Parallelism in the Cloud: 

*MapReduce*

Kyle Jamieson
UCL Computer Science

COMPM038/COMPGZ06
1st February 2013
Motivation

• Many web data sets very large (10s–100s of terabytes)
  – Cannot mine on a single server
  – Too much data, computation; supercomputer: expensive

• Standard architecture emerging
  – Cluster of commodity machines
  – Gigabit Ethernet interconnect

• How to organize computation on this architecture?
  – Mask issues such as hardware failure
Cluster architecture

1 Gbps between any pair of nodes in a rack

2-10 Gbps backbone between racks

Each rack contains 16 to 64 nodes
Stable storage

• **First order problem:** *if nodes can fail, how can we store data persistently?*

• **Answer:** A distributed file system
  – Provides a global file namespace abstraction
  – Examples: Google GFS; Hadoop File System (HDFS)

• **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
  – GFS: each chunk is 64 MB
  – Each chunk replicated, usually 3× (GFS)
  – Try to keep replicas in different racks (why?)

• Master node: 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

• Why? Large-scale data processing
  – Recall, want to use $10^3$–$10^4$ CPUs across $10^2$–$10^3$ machines
    • But don’t want to bear the burden of managing computation

• MapReduce architecture provides
  – Automatic parallelization and distribution
  – Fault tolerance; I/O scheduling; monitoring and status updates

• What are 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 and reduce in LISP (Scheme)

- (map \textit{f list})
  
  \[
  (\text{map square} \ (1 \ 2 \ 3))
  \]

  \[
  (1 \ 4 \ 9)
  \]

- (reduce \textit{f initval list})
  
  \[
  (\text{reduce} + \ 0 \ (1 \ 4 \ 9))
  \]

  \[
  (+ \ 9 \ (+ \ 4 \ (+ \ 1 \ 0)))
  \]

  \[
  14
  \]
Warm up: Word count

• We have many large files of words
  – Problem: Count the number of times each distinct word appears in the set of files
  – Sample application: analyze web server logs to find popular URLs

• Case 1: All files fit in memory
• Case 2: Files too large for memory, but all (word, count) pairs fit in memory
• Case 3: Files on disk, too many distinct words to fit just the unique words in memory
Many of the above computations can occur in parallel.
MapReduce

• Input: a set of key/value pairs

• User supplies two functions:
  – map(k1, v1) → list(k2, v2)
  – reduce(k2, list(v2)) → list(v2)

• (k2, v2) is an intermediate key/value pair

• Output is the set of (k2, v2) pairs
Word count using MapReduce

\[ map(key, \text{value}): \]
\hspace{1cm} // key: document name
\hspace{1cm} // value: text of document
\hspace{1cm} foreach word \textit{w} in value:
\hspace{2cm} EmitIntermediate(\textit{w}, 1)

\[ reduce(key, \text{values}): \]
\hspace{1cm} // key: a word
\hspace{1cm} // values: an list of counts
\hspace{1cm} result = 0
\hspace{1cm} foreach count \textit{v} in values:
\hspace{2cm} result += \textit{v}
\hspace{1cm} Emit(key, result) \]
Implementation overview

• Typical cluster: 100s of 2-CPU machines, 4–8 GB RAM
  – Limited bisection bandwidth: divide network into any two parts; take minimum bandwidth between parts
  – GFS manages input and output data is managed on local IDE disks

• MapReduce Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

• Implementation is a C++ library linked into user programs
MapReduce execution overview

- **Input files (in GFS)**
- **Map phase**
- **Intermediate files (on local disks)**
- **Reduce phase** ($R=2$ workers)
- **Output files** ($R$ files in GFS)
How many map and reduce jobs?

- $M$ map jobs, $R$ reduce jobs

- Rule of thumb: $M, R \gg$ cluster size
  - Improves dynamic load balancing, speeds failure recovery
  - Master needs to make $O(M + R)$ scheduling decisions (one for each map or reduce task)
  - Master needs to keep $O(M \times R)$ state in memory (names of $R$ intermediate files for each of $M$ map tasks)
  - One GFS chunk per map job is common

- Usually $R < M$, because output is spread across $R$ files
Parallel execution in MapReduce

Handles failures automatically, e.g., restarts tasks if a node fails; runs multiples copies of the same task to avoid a slow task slowing down the whole job.
Coordination

• Data structures maintained at the master
  – For each map task and reduce task:
    1. Task status: (idle, in-progress, completed), starts idle
    2. Identity of machine
  – Idle tasks scheduled as workers become available

• Master pings workers periodically in order to detect failures
Fault tolerance

• **Map worker failure**
  – Map tasks *in-progress/completed* at worker reset to *idle*
  – Reduce workers notified when task 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
  – Rare occurrence because there is only one master
Semantics in the presence of failures

• **Claim**: Distributed MapReduce implementation produces **same output** as a **non-faulting local execution** (with deterministic operators)

• **Atomic commits** of map/reduce tasks: Either it all happens **exactly once, or no effects** are observable.
  – **Map tasks**: worker sends **completion message** to master with names of $R$ temporary files **after writes finish**; master accepts only one.
  – **Reduce task**: **Atomic rename** of temporary output file to final output file **after writes finish**; file system guarantees just one execution.
Combiners

• Often a map task will produce many pairs of the form \((k, v_1), (k, v_2), \ldots\) for the same key \(k\)
  – e.g., popular words in word count example

• **Can save network time by “pre-aggregating” at the map worker**
  – \(\text{combine}(k_1, \text{list}(v_1)) \rightarrow v_2\)
  – Usually same as reduce function

• Works only if reduce function is commutative and associative
Partitioning 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., $\text{hash(key)} \mod R$

- Sometimes useful to override:
  - e.g., suppose we are counting URL access frequencies
    - Partitioning function $\text{hash(hostname(URL))} \mod R$ ensures all URLs from a given host end up in the same output file
Performance: Stragglers

- Machine that takes unusually long time to complete one of the last few map or reduce tasks in a computation

- Why does this happen? Bad disk: errors slow down. CPU load high. Configuration bug disables CPU cache.

