Map Reduce Architecture

Adapted from Lectures by
Anand Rajaraman (Stanford Univ.)
and Dan Weld (Univ. of Washington)
and T.K. Prasad (Wright State)
Motivation for MapReduce (why)

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

- MapReduce Architecture provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates
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
- Nice retry/failure semantics
Map in LISP (Scheme)

- `(map f list [list2 list3 ...])`

- `(map square '(1 2 3 4))
  (1 4 9 16)"
Reduce in LISP (Scheme)

- `(reduce f id list)`

- `(reduce + 0 '(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 (+ 1 0)) ))
  - 30

- `(reduce + 0
      (map square (map – l₁ l₂))))`
Warm up: Word Count

- We have a large file of words, one word to a line
- Count the number of times each distinct word appears in the file

Sample application: analyze web server logs to find popular URLs
Case 1: Entire file fits in memory

Case 2: File too large for mem, but all <word, count> pairs fit in mem

Case 3: File on disk, too many distinct words to fit in memory

```
    sort datafile | uniq -c
```
To make it slightly harder, suppose we have a large corpus of documents.

Count the number of times each distinct word occurs in the corpus:

```
words(docs/*) | sort | uniq -c
```

where `words` takes a file and outputs the words in it, one to a line.

The above captures the essence of MapReduce.

- Great thing is it is naturally parallelizable.
MapReduce

- Input: a set of key/value pairs
- User supplies two functions:
  - map(k,v) → list(k1,v1)
  - reduce(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs
Word Count using MapReduce

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; values: an iterator over counts
result = 0
for each count v in values:
    result += v
emit(key, result)
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 run
see spot throw

see 1
bob 1
run 1
see 2
spot 1
throw 1
Model is Widely Applicable
MapReduce Programs In Google Source Tree

Example uses:
- distributed grep
- term-vector / host
- document clustering
- distributed sort
- web access log stats
- machine learning
- web link-graph reversal
- inverted index construction
- statistical machine translation

...
Implementation Overview

- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data (SOSP'03)
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs
Distributed Execution Overview

User Program

Master

fork

fork

fork

assign map

assign reduce

Input Data

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Worker

Split 0

Split 1

Split 2

read

local write

remote read, sort

write

Output File 0

Output File 1
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
Coordination

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers

- Master pings workers periodically to detect failures
Failures

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker

- Reduce worker failure
  - Only in-progress tasks are reset to idle

- Master failure
  - MapReduce task is aborted and client is notified
Execution

Input

Intermediate

Group by Key

Grouped

Output
Parallel Execution
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)) \( \rightarrow \) v2
  - Usually same as reduce function
- Works only if reduce function is commutative and associative
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
Execution Summary

- 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!
Slow Servers

- Slow tasks are called Stragglers
- The slowest task slows the entire job down (why?)
- Due to Bad Disk, Network Bandwidth, CPU, or Memory
- Keep track of “progress” of each task (% done)
- Perform proactive backup (replicated) execution of some straggler tasks
  - A task considered done when its first replica complete (other replicas can then be killed)
  - Approach called Speculative Execution.

Barrier at the end of Map phase!
Exercise 1: Host size

- Suppose we have a large web corpus
- Let’s look at the metadata file
  - Lines of the form (URL, size, date, ...)
- For each host, find the total number of bytes
  - i.e., the sum of the page sizes for all URLs from that host

map (URL, &size):
emit (key=URL.host, value=size)
reduce (host, [sizes]):
emit (host, sum(sizes))
Exercise 2: Distributed Grep

- Find all occurrences of the given pattern in a very large set of files
Grep

- Input consists of (url+offset, single line)
- map(key=url+offset, val=line):
  - If contents matches regexp, emit (line, “1”)
- reduce(key=line, values=uniq_counts):
  - Don’t do anything; just emit line
Exercise 3: Graph reversal

- Given a directed graph as an adjacency list:
  - src1: dest11, dest12, ...
  - src2: dest21, dest22, ...

