The Map-Reduce Framework

Compiled by Mark Silberstein, using slides from Dan Weld’s class at U. Washington, Yaniv Carmeli and some other
The MapReduce Paradigm

You are given a phonebook:

- How many entries have the last name “Smith”?
- Create a list of all last names, with the numbers of entries they appear in.
- How many entries have the first name “John”?
- Create a list of all first names, with the numbers of entries they appear in.

How can these tasks be distributed to many computers?
Motivation

◆ Large-Scale Data Processing
  ♦ Want to use 1000s of CPUs
    ♦ But don’t want hassle of *managing* things

◆ MapReduce framework provides
  ♦ Automatic parallelization & distribution
  ♦ Fault tolerance
  ♦ I/O scheduling
  ♦ Monitoring & status updates
Map/Reduce

- Map/Reduce
  - Programming model from Lisp
  - (and other functional languages)

- Many problems can be phrased this way
- Easy to distribute across nodes
- Nice retry/failure semantics
Map in Lisp (Scheme)

- (map \( f \) list \([\text{list}_2 \ \text{list}_3 \ldots]\))

- (map square `(1 2 3 4))
  - `(1 4 9 16)

- (reduce + `(1 4 9 16))
  - `(+ 16 (+ 9 (+ 4 1)))
  - 30

- (reduce + (map square (map − l₁ l₂))))
Map/Reduce ala Google

- **map(key, val)** is run on each item in set
  - emits new-key / new-val pairs

- **reduce(new-key, new-vals)** is run for each unique new-key emitted by map()
  - emits final output
count words in docs

- Input consists of (url, contents) pairs

  - map(key=url, val=contents):
    - For each word w in contents, emit (w, "1")

  - reduce(key=word, values=uniq_counts):
    - Sum all "1"s in values list
    - Emit result "(word, sum)"
Count, Illustrated

map(key=url, val=contents):
For each word $w$ in contents, emit ($w$, “1”)

reduce(key=word, values=uniq_counts):
Sum all “1”s in values list
Emit result “(word, sum)”

see bob throw
see spot run

see 1
bob 1
run 1
see 1
spot 1
throw 1

bob 1
run 1
see 2
spot 1
throw 1
Grep

- Input consists of (file, regexp)
- map(file, regexp):
  - If contents matches regexp, emit (regexp, #line)
- reduce(regexp, #line):
  - Don’t do anything; just emit line
Reverse Web-Link Graph

◆ Map
  - For each URL linking to target, ...
  - Output <target, source> pairs

◆ Reduce
  - Concatenate list of all source URLs
  - Outputs: <target, list(source)> pairs
Inverted Index

❖ Result: `<word : pages where it appears>`
❖ Map
  ✤ For each page,
  ✤ Output `<word: page>`
❖ Reduce
  ✤ Concatenate all pages of the word
  ✤ Output `<word: page_1, page_2,...>`
Map-Reduce Framework

Input: a collection of keys and their values

User-written Map function

Each input (k,v) mapped to set of *intermediate* key-value pairs

User-written Reduce function

sort all key-value pairs by key

One list of intermediate values for each key: (k, [v₁,…,vₙ])

Shuffle
Parallelization

- How is this distributed?
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. Shuffle: After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. Now partition space of output map keys, and run reduce() in parallel

- If map() or reduce() fails, reexecute!
Data store 1

map

(key 1, 
values...)

(key 2, 
values...)

(key 3, 
values...)

map

(key 1, ... 
values
key 3, 
intermediate 
values
final key 1 
values
final key 2 
values
final key 3 
values
...

Shuffle

== Barrier == : Aggregates intermediate values by output key

reduce

final key 1 
values

reduce

final key 2 
values

reduce

final key 3 
values

Data store n
Semantics Assumptions on Map and Reduce operations

- Map has no side effects
- Reduce is
  - Associative \((a \# b) \# c = a \# (b \# c)\)
  - Commutative \((a \# b) = (b \# a)\)
  - Idempotent
Another example

In a credit card company:

Given:

- “credit card number” -> “bargain cost”

Find the average bargain cost per month per card

Map(card, {bargain, date})

Reduce({{card, month}, {total cost, #bargains}})
Implementation

- A program forks a *master* process and many *worker* processes.
- Input is partitioned into some number of *splits*.
- Worker processes are assigned either to perform Map on a split or Reduce for some set of intermediate keys.
Distributed Execution Overview

Problems with efficiency of parallel implementation???
Distributed Execution 2
Another example: distributed sort

- Map ?
- Reduce ?
- Something is missing ....
- Shuffle phase must allow distributed sort!
Responsibility of the Master

1. Assign Map and Reduce tasks to Workers.
2. Check that no Worker has died (because its processor failed).
3. Communicate results of Map to the Reduce tasks.
Communication from Map to Reduce

- Select a number $R$ of reduce tasks.
- Divide the intermediate keys into $R$ groups, e.g. by hashing.
- Each Map task creates, at its own processor, $R$ files of intermediate key-value pairs, one for each Reduce task.
Locality

- Master program divvies 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 64-128 MB blocks: same size as Google File System chunks
Fault Tolerance

- Master detects worker failures
  - Re-executes failed 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
Strugglers

- No reduce can start until map is complete:
  - A single slow disk controller can rate-limit the whole process
- Master redundantly executes “slow” map tasks; uses results of first correctly-terminating replica
Task Granularity & Pipelining

◆ Fine granularity tasks: \#map tasks >> \#machines
  ♦ Minimizes time for fault recovery
  ♦ Can pipeline shuffling with map execution
  ♦ Better dynamic load balancing

◆ Often use 200,000 map & 5000 reduce tasks

◆ Running on 2000 machines
Other Refinements

- Sorting guarantees
  - within each reduce partition
- Compression of intermediate data
- Combiner
  - Perform reduce on local map results before shuffle
  - Useful for saving network bandwidth
Yet another example

- Distributed single source shortest path
  - adjacency list representation (node -> list of neighbors)
  - Until all nodes are done:
    - map(node, distance from source, {neighbors})
    - reduce(neighbor, {distance,distance,....})
      - Output: {neighbor, minimal distance}

“Graph Algorithms with MapReduce”:
http://www.umiacs.umd.edu/~jimmylin/cloud-computing/Session5.ppt
Hadoop

◆ An open-source implementation of Map-Reduce in Java.

◆ Download from:
  http://apache.org/hadoop

◆ Main reference