MapReduce & GraphLab: Programming Models for Large-Scale Parallel/Distributed Computing

Motivation

- The Age of “Big Data”
  - ~45 Billion Webpages
  - ~1.06 Billion Facebook Users
  - ~24 Million Wikipedia Pages
  - ~6 Billion Flickr Photos

- Infeasible to analyze on a single machine
- Solution: Distribute across many computers

Outline

- Motivation
- MapReduce Overview
  - Design Issues & Abstractions
  - Examples and Results
  - Pros and Cons
- Graph Lab
  - Graph Parallel Abstraction
  - MapReduce vs Pregel vs GraphLab
  - Implementation Issues
  - Pros and Cons

Motivation

- Challenges: We repeatedly solve the same system-level problems
  - Communication/Coordination among the nodes
  - Synchronization, Race Conditions
  - Load Balancing, Locality, Fault Tolerance, ...
- Need a higher level programming Model
  - That hides these messy details
  - Applies to a large number of problems
MapReduce: Overview

- A programming model for large-scale data-parallel applications
  - Introduced by Google (Dean & Ghemawat, OSDI'04)
- Petabytes of data processed on clusters with thousands of commodity machines
  - Suitable for programs that can be decomposed in many embarrassingly parallel tasks
- Hides low level parallel programming details

Example: Count word frequencies from web pages

- Input: Web documents
  - Key = web document URL, Value = document content
- Output: Frequencies of individual words

Steps 1: specify a Map function

<table>
<thead>
<tr>
<th>Input:</th>
<th>&lt;key=url, value=content&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;www.abc.com&quot;, abc ab cc ab</td>
</tr>
</tbody>
</table>

Output:

<table>
<thead>
<tr>
<th>&lt;key=word, value=partialCount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;, 1</td>
</tr>
<tr>
<td>&quot;ab&quot;, 1</td>
</tr>
<tr>
<td>&quot;cc&quot;, 1</td>
</tr>
<tr>
<td>&quot;ab&quot;, 1</td>
</tr>
</tbody>
</table>

Steps 2: specify a Reduce function

- Collect the partial sums provided by the map function
- Compute the total sum for each individual word

Input: Intermediate files <key=word, value=partialCount>

```
key = "abc"  values = 1
key = "ab"   values = 1, 1
key = "cc"   values = 1
```

Output:

```
key = "abc", 2
key = "ab", 1
key = "cc", 1
```

Example: Count word frequencies from web pages
Example code: Count word

```java
void map(String key, String value):
    // key: webpage url
    // value: webpage contents
    for each word w in value:
        EmitIntermediate(w, "1");

void reduce(String key, Iterator partialCounts):
    // key: a word
    // partialCounts: a list of aggregated partial counts
    int result = 0;
    for each pc in partialCounts:
        result += ParseInt(pc);
    Emit(AsString(result));
```

How MapReduce Works?

Ref: J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI, 2004

Implementation Details: Scheduling

- One Master, many workers
  - MapReduce Library splits input data into \( M \) pieces (16MB or 64MB per piece, uses GFS)
  - Master assigns each idle worker to a map or reduce task
  - Worker completes the map task, buffers the intermediate (key, value) in memory, and periodically writes to local disk
  - Location of buffered pairs are returned to Master
  - Master assigns completed map tasks to Reduce workers
    - Reduce worker reads the intermediate files using RPC
    - Sorts the keys and performs reduction

Fault Tolerance

- Worker Failure: Master pings the workers periodically
  - If no response, then master marks the worker as failed
  - Any map task or reduce task in progress is marked for rescheduling
  - Completed reduce tasks don't have to be recomputed
- Master Failure
  - Master writes periodic checkpoints to GFS. On failure, new master recovers to that checkpoint and continues
  - Often not handled, aborts if master fails (failure of master is less probable)
Locality

- Bandwidth is an important resource
  - Communicating large datasets to worker nodes can be very expensive
- Do not transfer data to worker
  - Assign task to the worker that has the data locally
- Create multiple replications of data (Typically 3)
- Master assigns to one of these computers having the data in local file system

Other Refinements

- Task Granularity:
  - M map tasks, R reduce tasks
  - We want to make M and R larger
    - Dynamic load balancing
    - Faster recovery from failure
    - BUT, increases the number of scheduling decisions increases with M and R
  - Finally we’ll get R output files. So R should not be too large
    - Typical settings: for 2000 worker machines:
      - M = 200,000 and R = 5000

Other Refinements: Backup Tasks

- Some machines can be extremely slow ("straggler")
  - Perhaps a bad disk that frequently encounters correctable errors
- Solution: Backup tasks
  - Near the end of map reduce, master schedules some backup tasks for each of the remaining tasks
  - A task is marked as complete if either the primary or the backup execution completes

Results: Sorting

M=15k, R=4k, ~1800 machines

Figure 3: Data transfer rates over time for different executions of the sort program
Example: Word Frequency: MAP

```cpp
#include "mapreduce/mapreduce.h"
// User's map function
class WordCounter : public Mapper {
public:
  virtual void Map(const MapInput& input) {
    // perform map operation, parse input ...
    // for each word
    Emit(word, "1")
  }
};
REGISTER_MAPPER(WordCounter);
```