- Construct the graph in which all the links are reversed
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
Chains

- map -> reduce -> map -> reduce
- E.g., output the most common words by frequency
  
  `sort datafile | uniq -c | sort -n`

- Same for mapreduce
  - Map1: output (“word”, 1)
  - Reduce1: output (“word”, count)
  - Map2: output ((count, “word”), nothing)
  - Reduce2: identity
Hadoop

- An open-source implementation of Map Reduce in Java
  - Uses HDFS for stable storage
- Download from: 
public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one =
        new IntWritable(1);
    private Text word = new Text();

    public void map( LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
public static class ReduceClass extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(
        Text key,
        Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter)
            throws IOException {
            // key is word, values is a list of 1's
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
    }
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
Hadoop Code - Driver

// Tells Hadoop how to run your Map-Reduce job
public void run (String inputPath, String outputPath) throws Exception {
    // The job. WordCount contains MapClass and Reduce.
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("mywordcount");
    // The keys are words (strings) conf.setOutputKeyClass(Text.class);
    // The values are counts (ints)
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(MapClass.class);
    conf.setReducerClass(ReduceClass.class);
    FileInputFormat.addInputPath(conf, newPath(inputPath));
    FileOutputFormat.setOutputPath(conf, new Path(outputPath));
    JobClient.runJob(conf);
} // Source: http://developer.yahoo.com/hadoop/tutorial/module4.html#wordcount
Reading

- Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*
  
  http://labs.google.com/papers/mapreduce.html

  
  http://labs.google.com/papers/gfs.html
Conclusions

- MapReduce proven to be useful abstraction
- Greatly simplifies large-scale computations
- Fun to use: 
  - focus on problem,
  - let library deal w/ messy details
GFS – The Google File System
Overview

• Distributed filesystem designed at Google for storing large amounts of data (TB in 2003, PB now)
• Storage backend for MapReduce inputs / outputs
• Inspired HDFS, which is the storage backend for Hadoop
Features

• Large files
  • GBs in each file
  • TBs/PBs total storage

• Random writes are rare
  • Appends, seq reads, random reads

• Failures are common
  • Replication necessary

• Throughput > Latency
Organization and Metadata

- Files are organized into directories
  - `/a/b/c/file`
- Each file mapped into chunks
  - Logical block (does not encode location)
  - Large (64MB)
- Chunks are mapped to replicas that store them
  - `chunk1` → `rep1`
  - `chunk2` → `rep2`

All metadata stored on master server (w/ shadow backup)

Master server issues leases on data to replicas, clients

- Allows caching of information for a period of time w/o contacting server
- Note: only metadata are cached!
Operation Overview

1. Contact master with filename, offset
2. Receive chunk handle (ID), locations, lease
3. Contact chunk server storing each chunk, read/write directly
4. Renew lease after expiry
Write operation

1. Contact master with filename, chunk offset
2. Receive list of replicas
3. Set up TCP pipeline through replicas by shortest total distance
4. Contact primary to serialize write
5. Primary contacts secondaries
6. Secondaries reply
7. Primary replies success/failure to client
Weak consistency

- Metadata updates atomic, use recovery log
- Successful writes create consistent state
- Unsuccessful writes can result in stale reads (like Cassandra)
- Concurrent writes can be interleaved but produce consistent reads
- Atomic record append with at least once semantics
Snapshot mechanism

• Fast snapshot of entire filesystem or a subdirectory
1. Revoke all leases
2. Save a copy of all relevant metadata
   • In memory at master, so fast
3. Mark all chunks for copy-on-write
   • Copy chunk to new logical chunk, send write there
Master Failure

- Shadow master uses *log replication* to keep track of most metadata
  - Directory -> file mapping
  - File -> chunk list mapping
- Chunk servers have **final say** on chunk -> server mapping
  - No point in having master tell server it stores a chunk it doesn’t have!
- Shadows satisfy reads if master unavailable, take over if master fails