MapReduce

Roy Campbell

2/8/2013
Lecture Outline

• Motivation: Why MapReduce model
• Programming Model
• Examples
  – Word Count
  – Pi Estimation
  – Image Smoothing
  – PageRank
• MapReduce execution
• Misc. Remarks
• Next Gen Hadoop
Motivation, What is MapReduce
Challenges with Traditional Programming Models (MPI)

- MPI gives you MPI_Send, MPI_Recieve
- Deadlock is possible...
  - Blocking communication can cause deadlock
    - "crossed" calls when trading information
    - example:
      - Proc1: MPI_Receive(Proc2, A); MPI_Send(Proc2, B);
      - Proc2: MPI_Receive(Proc1, B); MPI_Send(Proc1, A);
    - There are some solutions - MPI_SendRecv()
- Large overhead from comm. mismanagement
  - Time spent blocking is wasted cycles
  - Can overlap computation with non-blocking comm.
- Load imbalance is possible! Dead machines?
- Things are starting to look hard to code!
Commodity Clusters

• Web data sets can be very large
  – Tens to hundreds of terabytes
• Cannot mine on a single server (why?)
• Standard architecture emerging:
  – Cluster of commodity Linux nodes
  – Gigabit ethernet interconnect
• How to organize computations on this architecture?
  – Mask issues such as **hardware failure**
Solution

• Use distributed Storage
  - 6-24 disks attached to a blade
  - 32-64 blades in a rack connected by Ethernet
• Push computations down to storage
  - Computations process contents of disks
  - Data on disks read sequentially from beginning to end
  - Rate limited by speed of disks (speed can get at data)
Cluster Architecture

Each rack contains 16-64 nodes

2-10 Gbps backbone between racks

1 Gbps between any pair of nodes in a rack
Motivation: Large Scale Data Processing

• Many tasks composed of processing lots of data to produce lots of other data
• Large-Scale Data Processing
  – Want to use 1000s of CPUs
    • But don’t want hassle of *managing* things
• MapReduce provides
  – User-defined functions
  – Automatic parallelization and distribution
  – Fault-tolerance
  – I/O scheduling
  – Status and monitoring
Stable storage

- First order problem: if nodes can fail, how can we store data persistently?
- Answer: Distributed File System
  - Provides global file namespace
  - Google GFS; Hadoop HDFS; Kosmix KFS
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common
Distributed File System

- **Chunk Servers**
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks

- **Master node**
  - a.k.a. Name Nodes in HDFS
  - Stores metadata
  - Might be replicated

- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data
MapReduce Programming Model
What is Map/Reduce

• Map/Reduce
  – Programming model from LISP
  – (and other functional languages)
• Many problems can be phrased this way
• Easy to distribute across nodes
  – Imagine 10,000 machines ready to help you compute anything you could cast as a MapReduce problem!
    • This is the abstraction Google is famous for authoring
  – It hides LOTS of difficulty of writing parallel code!
  – The system takes care of load balancing, dead machines, etc.
• Nice retry/failure semantics
Programming Concept

• Map
  – Perform a function on **individual values** in a data set to create a **new list** of values
  – Example: \( \text{square } x = x \times x \)
    map square \([1,2,3,4,5]\)
    returns \([1,4,9,16,25]\)

• Reduce
  – Combine values in a data set to create a new value
  – Example: \( \text{sum } = (\text{each elem in arr, total } +=) \)
    reduce \([1,2,3,4,5]\)
    returns 15 (the sum of the elements)
MapReduce Programming Model

Input & Output: each a set of key/value pairs
Programmer specifies two functions:

**map** (in_key, in_value) →
  list(out_key, intermediate_value)
  – Processes input key/value pair
  – Produces set of intermediate pairs

**reduce** (out_key, list(intermediate_value)) →
  list(out_value)
  – Combines all intermediate values for a particular key
  – Produces a set of merged output values (usually just one)
Examples
Warm up: Word Count