Word Frequency: MAIN (Simplified)

```cpp
int main(int argc, char** argv) {
  MapReduceSpecification spec;
  // Store list of input files into "spec"
  MapReduceInput* input = spec.add_input();
  input->set_mapper_class("WordCounter");
  // specify output files
  MapReduceOutput* out = spec.output();
  out->set_reducer_class("Adder");
  // Tuning parameters: use at most 2000
  spec.set_machines(2000);
  // Now run it
  MapReduceResult result;
  MapReduce(spec, &result);
  return 0;
}
```

Word Frequency: REDUCE

```cpp
// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();
    }
    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};
REGISTER_REDUCER(Adder);
```

MapReduce Instances at Google

Ref: PACT 06' Keynote slides by Jeff Dean, Google, Inc.
Pros: MapReduce

- **Simplicity** of the model
  - Programmers specifies few simple methods that focuses on the functionality not on the parallelism
  - Code is generic and portable across systems
- **Scalability**
  - Scales easily for large number of clusters with thousands of machines
- **Applicability** to many different systems and a wide variety of problems
  - Distributed Grep, Sort, Inverted Index, Word Frequency count, etc.

Cons: MapReduce

- Restricted programming constructs (only map & reduce)
- Does not scale well for dependent tasks (for example Graph problems)
- Does not scale well for iterative algorithms (very common in machine learning)

Summary: MapReduce

- **MapReduce**
  - Restricted but elegant solution
  - High level abstraction
  - Implemented in C++
- Many Open-source Implementation
  - Hadoop (Distributed, Java)
  - Phoenix, Metis (Shared memory, for multicore, C++)

GraphLab
GraphLab: Motivation

- MapReduce is great for Data Parallel applications
  - Can be easily decomposed using map/reduce
  - Assumes independence among the tasks
- Independence assumption can be too restrictive
- Many interesting problems involve graphs
  - Need to model dependence/interactions among entities
  - Extract more signal from noisy data
  - MapReduce is not well suited for these problems

Graph-based Abstraction

- Machine learning practitioners typically had two choices:
  - Simple algorithms + “Big data” vs
  - Powerful algorithms (with dependency)+ “Small data”
- Graph-based Abstraction
  - Powerful algorithms + “Big data”

Graph parallel programming models:

- Pregel: Bulk Synchronous Parallel Model (Google, SIGMOD 2010)
- GraphLab: asynchronous model [UAI 2010, VLDB 2012]

Bulk Synchronous Parallel Model: Pregel

[Małewicz et al. '2010]

Tradeoffs of the BSP Model

- Pros:
  - Scales better than MapReduce for Graphs
  - Relatively easy to build
  - Deterministic execution
- Cons:
  - Inefficient if different regions of the graph converge at different speed
  - Can suffer if one task is more expensive than the others
  - Runtime of each phase is determined by the slowest machine
Synchronous vs Asynchronous Belief Propagation

[Gonzalez, Low, Guestrin. '09]

GraphLab vs. Pregel (Page Rank)

[Low et al. PVLDB’12]

GraphLab Framework: Overview

- GraphLab allows asynchronous iterative computation
- The GraphLab abstraction consists of 3 key elements:
  - Data Graph
  - Update Functions
  - Sync Operation
- We explain GraphLab using PageRank example

Case Study: PageRank

- Iterate:
  \[ R[i] = \alpha + (1 - \alpha) \sum_{(j,i) \in E} \frac{1}{L[j]} R[j] \]
- Where:
  - \( \alpha \) is the random reset probability
  - \( L[j] \) is the number of links on page \( j \)

PageRank (25M Vertices, 355M Edges, 16 processors)

http://graphlab.org/powergraph-presented-at-osdi/
1. Data Graph

- Data Graph: a directed acyclic graph $G = (V, E, D)$
  - Data $D$ refers to model parameters, algorithm states and other related data

  **Vertex Data:**
  - User profile text
  - Current interests estimates

  **Edge Data:**
  - Similarity weights

Graph:
- Social Network

2. Update Functions

**An update function** is a user-defined program (similar to a map) applied to a vertex, transforms the data in the scope of the vertex.

```cpp
struct pagerank : public iupdate_functor<graph, pagerank> {
    void operator()(icontext_type& context) {
        double sum = 0;
        foreach (edge_type edge, context.in_edges())
            sum += 1.0 / context.num_out_edges(edge.source()) * context.vertex_data(edge.source());

        double& rank = context.vertex_data(v);
        double old_rank = rank;
        rank = RESET_PROB + (1.0 - RESET_PROB) * sum;
        double residual = abs(rank - old_rank);
        if (residual > EPSILON)
            context.reschedule_out_neighbors_of(v);
    }
};
```