- Mitigation strategy: When close to completion, master schedules backup executions of remaining in-progress tasks
  - Task completes when either primary or backup completes.
  - Atomicity properties above guarantee correctness.
Data transfer rates for sort program

(a) Normal execution
Effect of disabling backup tasks

(a) Normal execution

(b) No backup tasks
Effect of machine failures

(a) Normal execution  (b) No backup tasks  (c) 200 tasks killed
2003–2004: MapReduce catches on

- Many uses: machine learning, Google News, Google Product Search, extraction of data for reports, extraction of geographical locations from large corpus of web pages for localized search

- Processed three Petabytes of data in August 2004!

- Rewrote the indexing system that produces data structures for Google web search
  - Processes documents retrieved by crawler (>20 TB), stored in GFS
  - Simpler, smaller indexing code

### MapReduce jobs in August 2004

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of jobs</td>
<td>29,423</td>
</tr>
<tr>
<td>Average job completion time</td>
<td>634 secs</td>
</tr>
<tr>
<td>Machine days used</td>
<td>79,186 days</td>
</tr>
<tr>
<td>Input data read</td>
<td>3,288 TB</td>
</tr>
<tr>
<td>Intermediate data produced</td>
<td>758 TB</td>
</tr>
<tr>
<td>Output data written</td>
<td>193 TB</td>
</tr>
<tr>
<td>Average worker machines per job</td>
<td>157</td>
</tr>
<tr>
<td>Average worker deaths per job</td>
<td>1.2</td>
</tr>
<tr>
<td>Average map tasks per job</td>
<td>3,351</td>
</tr>
<tr>
<td>Average reduce tasks per job</td>
<td>55</td>
</tr>
<tr>
<td>Unique map implementations</td>
<td>395</td>
</tr>
<tr>
<td>Unique reduce implementations</td>
<td>269</td>
</tr>
<tr>
<td>Unique map/reduce combinations</td>
<td>426</td>
</tr>
</tbody>
</table>
PageRank: Random walks on the web

• If a user starts at a random web page and surfs by clicking links and randomly entering new URLs, what is the probability that s/he will arrive at a given page?

• The PageRank of a page captures this notion
  – More “popular” or “worthwhile” pages get a higher rank
PageRank: Visually
PageRank: Defined

• Given page x with incoming links $t_1 \ldots t_n$, where
  – $C(t_i)$ is the out-degree of $t_i$
  – $\alpha$ is probability of randomly entering new URL (usually 0.15)
  – $N$ is the total number of nodes in the graph

• Can’t compute directly: recursively defined in PR

$$PR(x) = \alpha \left( \frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$$
PageRank computation: Intuition

- Calculation is iterative: maintain an estimate of PageRank at iteration $i$: $PR_i$

- Sketch of algorithm
  - Start with seed $PR_i$ values
  - $PR_{i+1}$ is based on $PR_i$: each page distributes $PR_i$ “credit” to all pages it links to
  - Each target page adds up “credit” from in-bound links to compute $PR_{i+1}$
  - Iterate until values converge

- Each page distributes its $PR_i$ to all pages it links to. Linkees add up their awarded rank fragments to find their $PR_{i+1}$
Map step: break page rank into even fragments to distribute to link targets

Reduce step: add together fragments into next PageRank

Iterate for next step...
Phase 1: Parse HTML

• **Map** task takes \((\text{URL}, \text{page content})\) pairs and maps them to \((\text{URL}, (\text{init-rank}, \text{list-of-urls}))\)
  – **init-rank** is the “seed” PageRank for URL
  – **list-of-urls** contains all pages pointed to by URL

• **Reduce** task is just the identity function
Phase 2: PageRank distribution

• **Map** task takes \((\text{URL}, (\text{cur\_rank}, \text{url\_list}))\)
  – For each \(u\) in \(\text{url\_list}\), emit \((u, \text{cur\_rank}/|\text{url\_list}|)\)
  – Also emit \((\text{URL}, \text{url\_list})\) to carry the points-to list along through iterations

• **Reduce** task gets \((\text{URL}, \text{url\_list})\) and many \((\text{URL}, \text{val})\) values
  – Sum \(\text{vals}\), weight with \((1-\alpha)\), add \(\alpha(1/N)\) term
  – Emit \((\text{URL}, (\text{new\_rank}, \text{url\_list}))\)

• Iterate until finished

\[
PR(x) = \alpha \left( \frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}
\]
PageRank conclusions

- MapReduce runs the “heavy lifting” in iterated computation

- Key element in parallelization is independent PageRank computations in a given step

- Parallelization requires thinking about minimum data partitions to transmit (e.g., compact representations of graph rows)
  - Even the implementation shown today doesn't actually scale to the whole Internet; but it works for intermediate-sized graphs
Follow-on readings

**Percolator**, Peng and Dabek (USENIX OSDI ’10)

- System for incrementally processing updates to a large data set
- Deployed to create the Google web search index
- Replaces batch-based indexing with incremental processing
- Reduces the average age of documents in Google search results by 50%.