load all in-edges in memory

Vertices 1..100
Shard 1

Vertices 101..700
Shard 2

Vertices 701..1000
Shard 3

Vertices 1001..10000
Shard 4

Out-edge blocks in memory

in-edges for vertices 1..100
sorted by source_id
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 $\rightarrow$ RDG, resilient distributed graph)
Idea 1: 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.
Operators

• Table (RDD) operators are inherited from Spark:

  map  reduce  sample
  filter  count  take
  groupBy  fold  first
  sort  reduceByKey  partitionBy
  union  groupByKey  mapWith
  join  cogroup  pipe
  leftOuterJoin  cross  save
  rightOuterJoin  zip  ...

...
class Graph [ V, E ] {
    def Graph(vertices: Table[ (Id, V) ],
              edges: Table[ (Id, Id, E) ])
    // Table Views -------------------------------
    def vertices: Table[ (Id, V) ]
    def edges: Table[ (Id, Id, E) ]
    def triplets: Table [ ((Id, V), (Id, V),
    // Transformations --------------------------------
    def reverse: Graph[V, E]
    def subgraph(pV: (Id, V) => Boolean,
                  pE: Edge[V,E] => Boolean): Graph[V,E]
    def mapV(m: (Id, V) => T): Graph[T,E]
    def mapE(m: Edge[V,E] => T): Graph[V,T]
    // Joins -------------------------------------
    def joinV(tbl: Table [((Id, T))]: Graph[(V, T), E ]
    def joinE(tbl: Table [((Id, Id, T))]: Graph[V, (E, T)]
    // Computation -------------------------------
    def mrTriplets(mapF: (Edge[V,E]) => List[((Id, T))],
                   reduceF: (T, T) => T): Graph[T, E]
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
Performance Comparisons

Live-Journal: 69 Million Edges

<table>
<thead>
<tr>
<th>Tool</th>
<th>Runtime (in seconds, PageRank for 10 iterations)</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
Summary

• Large immutable data structures on (distributed) disk, processing by sweeping through them 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