Lecture 10 — Parallel Computing with MapReduce

Jin, Hai

School of Computer Science and Technology
Huazhong University of Science and Technology
Outline

- MapReduce Programming Model
- Typical Problems Solved by MapReduce
- MapReduce Examples
- A Brief History
- MapReduce Execution Overview
- Hadoop
Motivation: Large Scale Data Processing

- Want to process lots of data (>1TB)
- Want to parallelize across hundreds/thousands of CPUs
- ... Want to make this easy
MapReduce

- “A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.”

- More simply, MapReduce is
  - A parallel programming model and associated implementation

*Dean and Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Google Inc.*
Some MapReduce Terminology

- **Job** – A “full program” - an execution of a Mapper and Reducer across a data set
- **Task** – An execution of a Mapper or a Reducer on a slice of data
  - a.k.a. Task-In-Progress (TIP)
- **Task Attempt** – A particular instance of an attempt to execute a task on a machine
Terminology Example

- Running “Word Count” across 20 files is one job
- 20 files to be mapped imply 20 map tasks + some number of reduce tasks
- At least 20 map task attempts will be performed… more if a machine crashes, etc.
Task Attempts

- A particular task will be attempted at least once, possibly more times if it crashes
  - If the same input causes crashes over and over, that input will eventually be abandoned

- Multiple attempts at one task may occur in parallel with speculative execution turned on
  - Task ID from TaskInProgress is not a unique identifier
MapReduce Programming Model

- Process data using special `map()` and `reduce()` functions

  - The `map()` function is called on every item in the input and emits a series of intermediate key/value pairs
  - All values associated with a given key are grouped together
  - The `reduce()` function is called on every unique key, and its value list, and emits a value that is added to the output
map

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line)

- map() produces one or more intermediate values along with an output key from the input

  - map  (in_key, in_value) ->
    (out_key, intermediate_value) list
reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- `reduce()` combines those intermediate values into one or more *final values* for that same output key

```python
- reduce (out_key, intermediate_value list) -> out_value list
```
reduce

reduce (out_key, intermediate_value list) -> out_value list

initial

returned
MapReduce Architecture

- **Input key-value pairs**

  - Data store 1
  - Map
    - (key 1, values...)
    - (key 2, values...)
    - (key 3, values...)
  - Data store n
  - Map
    - (key 1, values...)
    - (key 2, values...)
    - (key 3, values...)

- **Barrier**
  - Aggregates intermediate values by output key

- **Reduce**
  - Key 1, intermediate values
    - Final key 1 values
  - Key 2, intermediate values
    - Final key 2 values
  - Key 3, intermediate values
    - Final key 3 values
MapReduce Programming Model

- More formally,
  - Map(k1,v1) --> list(k2,v2)
  - Reduce(k2, list(v2)) --> list(v2)
MapReduce in One Picture

Tom White, *Hadoop: The Definitive Guide*
MapReduce Runtime System

1. Partitions input data
2. Schedules execution across a set of machines
3. Handles machine failure
4. Manages interprocess communication
Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can’t start until map phase is completely finished
Locality

- Master program divides up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks
Fault Tolerance

- Master detects worker failures
  - Re-executes completed & in-progress map() tasks
  - Re-executes in-progress reduce() tasks

- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution
  - Effect: Can work around bugs in third-party libraries!
Optimizations

- No reduce can start until map is complete
  - A single slow disk controller can rate-limit the whole process

- Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish

- “Combiner” functions can run on same machine as a mapper

- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth
Optimizations
MapReduce Benefits

- Greatly reduces parallel programming complexity
  - Reduces synchronization complexity
  - Automatically partitions data
  - Provides failure transparency
  - Handles load balancing
Outline

- MapReduce Programming Model
- Typical Problems Solved by MapReduce
- MapReduce Examples
- A Brief History
- MapReduce Execution Overview
- Hadoop
MapReduce: High Level

MapReduce job submitted by client computer

Master node

JobTracker

Slave node

TaskTracker

Task instance
Nodes, Trackers, Tasks

- Master node runs *JobTracker* instance, which accepts *Job* requests from clients

- *TaskTracker* instances run on slave nodes

- *TaskTracker* forks separate Java process for task instances
Typical Problems Solved by MapReduce

- Read a lot of data
- **Map**: extract something you care about from each record
- **Shuffle** and **Sort**
- **Reduce**: aggregate, summarize, filter, or transform
- Write the results

- Outline stays the same, but **map** and **reduce** change to fit the problem
Outline

- MapReduce Programming Model
- Typical Problems Solved by MapReduce
- MapReduce Examples
- A Brief History
- MapReduce Execution Overview
- Hadoop
**MapReduce Examples**

- **Word frequency**

![Diagram](Image)

- Document (`doc`)
- Map function
- Reduce function
- Runtime System

Input: `doc` -> Map: `<word,1>`, `<word,1>`, `<word,1>` -> Reduce: `<word,3>`
Example: Count Word Occurrences

```java
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += parseInt(v);
    Emit(AsString(result));
```
Example: Count Word Occurrences

