Big Data I: Graph Processing, Distributed Machine Learning

CS 240: Computing Systems and Concurrency
Lecture 21

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material. Selected content adapted from J. Gonzalez.
Patient presents abdominal pain. Diagnosis?

Patient ate which contains purchased from with E. Coli infection.
Big Data is Everywhere

- Machine learning is a reality
- How will we design and implement “Big Learning” systems?
We could use ....

 Threads, Locks, & Messages

 “Low-level parallel primitives”
Shift Towards Use Of Parallelism in ML

• Programmers repeatedly solve the same parallel design challenges:
  – Race conditions, distributed state, communication...

• Resulting code is very specialized:
  – Difficult to maintain, extend, debug...

Idea: Avoid these problems by using high-level abstractions
... a better answer:

MapReduce / Hadoop

Build learning algorithms on top of high-level parallel abstractions
MapReduce – Map Phase

Embarrassingly Parallel independent computation
No Communication needed
MapReduce – Map Phase

Image Features
MapReduce – Map Phase

Embarrassingly Parallel independent computation
MapReduce – Reduce Phase

Outdoor Picture Statistics

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outdoor Pictures</td>
<td>Indoor Pictures</td>
</tr>
<tr>
<td><img src="image1" alt="" /></td>
<td><img src="image2" alt="" /></td>
</tr>
</tbody>
</table>

Image Features

<table>
<thead>
<tr>
<th>Outdoor Pictures</th>
<th>Indoor Pictures</th>
</tr>
</thead>
<tbody>
<tr>
<td>![image3]</td>
<td>![image4]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU 1</th>
<th>CPU 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outdoor Pictures</td>
<td>Indoor Pictures</td>
</tr>
<tr>
<td>![image5]</td>
<td>![image6]</td>
</tr>
</tbody>
</table>
Map-Reduce for Data-Parallel ML

- Excellent for large data-parallel tasks!

Data-Parallel

Map Reduce

Feature Extraction
Algorithm Tuning
Basic Data Processing

Is there more to Machine Learning?
Exploiting Dependencies
Graphs are Everywhere

Social Network

Collaborative Filtering

Probabilistic Analysis

Text Analysis
Concrete Example

Label Propagation
Label Propagation Algorithm

- **Social Arithmetic:**
  
  50% What I list on my profile
  40% Sue Ann Likes
  + 10% Carlos Like

  I Like: 60% Cameras, 40% Biking

- **Recurrence Algorithm:**
  \[ Likes[i] = \sum_{j \in \text{Friends}[i]} W_{ij} \times Likes[j] \]

  – iterate until convergence

- **Parallelism:**
  – Compute all \( Likes[i] \) in parallel
Properties of Graph Parallel Algorithms

- Dependency Graph
- Factored Computation
- Iterative Computation

What I Like
What My Friends Like
Map-Reduce for Data-Parallel ML

• Excellent for large data-parallel tasks!

Data-Parallel

Graph-Parallel

MapReduce

Feature Extraction

Algorithm Tuning

Basic Data Processing

MapReduce?

Lasso

Label Propagation

Kernel Methods

Belief Propagation

Tensor Factorization

Deep Belief Networks

PageRank

Neural Networks
Problem: Data Dependencies

- MapReduce *doesn’t* efficiently express data dependencies
  - User *must code* substantial data transformations
  - Costly *data replication*
Iterative Algorithms

- MR doesn’t efficiently express iterative algorithms:

![Diagram showing iterative algorithms and data flow between CPUs and barriers.]
MapAbuse: Iterative MapReduce

• Only a subset of data needs computation:
MapAbuse: Iterative MapReduce

- System is **not optimized** for iteration:

![Diagram showing the relationship between startup and disk penalties in iterative MapReduce](image-url)
ML Tasks Beyond Data-Parallelism

Data-Parallel

Map Reduce

Graph-Parallel

Feature Extraction
Cross Validation
Computing Sufficient Statistics

Graphical Models
Gibbs Sampling
Belief Propagation
Variational Opt.

Collaborative Filtering
Tensor Factorization

Semi-Supervised Learning
Label Propagation
CoEM

Graph Analysis
PageRank
Triangle Counting
ML Tasks Beyond Data-Parallelism

Data-Parallel

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Graph-Parallel

GraphLab

Pregel

Spark

APACHE GIRAPH

Arabesque
• **Limited** CPU Power
• **Limited** Memory
• **Limited** Scalability
Distributed Cloud

Scale up computational resources!

Challenges:
- Distribute state
- Keep data consistent
- Provide fault tolerance
The GraphLab Framework

Graph Based Data Representation

Update Functions User Computation

Consistency Model
Data Graph

Data is associated with both vertices and edges

Graph:
• Social Network

Vertex Data:
• User profile
• Current interests estimates

Edge Data:
• Relationship
  (friend, classmate, relative)
Distributed Data Graph

Partition the graph across multiple machines:
Distributed Data Graph

- *Ghost vertices* maintain adjacency structure and replicate remote data.
Distributed Data Graph

• Cut efficiently using **HPC Graph partitioning tools** (ParMetis / Scotch / ...)

