MapReduce: Simplified Data Processing on Large Clusters

CSE 454

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

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 [list$_2$ list$_3$ …]***)
- (map square '(1 2 3 4))
  - (1 4 9 16)
- (reduce + '(1 4 9 16))
  - 30
- (reduce + (map square (map – l_1 l_2))))

Unary operator

Binary operator

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)"

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 (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

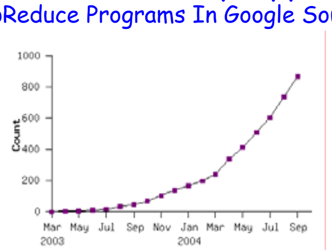
Reverse Web-Link Graph

- 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

Model is Widely Applicable
MapReduce Programs In Google Source Tree



Example uses:

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

Implementation Overview

Typical cluster:

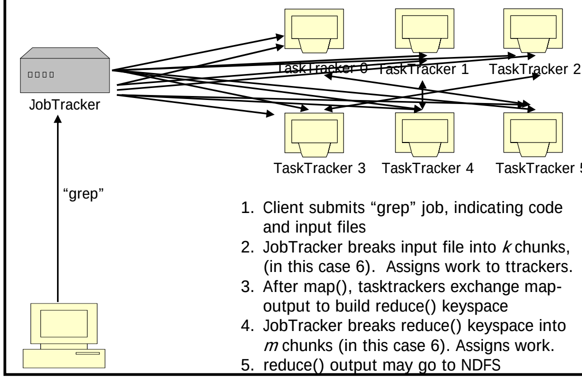
- 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
- Limited bisection bandwidth
- Storage is on local IDE disks
- GFS: distributed file system manages data (SOSP'03)
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines

Implementation is a C++ library linked into user programs

Execution

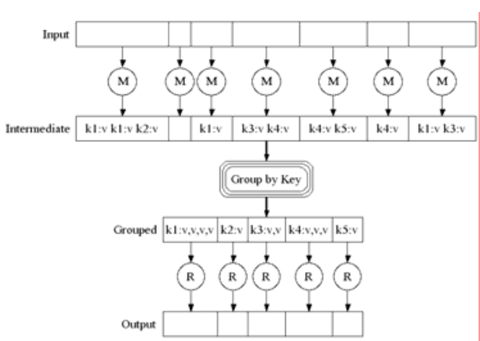
- How is this distributed?
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. Now partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails,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

Job Processing

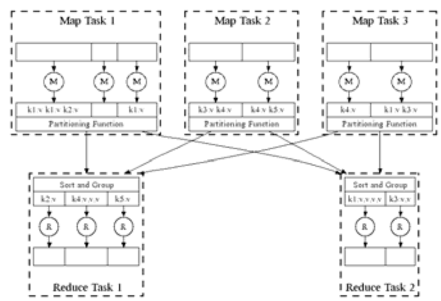


1. Client submits "grep" job, indicating code and input files
2. JobTracker breaks input file into *k* chunks, (in this case 6). Assigns work to ttrackers.
3. After map(), tasktrackers exchange map-output to build reduce() keyspace
4. JobTracker breaks reduce() keyspace into *m* chunks (in this case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

