MapReduce for Big Data

CAP6676: Knowledge Representation
Instructor: Guo-Jun Qi
Big Data

• Large-scale databases are created in modern computing tasks
  • Web data
  • E-commerce
  • Bank/Credit Card
  • Social Media
  • Streamed live videos
  • And many others
Divide and Conquer

- A natural framework to compute with big data is “divide and conquer”
An example: Distributed Grep

Very big data → `grep` → `matches` → `cat` → All matches
More example: Distributed Word Count
Parallelization Challenges

• How do we assign task units to workers? (Scheduling)
• What if we have more task units than workers? (Scheduling)
• What if workers need to share partial results? (Synchronization)
• How do we aggregate partial results? (combining)
• How do we know all the workers have finished? (Synchronization)
• What if workers die unexpectedly? (handling failures)
Common Theme

• Parallelization problems arise from
  • Communication between workers
    • If all workers have finished
    • Intermediate results
    • Shared variables
  • Access to shared resources

• A synchronization mechanism is indispensable.
Managing Multiple Workers in parallel

• Difficult because we do not know
  • The order in which workers are scheduled
  • When workers interrupt each other
  • The order in workers access shared data

• So we need
  • Lock/unlock shared data
  • Conditional variables
  • Barriers

• Lots of unexpected problems would happen
  • Deadlock, race conditions
A black box for concurrency computing

• Right level of abstraction for
  • Multi-core/cluster environment

• Hide system-level details from the developers
  • Handling communication issues, scheduling the workers etc.

• Separating what from how
  • Developers only need to specify the computation to be performed
  • Runtime framework handles how to execute in parallel
MapReduce

• Facts
  • Google has used it to process its “big-data” (~20000 peta bytes per day)
  • Users specify the computation in terms of a map and a reduce function
  • Underlying runtime automatically parallelizes the computation across large-scale clusters of machines
  • It also handles machine failures, efficient communications and performance issues
MapReduce refers to

• A programming model, or
• The runtime environment
• The specific implementation of your tasks
Typical Big-data problem

• **Map**
  • Iterate over a large number of records
  • Extract something of interest from each, or apply a specific computation to each record

• **Reduce**
  • Shuffle and sort intermediate results from Map
  • Aggregate intermediate results
  • Generate final output

• A functional abstraction for these two operations
MapReduce Programming Model

\[
\text{map} \ (k, v) \rightarrow \left[ (k', v') \right] \\
\text{reduce} \ (k', [v']) \rightarrow \left[ (k', v') \right]
\]

• Map receives each record, and emits intermediate results from the input record
• All values with the same key are sent to the same reducer
• Reducer outputs final result associated with each input key \( k' \).
• The frame handles details.
Architecture Overview

User -> Job Tracker

Job Tracker -> Slave Node 1, Slave Node 2, Slave Node N

Slave Node 1, Slave Node 2, Slave Node N -> Task Tracker

Task Tracker -> Workers

Worker Nodes: 
- Slave Node 1
- Slave Node 2
- Slave Node N
MapReduce Implementations

• Google MapReduce
  • Not available outside Google

• Hadoop
  • An open source implementation in Java
  • Developed by Yahoo
  • An Apache Project
Who uses Hapood?

• Amazon/A9
• Facebook
• Google
• IBM
• Joost
• Last.fm
• New York Times
• PowerSet
• Veoh
• Yahoo!
• …
Distributed File System

• Move workers to the data, rather than moving data to workers
  • Store data on the local disks of nodes in the cluster
  • Start up the workers on the node that has the local data

• Why splitting the data into DFS?
  • Limited RAM to fit all the data in the memory

• Examples
  • GFS (Google File System) for Google’s MapReduce
  • HDFS (Hadoop Distributed File System) for Hadoop
Distributed File Systems

• Chunk Servers
  • Files are split into contiguous data chunks
  • Typical size: 16-64 MB
  • Each chunk replicated (2X or 3X), and keep replicas in different racks

• Master node
  • A.s.a, name nodes in HDFS
  • Stores metadata, e.g., where to find the data chunks
  • Replicated

• Client node
  • Talks to master node to find data chunks
  • Connects directly to chunk servers to access data
Hadoop HDFS