1. Data Graph

- PageRank: $G = (V, E, D)$
  - Each vertex $v$ corresponds to a webpage
  - Each edge $(u, v)$ corresponds to a link from $u \rightarrow v$
  - Vertex data $D_v$ stores the rank of the webpage i.e. $R(v)$
  - Edge data: $D_{u \rightarrow v}$ stores the weight of the link $(u \rightarrow v)$

$$R[v] \leftarrow \alpha + (1 - \alpha) \sum_{u \in N[v]} W_{uv} \times R[u];$$
3. Sync Operation

- Global operation, usually performed periodically in the background
  - Useful for maintaining some global statistics of the algorithm
    - Example: PageRank may want to return a list of 100 top ranked web pages
- Determine the global convergence criteria. For example, estimate log-likelihood.
  - Estimate total log-likelihood for Expectation Maximization
- Similar to Reduce functionality in MapReduce

GraphLab: Hello World!

```cpp
#include <graphlab.hpp>
int main(int argc, char** argv) {
    graphlab::mpi_tools::init(argc, argv);
    Graphlab::distributed_control dc;
    dc.cout() << "Hello World!\n";
    graphlab::mpi_tools::finalize();
}
```

- Let the file name: my_first_app.cpp
- Use “make” to build
- Execute: mpiexec -n 4 ./my_first_app

dc.cout() provides a wrapper around standard std::cout
When used in a distributed environment, only one copy will print, even though all machines execute it.

Scheduling

**Algorithm 2: GraphLab Execution Model**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( v \leftarrow \text{RemoveNext}(\mathcal{T}) )</td>
</tr>
<tr>
<td>2</td>
<td>((\mathcal{T}, S_v) \leftarrow f(v, S_v))</td>
</tr>
<tr>
<td>3</td>
<td>( \mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}' )</td>
</tr>
</tbody>
</table>

Output: Modified Data Graph \( G = (V, E, D') \)

[Low et al. PVLDB’12]

GraphLab Tutorials

- For more interesting examples, check:
  - [http://docs.graphlab.org/using_graphlab.html](http://docs.graphlab.org/using_graphlab.html)
- A step by step tutorial for implementing PageRank in GraphLab
Few Additional Details

- Fault Tolerance using checkpointing
- The GraphLab described here is GraphLab 2.1
  - GraphLab for distributed systems
- The first version was proposed only for multicore systems
- Recently, PowerGraph was proposed
  - GraphLab on Cloud

Tradeoffs of GraphLab

- **Pros:**
  - Allows dynamic asynchronous scheduling
  - More expressive consistency model
  - Faster and more efficient runtime performance
- **Cons:**
  - Non-deterministic execution
  - Substantially more complicated to implement

Summary

- **MapReduce:** efficient for independent tasks
  - Simple framework
  - Independence assumption can be too restrictive
  - Not scalable for graphs or dependent tasks, or iterative tasks
- **Pregel:** Bulk Synchronous Parallel Models
  - Can model dependent and iterative tasks
  - Easy to Build, Deterministic
  - Suffers from inefficiencies due to synchronous scheduling
- **GraphLab:** Asynchronous Model
  - Can model dependent and iterative tasks
  - Fast, efficient, and expressive
  - Introduces non-determinism, relatively complex implementation

References

- **MapReduce:** Simplified Data Processing on Large Clusters, Jeffrey Dean & Sanjay Ghemawat, OSDI 2004.
- **GraphLab:** A new framework for parallel machine learning, Low et al., UAI 2010.
- **GraphLab:** a distributed framework for machine learning in the cloud, Low et al., PVLDB 2012.
- **Pregel:** a system for large-scale graph processing, Malewicz et al., SIGMOD 2010.
Disclaimer

- Many figures and illustrations were collected from Carlos Guestrin's GraphLab tutorials
  
  http://docs.graphlab.org/using_graphlab.html

Additional GraphLab Implementation Issues

Scheduler

Consistency Model

Scheduling

The scheduler determines the order that vertices are updated

The process repeats until the scheduler is empty

- What happens for boundary vertices (across machines)?
- Data is classified as edge data and vertex data
- Partition a huge graph across multiple machines
  - Ghost vertices (along the cut) maintains adjacency information
  - Graph must be cut efficiently. Use parallel graph partitioning tools (ParMetis)
Consistency Model

- Race conditions may occur if updating shared data
- If overlapping computations run simultaneously

Consistency Model: How to Avoid Race?

- Ensure update functions for two vertices simultaneously operate only if
two scopes do not overlap

- Three consistency models:
  - Full consistency
  - Edge consistency
  - Vertex consistency

Consistency vs Parallelism

(b) Consistency Models

(c) Consistency and Parallelism
Consistency Model: Implementation

- **Option 1: Chromatic Engine:**
  - Graph coloring: neighboring vertices have different colors
  - Simultaneous update only for vertices with the same color

![Graph Coloring Diagram]

Consistency using Distributed Locks

- **Distributed Locking**
  - Non-blocking locks allow computation to proceed while locks/data are requested.
  - Request locks in a canonical order to avoid deadlock

![Distributed Locking Diagram]