Execution

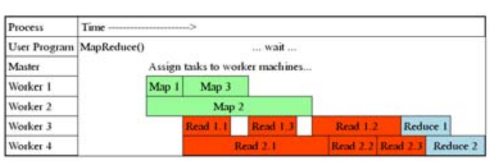


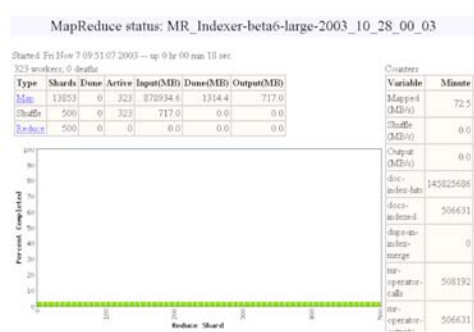
Parallel Execution

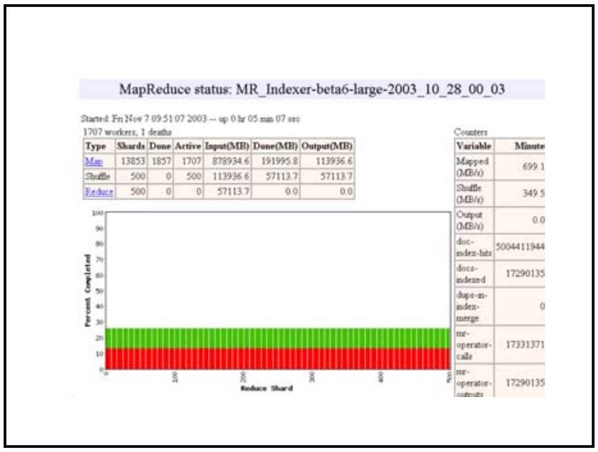


Task Granularity & Pipelining

- 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







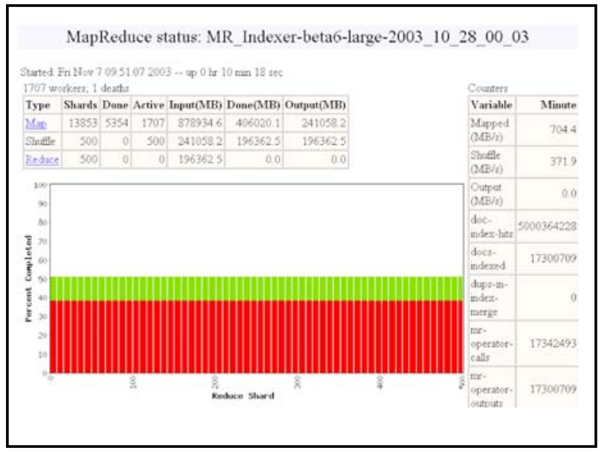
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 05 min 07 sec
1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	1857	1707	878934.6	191995.8	113936.6
Shuffle	500	0	500	113936.6	57113.7	57113.7
Reduce	500	0	0	57113.7	0.0	0.0

Counters

Variable	Minute
Mapped (MB/s)	699.1
Shuffle (MB/s)	349.5
Output (MB/s)	0.0
doc-index-hits	5004411944
docs-indexed	17290135
dups-in-index-merge	0
mr-operator-calls	17331371
mr-operator-outputs	17290135



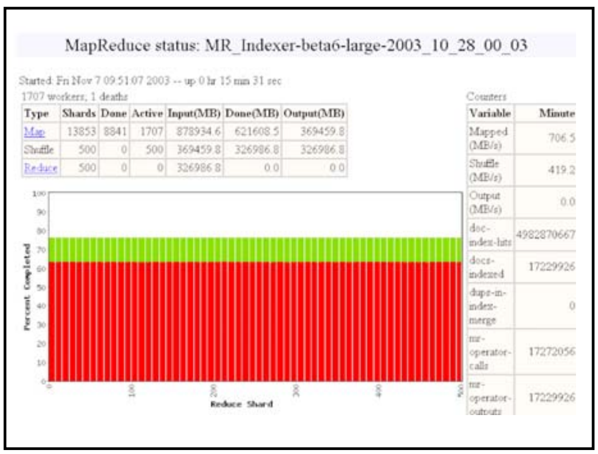
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 10 min 18 sec
1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	5354	1707	878934.6	406020.1	241058.2
Shuffle	500	0	500	241058.2	196362.5	196362.5
Reduce	500	0	0	196362.5	0.0	0.0

Counters

Variable	Minute
Mapped (MB/s)	704.4
Shuffle (MB/s)	371.9
Output (MB/s)	0.0
doc-index-hits	5000364228
docs-indexed	17300709
dups-in-index-merge	0
mr-operator-calls	17342493
mr-operator-outputs	17300709