NameNode:
Stores metadata only

METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5

DataNodes: Store blocks from files
Hadoop Cluster Architecture

From Jimmy Lin's slides
Map+Reduce

• Map:
  – Accepts *input* key/value pair
  – Emits *intermediate* key/value pair

• Reduce:
  – Accepts *intermediate* key/value* pair
  – Emits *output* key/value pair
The Map Step
The Reduce Step
The Reduce Step
MapReduce

• Input: A set of key/value pairs
• User supplies two functions
  
  \[
  \text{map}(k,v) \rightarrow \text{list}(k_1,v_1) \\
  \text{reduce}(k_1, \text{list}(v_1)) \rightarrow (k_1,v_2)
  \]

• \((k_1,v_1)\) is an intermediate key/value pairs generated from the original input record \((k,v)\)
• Output is \((k_1,v_2)\) pairs by aggregating all the intermediate results with the same key \(k_1\).
Word Count

• A large collection of documents
• Count the times each distinct word appears in the collection
Word Count Execution

Input: the quick brown fox
      the fox ate the mouse
      how now brown cow

Map: the, 1  brown, 1  fox, 1
     the, 1  fox, 1  the, 1
     how, 1  now, 1  brown, 1

Shuffle & Sort: the, 1  brown, 1  fox, 1
                fox, 1  the, 1
                now, 1  how, 1

Reduce: brown, 2  fox, 2  how, 1  now, 1  the, 3

Output: brown, 2  fox, 2  how, 1  now, 1  the, 3
        ate, 1  cow, 1  mouse, 1  quick, 1
MapReduce operations for Word Count

**map** (key, value):

// key: document name; value: text of document
for each word w in value:
    emit(w, 1)

**reduce** (key, values):

// key: a word; value: an iterator over counts
result = 0
for each count v in values:
    result += v
emit(result)
Combiners

• A map task may produce many key/value pairs with the same key
• Save communication overhead by pre-aggregating at mapper
• Decreases the size of intermediate data
• Example: local counting for word count:

```python
def combiner(key, values):
    output(key, sum(values))
```
Word Count with Combiner

```
the quick brown fox
the fox ate the mouse
how now brown cow

Map & Combine
- the, 1
the, 2
brown, 1
fox, 1
fox, 1

Shuffle & Sort
- quick, 1
how, 1
now, 1

Reduce
- the, 3
brown, 2
fox, 2
how, 1
now, 1

Output
- ate, 1
cow, 1
mouse, 1
quick, 1
```
MapReduce K-means

• Partition \{x_1, \ldots, x_n\} of n examples into K clusters
• Initialization
  • Specify the initial cluster centers
• Iteration until no change
  • For each object \(x_i\)
    • Calculate its distance to the K cluster centers
    • Assign it to the cluster whose center is the closest to the object
  • Update the cluster centers based on the current assignment
Traditional K-means

AssignCluster():
- For each point p
  Assign p the closest c

UpdateCentroids():
- For each cluster
  Update cluster center

Kmeans():
- While not converge:
  - AssignCluster()
  - UpdateCentroids()
MapReduce K-means

Map: assign each \( p \) to closest centroids
Reduce: update each centroid with its new location (total, count)

KmeansIter()
Map(p) // Assign Cluster
- For c in clusters:
  - If dist(p,c)<minDist,
    then minC=c, minDist = dist(p,c)
  - Emit(minC.id, (p, 1))
Reduce() //Update Centroids
- For all values (p, c):
  - total += p; count += c;
- Emit(key, (total, count))
MapReduce K-means

- Runs multiple iteration jobs using the following Mapper+Combiner+Reducer

- Mapper: assign cluster
  - Input: data points
  - Intermediate output: (cluster id, data)

- Reducer: update cluster centers
  - Input: (cluster id, *data_list)
  - Output: (cluster id, cluster center)

- Combiner
  - Input: (cluster id, *data_list)
  - Output: (cluster id, (partial sum, number of points))
Summary

• MapReduce programming model
• How to design map, reduce, combiner
• MapReduce implementation of WordCount and K-means
  • Amenable to one iteration task
  • OK with multiple iteration algorithms
    • If small shared info needs to be synchronized across iterations (e.g., cluster centers)
  • Not good for algorithms with large shared info because of heavy synchronization overhead (e.g., SVMs, NNs)
Course Project

- Deadline for report submission
  - December 11st

- Presentation
  - Shall we?

- Oral/Poster Presentation