The overall MapReduce word count process

Input

Splitting

Mapping

Shuffling

Reducing

Final result

Deer Bear River
Car Car River
Deer Car Bear

Deer, 1
Bear, 1
River, 1

Car, 1
Car, 1
River, 1

Bear, 2
Bear, 1

Car, 1
Car, 1
Car, 1

Deer, 2
Deer, 1

River, 1
River, 1

Bear, 2
Car, 3
Deer, 2
River, 2
MapReduce Examples

- Distributed grep
  - Map function emits `<word, line_number>` if word matches search criteria
  - Reduce function is the identity function

- URL access frequency
  - Map function processes web logs, emits `<url, 1>`
  - Reduce function sums values and emits `<url, total>`
Outline

- MapReduce Programming Model
- Typical Problems Solved by MapReduce
- MapReduce Examples
- A Brief History
- MapReduce Execution Overview
- Hadoop
MapReduce is a new use of an old idea in Computer Science

- Map: Apply a function to every object in a list
  - Each object is independent
    - Order is unimportant
    - Maps can be done in parallel
  - The function produces a result

- Reduce: Combine the results to produce a final result

You may have seen this in a Lisp or functional programming course
Outline

- MapReduce Programming Model
- Typical Problems Solved by MapReduce
- MapReduce Examples
- A Brief History
- MapReduce Execution Overview
- Hadoop
1. The user program, via the MapReduce library, shards the input data

* Shards are typically 16-64MB in size
Getting Data To The Mapper

Input file

InputSplit

RecordReader

Mapper

(intermediates)

Input file

InputSplit

RecordReader

Mapper

(intermediates)

Input file

InputSplit

RecordReader

Mapper

(intermediates)

Input file

InputSplit

RecordReader

Mapper

(intermediates)

InputFormat
MapReduce Execution Overview

2. The user program creates process copies distributed on a machine cluster. One copy will be the “master” and the others will be worker threads.
MapReduce Execution Overview

3. The master distributes $M$ map and $R$ reduce tasks to idle workers

- $M ==$ number of shards
- $R ==$ the intermediate key space is divided into $R$ parts
Partition and Shuffle

Mapper

(intermediates)

Mapper

(intermediates)

Mapper

(intermediates)

Mapper

(intermediates)

Reducer

Reducer

Reducer

Partitioner

Partitioner

Partitioner

Partitioner

shuffling

(intermediates)

(intermediates)

(intermediates)
4. Each map-task worker reads assigned input shard and outputs intermediate key/value pairs

- Output buffered in RAM
MapReduce Execution Overview

5. Each worker flushes intermediate values, partitioned into $R$ regions, to disk and notifies the Master process.
6. Master process gives disk locations to an available reduce-task worker who reads all associated intermediate data.
MapReduce Execution Overview

7. Each reduce-task worker sorts its intermediate data. Calls the reduce function, passing in unique keys and associated key values. Reduce function output appended to reduce-task’s partition output file.
8. Master process wakes up user process when all tasks have completed. Output contained in $R$ output files
Writing The Output

Reducer

RecordWriter

output file

Reducer

RecordWriter

output file

Reducer

RecordWriter

output file

OutputFormat
MapReduce Execution Overview

☐ Fault Tolerance

➢ Master process periodically pings workers
  • Map-task failure
    ✔ Re-execute
      ▶ All output was stored locally
  • Reduce-task failure
    ✔ Only re-execute partially completed tasks
      ▶ All output stored in the global file system
Outline

- MapReduce Programming Model
- Typical Problems Solved by MapReduce
- MapReduce Examples
- A Brief History
- MapReduce Execution Overview
- Hadoop
Hadoop

- Open source MapReduce implementation

<table>
<thead>
<tr>
<th>Google calls it</th>
<th>Hadoop equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td>Hadoop</td>
</tr>
<tr>
<td>GFS</td>
<td>HDFS</td>
</tr>
<tr>
<td>Bigtable</td>
<td>HBase</td>
</tr>
<tr>
<td>Chubby</td>
<td>(nothing yet… but planned)</td>
</tr>
</tbody>
</table>
HDFS Architecture

Metadata ops

Namenode

Metadata (Name, replicas, ...): /home/foo/data, 3, ...

Client

Read

Datanodes

Block ops

Datanodes

Replication

Rack 1

Write

Client

Rack 2

Blocks
Hadoop Related Projects

- **Ambari**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters which includes support for Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig and Sqoop. Ambari also provides a dashboard for viewing cluster health such as heat maps and ability to view MapReduce, Pig and Hive applications visually along with features to diagnose their performance characteristics in a user-friendly manner.

- **Avro**: A data serialization system

- **Cassandra**: A scalable multi-master database with no single points of failure

- **Chukwa**: A data collection system for managing large distributed systems

- **HBase**: A scalable, distributed database that supports structured data storage for large tables (NoSQL)

- **Hive**: A data warehouse infrastructure that provides data summarization and ad hoc querying

- **Mahout**: A Scalable machine learning and data mining library

- **Pig**: A high-level data-flow language and execution framework for parallel computation

- **ZooKeeper**: A high-performance coordination service for distributed applications
References

- Introduction to Parallel Programming and MapReduce, Google Code University

- Distributed Systems

- MapReduce: Simplified Data Processing on Large Clusters

- Hadoop
  - http://hadoop.apache.org/core/