• We have a large file of words, Many words in each line

• Count the number of times each distinct word appears in the file(s)
Word Count using MapReduce

map(key = line, value=contents):
    for each word w in value:
        emit Intermediate(w, 1)

reduce(key, values):
    // key: a word; values: an iterator over counts
    result = 0
    for each v in intermediate values:
        result += v
    emit(key,result)
Word Count, Illustrated

map(key=line, 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 run
see spot throw

see 1
bob 1
run 1
see 1
spot 1
throw 1
bob 1
run 1
see 2
spot 1
throw 1
Exercise 1: Pi Estimation

• Using Monte Carlo Simulation, Estimate the value of Pi
• Throw darts
• Compute the ratio of the darts landed within the square vs the darts landed within the circle.
• Evaluating whether a particular dart landed within the circle is easy

\[ \pi = 4 \times \frac{\text{Circle Area}}{\text{Square Area}} \]
Exercise 1: Pi Estimation

• Mapper: Generate points in a unit square and then count points inside/outside of the inscribed circle of the square.

• Reducer: Accumulate points inside/outside results from the mappers.

• After the MapReduce Job, estimate Pi
  – The fraction numInside/numTotal is an approximation of the value (Area of the circle)/(Area of the square)
  – Then, Pi is estimated value to be 4(numInside/numTotal)

• bin/hadoop jar hadoop-0.18.0-examples.jar pi 10 1000000
Exercise 2: Image Smoothing

- To smooth an image, use a sliding mask and replace the value of each pixel
Exercise 2: Image Smoothing

• Map: Input key = x,y input value = R, G, B
  – Emit 9 points
    • (x-1, y-1, R,G,B)
    • (x, y-1, R,G,B)
    • (x+1, y-1, R,G,B)
    • Etc.

• Reduce: input key = x,y input value: list of R,G,B
  – Compute average R,G,B
  – Emit key=x,y value = average RGB
Exercise 4: PageRank
PageRank

- Program implemented by Google to rank any type of recursive “documents” using MapReduce.
- Initially developed at Stanford University by Google founders, Larry Page and Sergey Brin, in 1995.
- Still provides the basis for all of Google's web search tools.
- PageRank value for a page $u$ is dependent on the PageRank values for each page $v$ out of the set $B_u$ (all pages linking to page $u$), divided by the number $L(v)$ of links from page $v$

$$\text{PR}(u) = \sum_{v \in B_u} \frac{\text{PR}(v)}{L(v)}$$
Exercise 4: PageRank

• Phase 1: Propagation
• Phase 2: Aggregation

• Input: A pool of objects, including both vertices and edges
PageRank: Propagation

- **Map:** for each object
  - If object is vertex, emit key=URL, value=object
  - If object is edge, emit key=source URL, value=object
- **Reduce:** (input is a web page and all the outgoing links)
  - Find the number of edge objects \( \rightarrow \) outgoing links
  - Read the PageRank Value from the vertex object
  - Assign \( PR(edges) = PR(vertex) / \text{num\_outgoing} \)
PageRank: Aggregation

- **Map:** for each object
  - If object is vertex, emit key=URL, value=object
  - If object is edge, emit key=Destination URL, value=object
- **Reduce:** (input is a web page and all the incoming links)
  - Add the PR value of all incoming links
  - Assign \[ PR(\text{vertex}) = \sum PR(\text{incoming links}) \]
(Hadoop) MapReduce Execution
Execution

- How is this distributed?
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. 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!
Execution

[Diagram showing data flow from input through intermediate stages to output]

- Input
- Intermediate: k1:v, k1:v, k2:v, k1:v, k3:v, k4:v, k4:v, k5:v, k4:v, k1:v, k3:v
- Group by Key
- Grouped: k1:v, k2:v, k3:v, k4:v, k5:v
- Output
Execution Initialization

- Split input file into 64MB sections (GFS)
  - Read in parallel by multiple machines
- Fork off program onto multiple machines
- One machine is Master
- Master assigns idle machines to either Map or Reduce tasks
- Master Coordinates data communication between map and reduce machines
Partition Function