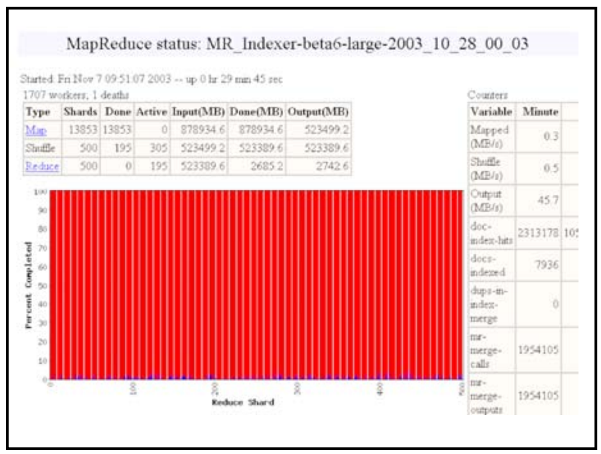
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 15 min 31 sec
1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	8841	1707	878934.6	621608.5	369459.8
Shuffle	500	0	500	369459.8	326986.8	326986.8
Reduce	500	0	0	326986.8	0.0	0.0

Counters

Variable	Minute
Mapped (MB/s)	706.5
Shuffle (MB/s)	419.2
Output (MB/s)	0.0
doc-index-hits	4982870667
docs-indexed	17229926
dups-in-index-merge	0
mr-operator-calls	17272056
mr-operator-outputs	17229926



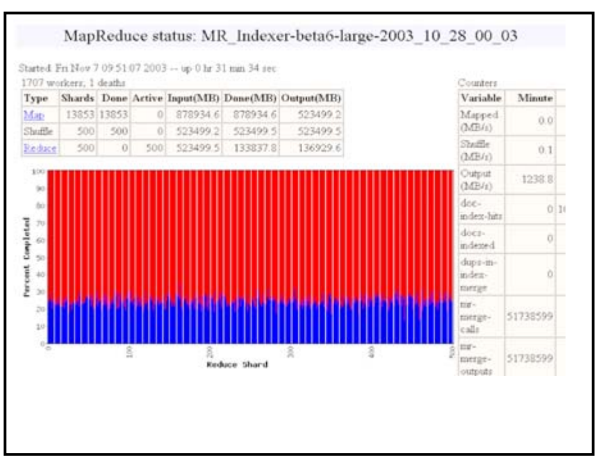
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 29 min 45 sec
1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	195	305	523499.2	523389.6	523389.6
Reduce	500	0	195	523389.6	2685.2	2742.6

Counters

Variable	Minute
Mapped (MB/s)	0.3
Shuffle (MB/s)	0.5
Output (MB/s)	45.7
doc-index-hits	2313178
docs-indexed	7936
dups-in-index-merge	0
mr-merge-calls	1954105
mr-merge-outputs	1954105



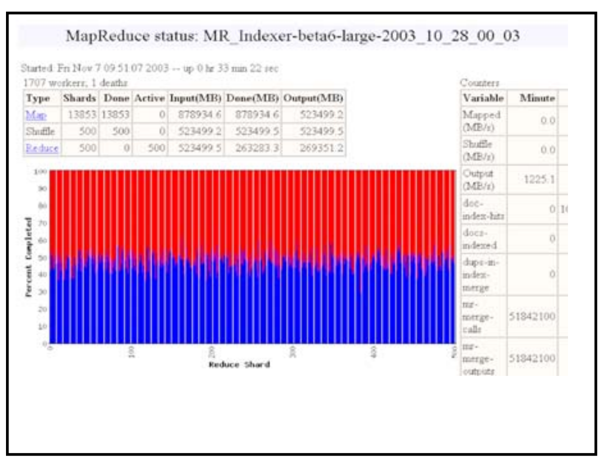
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 31 min 34 sec
1707 workers; 1 deaths

Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	133837.8	136929.6

Counters

Variable	Minute
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.1
Output (MB/s)	1238.8
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	51738599
mr-merge-outputs	51738599



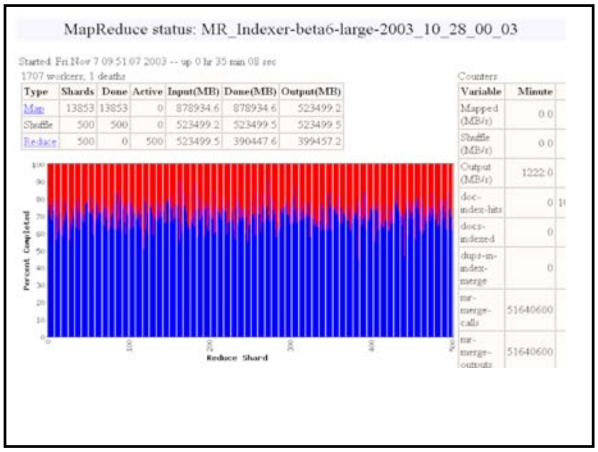
MapReduce status: MR_Indexer-beta6-large-2003_10_28_00_03

