Big Data: Graph Processing

COS 418: Distributed Systems
Lecture 21
Kyle Jamieson

[Content adapted from J. Gonzalez]

Patient presents abdominal pain. Diagnosis?

Patient ate which contains purchased from also sold to which contains E. Coli infection with.

Big Data is Everywhere

flickr
6 Billion Flickr Photos

28 Million Wikipedia Pages

900 Million Facebook Users

YouTube
72 Hours a Minute YouTube

• 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

MapReduce – Reduce Phase

Outdoor Picture Statistics
Indoor Picture Statistics

Outdoor Pictures
Indoor Pictures

Image Features

Map-Reduce for Data-Parallel ML

• Excellent for large data-parallel tasks!

Data-Parallel

Is there more to Machine Learning?

Exploiting Dependencies
Graphs are Everywhere

Social Network

Collaborative Filtering

Probabilistic Analysis

Text Analysis

Users

Movies

Wiki

Docs

Words

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

Concrete Example

Label Propagation

Properties of Graph Parallel Algorithms

Dependency Graph

Factored Computation

Iterative Computation
Map-Reduce for Data-Parallel ML

- Excellent for **large data-parallel** tasks!

**MapReduce**

- Feature Extraction
- Algorithm Tuning
- Basic Data Processing

**MapReduce?**

- Lasso
- Label Propagation
- Kernel Methods
- Belief Propagation
- PageRank
- Tensor Factorization
- Deep Belief 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:

**MapAbuse: Iterative MapReduce**

- Only a **subset** of data needs computation:
MapAbuse: Iterative MapReduce

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

ML Tasks Beyond Data-Parallelism

Data-Parallel

<table>
<thead>
<tr>
<th>Map Reduce</th>
<th>Graph-Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Extraction</td>
<td>Cross Validation</td>
</tr>
<tr>
<td>Computing Sufficient Statistics</td>
<td></td>
</tr>
</tbody>
</table>

Distributed Cloud

- **Limited** CPU Power
- **Limited** Memory
- **Limited** Scalability

Distributed Cloud

- **Challenges:**
  - Distribute state
  - Keep data consistent
  - Provide fault tolerance

Scale up computational resources!
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:

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

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

The GraphLab Framework

- Graph Based
  Data Representation

Update Function

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

Update function applied (asynchronously) in parallel until convergence

Many schedulers available to prioritize computation

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?

PageRank Revisited

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

    ...
}
```

PageRank data races confound convergence
**Racing PageRank: Bug**

```java
Pagerank(scope) {
    vertex.PageRank = α
    ForEach inPage:
        vertex.PageRank += (1 - α) × inPage.PageRank
    ...
}
```

**Racing PageRank: Bug Fix**

```java
Pagerank(scope) {
    tmp = α
    ForEach inPage:
        tmp += (1 - α) × inPage.PageRank
    vertex.PageRank = tmp
    ...
}
```

**Throughput != Performance**

- Higher Throughput (#updates/sec)
- No Consistency
- Potentially Slower Convergence of ML

**Serializable**

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

- Parallel
  - CPU 1
  - CPU 2
- Sequential
  - Single CPU
Serializability Example

Overlapping regions are only read.

Update functions one vertex apart can be run in parallel.

Stronger / Weaker consistency levels available

User-tunable consistency levels trades off parallelism & consistency

Edge Consistency

Distributed Consistency

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

• Solution 2: Distributed Locking

Chromatic Distributed Engine

Chromatic Distributed Engine

Matrix Factorization

• Netflix Collaborative Filtering
  – Alternating Least Squares Matrix Factorization

Model: 0.5 million nodes, 99 million edges

Netflix

Movies

Users

Movies

D

D
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.

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 -> Process request 1
scope 1 acquired -> update_function 1
release scope 1 -> Process release 1

Pipelining hides latency

GraphLab Idea: Hide latency using pipelining

lock scope 1 -> Process request 1
lock scope 2
lock scope 3
scope 1 acquired
scope 2 acquired
scope 3 acquired
update_function 1
release scope 1
update_function 2
release scope 2
Process release 1

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
  – 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
Snapshot Performance

How can we do better, leveraging GraphLab’s consistency mechanisms?

Async. Snapshot Performance

Chandy-Lamport checkpointing

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

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

First-in, first-out channels

First-in, first-out channels

Implemented within GraphLab as an Update Function

Summary

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

• Efficient implementations

• Asynchronous FT w/fine-grained Chandy-Lamport

Performance Efficiency Scalability

Useability
Friday Precept:
Roofnet performance
More Graph Processing

Monday topic:
Streaming Data Processing