“ghost” vertices
The GraphLab Framework

Graph Based
Data Representation

Update Functions
User Computation

Consistency Model
Update Function

A user-defined **program**, applied to a **vertex**; transforms data in **scope** of vertex

---

Pagerank(scope){
  // Update vertex data
  // Reschedule Neighbors if needed
  if vertex.PageRank changes then
    reschedule_all_neighbors;
}

---

Update function applied (asynchronously) in parallel until convergence

Many schedulers available to prioritize computation

---

Selectively triggers computation at neighbors
Distributed Scheduling

Each machine maintains a schedule over the vertices it owns

Distributed Consensus used to identify completion
Ensuring Race-Free Code

• How much can computation overlap?
The GraphLab Framework

Graph Based Data Representation

Update Functions User Computation

Consistency Model
PageRank Revisited

Pagerank(scope) {
    vertex.PageRank = \( \alpha \)
    ForEach inPage:
        vertex.PageRank += \((1 - \alpha) \times \text{inPage.PageRank}\)

    ...
}

...
PageRank data races confound convergence

![Graph showing L1 Error to Truth vs Runtime (s)]
Racing PageRank: Bug

Pagerank(scope) {

vertex.PageRank = \alpha

ForEach inPage:

vertex.PageRank += (1 - \alpha) \times inPage.PageRank

...

}
Racing PageRank: Bug Fix

Pagerank(scope) {

    \texttt{tmp} = \alpha

    \textbf{ForEach} inPage:

    \texttt{tmp} += (1 - \alpha) \times \texttt{inPage.PageRank}

    \texttt{vertex.PageRank} = \texttt{tmp}

    ...

}
Throughput $\neq$ Performance

Higher Throughput
($\#\text{updates/sec}$)

No Consistency

Potentially Slower Convergence of ML
Serializability

For every parallel execution, there exists a sequential execution of update functions which produces the same result.
Serializability Example

Update functions one vertex apart can be run in parallel.

Stronger / Weaker consistency levels available

User-tunable consistency levels trades off parallelism & consistency

Overlapping regions are only read.

Edge Consistency
Distributed Consistency

• **Solution 1:** *Chromatic Engine*
  – Edge Consistency via *Graph Coloring*

• **Solution 2:** Distributed Locking
Chromatic Distributed Engine

Execute tasks on all vertices of color 0

Ghost Synchronization Completion + Barrier

Execute tasks on all vertices of color 0

Execute tasks on all vertices of color 1

Ghost Synchronization Completion + Barrier
Matrix Factorization

• Netflix Collaborative Filtering
  – Alternating Least Squares Matrix Factorization

Model: 0.5 million nodes, 99 million edges
Netflix Collaborative Filtering

![Graph showing speedup vs # machines for different levels of D (D=20, D=100) and ideal case.](image1)

- **Ideal**
- **D=100**
- **D=20**

![Graph showing runtime(s) for different frameworks (MPI, Hadoop, GraphLab) and # machines (D=20).](image2)

- **MPI**
- **Hadoop**
- **GraphLab**

(D = 20)
Distributed Consistency

• **Solution 1:** *Chromatic Engine*
  – Edge Consistency via *Graph Coloring*
  – Requires a graph coloring to be available
  – Frequent barriers → **inefficient** when only some vertices active

• **Solution 2:** Distributed Locking
Distributed Locking

Edge Consistency can be guaranteed through locking.

: RW Lock
Consistency Through Locking

Acquire write-lock on center vertex, read-lock on adjacent.

Performance problem: Acquiring a lock from a neighboring machine incurs a latency penalty
Simple locking

- lock scope 1
- scope 1 acquired
- update_function 1
- release scope 1
- Process request 1
- Process release 1
Pipelining hides latency

GraphLab Idea: **Hide latency using pipelining**
Distributed Consistency

• **Solution 1: Chromatic Engine**
  – Edge Consistency via **Graph Coloring**
  – Requires a graph coloring to be available
  – Frequent barriers $\rightarrow$ **inefficient** when only some vertices active

• **Solution 2: Distributed Locking**
  – Residual BP on 190K-vertex/560K-edge graph, 4 machines
  – No pipelining: 472 sec; **with pipelining:** 10 sec
How to handle machine failure?

• *What when machines fail?* How do we provide fault tolerance?

• Strawman scheme: *Synchronous snapshot* checkpointing
  1. Stop the world
  2. Write each machines’ state to disk
How can we do better, leveraging GraphLab’s consistency mechanisms?
Chandy-Lamport checkpointing

Step 1. Atomically one \textit{initiator}
(a) Turns red, (b) Records its own state
(c) sends \textit{marker} to neighbors

Step 2. On receiving marker \textbf{non-red}
node atomically: (a) Turns red,
(b) Records its own state, (c) Sends markers along all outgoing channels

Implemented within GraphLab as an \textbf{Update Function}
Async. Snapshot Performance

No Snapshot

No system performance penalty incurred from the slow machine!
Summary

• Two different methods of achieving **consistency**
  – Graph Coloring
  – Distributed Locking with pipelining

• **Efficient implementations**

• **Asynchronous FT** w/fine-grained Chandy-Lamport
Sunday topic:
Streaming Data Processing and Cluster Coordination