MapReduce: Simplified Data Processing on Large Clusters

CSE 454

Motivation

- Large-Scale Data Processing
  - Want to use 1000s of CPUs
  - But don’t want hassle of managing things
- MapReduce 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 list2 list3 ...
- (map square '(1 2 3 4))
  - (1 4 9 16)
- (reduce + '1 4 9 16))
  - 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
- \( reduce(key, vals) \) is run for each unique 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)"

Slides based on those by Jeff Dean, Sanjay Ghemawat, Google, Inc.
Count, Illustrated

**Map**
- For each word w in contents, emit (w, "1")

**Reduce**
- Sum all "1"s in values list
- Emit result "(word, sum)"

<table>
<thead>
<tr>
<th>see bob throw</th>
<th>see spot run</th>
</tr>
</thead>
<tbody>
<tr>
<td>see 1</td>
<td>bob 1</td>
</tr>
<tr>
<td>bob 1</td>
<td>run 1</td>
</tr>
<tr>
<td>run 1</td>
<td>see 2</td>
</tr>
<tr>
<td>see 1</td>
<td>spot 1</td>
</tr>
<tr>
<td>spot 1</td>
<td>throw 1</td>
</tr>
<tr>
<td>throw 1</td>
<td></td>
</tr>
</tbody>
</table>

Grep

- Input consists of (url+offset, single line)
- **Map** (url+offset, val=line):
  - If contents matches regexp, emit (line, "1")
- **Reduce** (line, values=uniq_counts):
  - 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

- **Map**
- **Reduce**

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
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!

Job Processing

1. Client submits “grep” job, indicating code and input files.
2. JobTracker breaks input file into 6 chunks, assigns work to trackers.
3. After map(), tasktrackers exchange map-output to build reduce() keyspace.
4. JobTracker breaks reduce() keyspace into 6 chunks (in this case 6). Assigns work.
5. reduce() output may go to NDFS.

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.

Parallel Execution

JobTracker

TaskTracker 0

TaskTracker 1

TaskTracker 2

TaskTracker 3

TaskTracker 4

TaskTracker 5

Input

Intermediate

Group by Keys

Grouped

Output

Map Task 1

Map Task 2

Map Task 3

Reduce Task 1

Reduce Task 2

Reduce Task 3

MapReduce status: MR_Induce-beta-kung-2003_10_28_00_03

Process | Time | User/Program | MapReduce | Map 1 | Map 2 | Reduce 1 | Reduce 2 | Reduce 3
------- | ---- | ----------- | ---------- | ----- | ----- | -------- | -------- | --------
    0    | 0.00 | 0          | Hadoop     |       |       |          |          |          
    1    | 0.01 | 0          | Hadoop     |       |       |          |          |          
    2    | 0.02 | 0          | Hadoop     |       |       |          |          |          
    3    | 0.03 | 0          | Hadoop     |       |       |          |          |          
    4    | 0.04 | 0          | Hadoop     |       |       |          |          |          
    5    | 0.05 | 0          | Hadoop     |       |       |          |          |          
    6    | 0.06 | 0          | Hadoop     |       |       |          |          |          
    7    | 0.07 | 0          | Hadoop     |       |       |          |          |          
    8    | 0.08 | 0          | Hadoop     |       |       |          |          |          
    9    | 0.09 | 0          | Hadoop     |       |       |          |          |          

- "grep"
Fault Tolerance / Workers

- Handled via re-execution
  - Detect failure via periodic heartbeats
  - Re-execute completed + in-progress map tasks  
    - Why???
  - Re-execute in progress reduce tasks
  - Task completion committed through master

Robust: lost 1600/1800 machines once -> finished ok
Semantics in presence of failures: see paper

Master Failure

- Could handle, ... ?
- But don't yet
  - (master failure unlikely)
Refinement: Redundant Execution

Slow workers significantly delay completion time
- Other jobs consuming resources on machine
- Bad disks w/ soft errors transfer data slowly
- Weird things: processor caches disabled (!)

Solution: Near end of phase, spawn backup tasks
- Whichever one finishes first "wins"
Dramatically shortens job completion time

Refinement: Locality Optimization

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack

- Effect
  - Thousands of machines read input at local disk speed
    - Without this, rack switches limit read rate

Refinement: Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix
    - Not always possible - third-party source libraries
  - On segmentation fault:
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - If master sees two failures for same record:
    - Next worker is told to skip the record

Other Refinements

- Sorting guarantees
  - within each reduce partition
- Compression of intermediate data
- Combiner
  - Useful for saving network bandwidth
- Local execution for debugging/testing
- User-defined counters

Performance

Tests run on cluster of 1800 machines:
- 4 GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:
- **MR_GrepScan** 1010 100-byte records to extract records matching a rare pattern (92K matching records)
- **MR_SortSort** 1010 100-byte records (modeled after TeraSort benchmark)

MR_Grep

Locality optimization helps:
- 1800 machines read 1 TB at peak ~31 GB/s
- W/out this, rack switches would limit to 10 GB/s

Startup overhead is significant for short jobs
**Backup tasks reduce job completion time a lot!**
**System deals well with failures**

### Experience

Rewrote Google’s production indexing System using MapReduce

- Set of 10, 14, 17, 21, 24 MapReduce operations
- New code is simpler, easier to understand
  - 3800 lines C++ → 700
- MapReduce handles failures, slow machines
- Easy to make indexing faster
  - add more machines

### Usage in Aug 2004

- Number of jobs: 29,423
- Average job completion time: 634 secs
- Machine days used: 79,186 days
- Input data read: 3,288 TB
- Intermediate data produced: 758 TB
- Output data written: 193 TB
- Average worker machines per job: 157
- Average worker deaths per job: 1.2
- Average map tasks per job: 3,351
- Average reduce tasks per job: 55
- Unique map implementations: 395
- Unique reduce implementations: 269
- Unique map/reduce combinations: 426

### Related Work

- Programming model inspired by functional language primitives
- Partitioning/shuffling similar to many large-scale sorting systems
  - NOW-Sort ['97]
- Re-execution for fault tolerance
  - BAD-FS ['04] and TACC ['97]
- Locality optimization has parallels with Active Disks/Diamond work
  - Active Disks ['01], Diamond ['04]
- Backup tasks similar to Eager Scheduling in Charlotte system
  - Charlotte ['96]
- Dynamic load balancing solves similar problem as River’s distributed queues
  - River ['99]

### Conclusions

- MapReduce proven to be useful abstraction
- Greatly simplifies large-scale computations
- Fun to use:
  - focus on problem,
  - let library deal w/ messy details