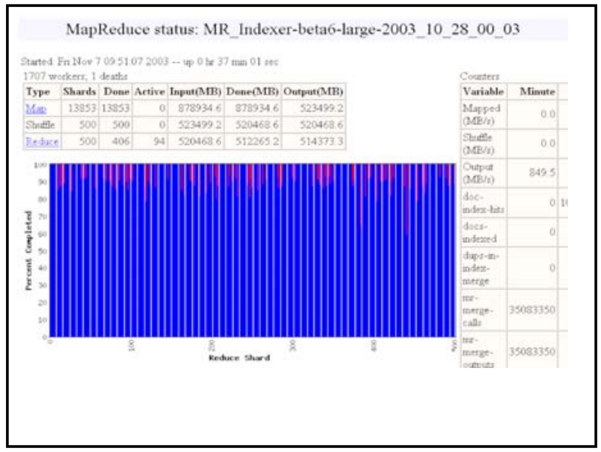
Started: Fri Nov 7 09:51:07 2003 -- up 0 hr 33 min 22 sec
1707 workers; 1 deaths

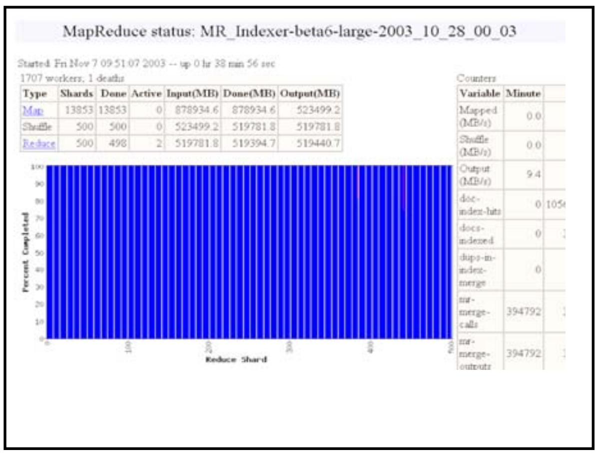
Type	Shards	Done	Active	Input(MB)	Done(MB)	Output(MB)
Map	13853	13853	0	878934.6	878934.6	523499.2
Shuffle	500	500	0	523499.2	523499.5	523499.5
Reduce	500	0	500	523499.5	263283.3	269351.2

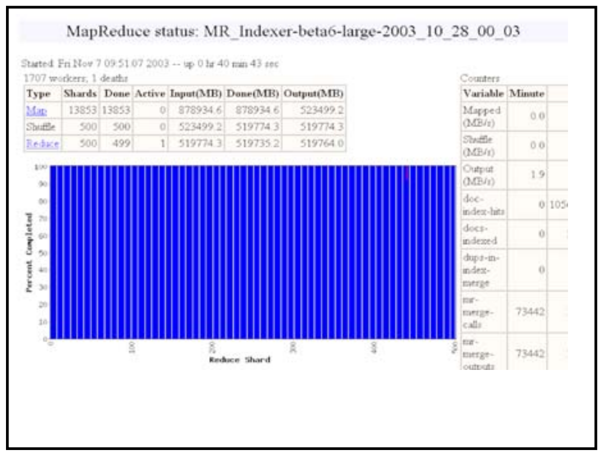
Counters

Variable	Minute
Mapped (MB/s)	0.0
Shuffle (MB/s)	0.0
Output (MB/s)	1225.1
doc-index-hits	0
docs-indexed	0
dups-in-index-merge	0
mr-merge-calls	51842100
mr-merge-outputs	51842100









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

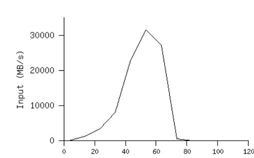
Performance

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

MR_Sort

- 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