• Inputs to map tasks are created by contiguous splits of input file
• For reduce, we need to ensure that records with the same intermediate key end up at the same worker
• System uses a default partition function e.g., hash(key) mod R
• Sometimes useful to override
  – E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file
Map-Machine

• Reads contents of assigned portion of input-file
• Parses and prepares data for input to map function (e.g. read <a /> from HTML)
  – Classes implementing InputFormat
• Passes data into map function and saves result in memory (e.g. <target, source>)
• Periodically writes completed work to local disk
• Notifies Master of this partially completed work (intermediate data)
Reduce-Machine

• Receives notification from Master of partially completed work
• Retrieves intermediate data from Map-Machine via remote-read
• Sorts intermediate data by key (e.g. by target page)
• Iterates over intermediate data
  — For each unique key, sends corresponding set through reduce function
• Appends result of reduce function to final output file (GFS)
Data flow

• Input, final output are stored on a distributed file system
  – Scheduler tries to schedule map tasks “close” to physical storage location of input data
• Intermediate results are stored on local FS of map and reduce workers
• Output is often input to another map reduce task
How many Map and Reduce jobs?

- M map tasks, R reduce tasks
- Rule of thumb:
  - Make M and R much larger than the number of nodes in cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds recovery from worker failure
- Usually R is smaller than M, because output is spread across R files
Combiners

• Often a map task will produce many pairs of the form (k,v1), (k,v2), ... for the same key k
  – E.g., popular words in Word Count
• Can save network time by pre-aggregating at mapper
  – combine(k1, list(v1)) \to v2
  – Usually same as reduce function
• Works only if reduce function is commutative and associative
1. Client submits “wordcount” job, indicating code and input files
2. JobTracker breaks input file into $k$ chunks, (in this case 6). Assigns work to T.trackers.
3. After map(), T.trackers exchange map-output to build reduce() keyspace
4. JobTracker breaks reduce() keyspace into $m$ chunks (in this case 6). Assigns work.
5. reduce() output may go to HDFS
Misc. Remarks
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 in HDFS (by default): 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!
MapReduce Advantages/Disadvantages

• Now it’s easy to program for many CPUs
  – Communication management effectively gone
    • I/O scheduling done for us
  – Fault tolerance, monitoring
    • machine failures, suddenly-slow machines, etc are handled
  – Can be much easier to design and program!
  – Can cascade several (many?) MapReduce tasks

• But … it further restricts solvable problems
  – Might be hard to express problem in MapReduce
  – Data parallelism is key
    • Need to be able to break up a problem by data chunks
MapReduce Conclusions

• MapReduce has proven to be a useful abstraction
• Greatly simplifies large-scale computations at Google
• Functional programming paradigm can be applied to large-scale applications
• Fun to use: focus on problem, let the middleware deal with messy details
Hadoop NextGen
NextGen Hadoop

- Split up the two major functions of JobTracker
  - Cluster resource management
  - Application life-cycle management
- MapReduce becomes **user-land** library
- Resource Manager
  - Global resource scheduler
  - Hierarchical queues
- Node Manager
  - Per-machine agent
  - Manages the life-cycle of container
  - Container resource monitoring
- Application Master
  - Per-application
  - Manages application scheduling and task execution
  - E.g. MapReduce Application Master
Architecture

MapReduce Status
Job Submission
Node Status
Resource Request

Client

Resource Manager

Node Manager

App Mstr

Container

Node Manager

App Mstr

Container

Node Manager

Container

Container

1/31/12
Improvements vis-à-vis current MapReduce

• Scalability
• Fault Tolerance and Availability
• Multiple versions of MapReduce can run in the same cluster
• Remove fixed partition of map and reduce slots
• Support for programming paradigms other than MapReduce
  – MPI
  – Master-Worker
  – Machine Learning
  – Iterative processing
  – Enabled by allowing use of paradigm-specific Application Master
  – Run all on the same Hadoop cluster