6.824 2015 Lecture 13: MapReduce

Why MapReduce?
  Second look for fault tolerance and performance
  Starting point in current enthusiasm for big cluster computing
  A triumph of simplicity for programmer
  Bulk orientation well matched to cluster with slow network
  Very influential, inspired many successors (Hadoop, Spark, &c)

Cluster computing for Big Data
  1000 computers + disks
  a LAN
  split up data+computation among machines
  communicate as needed
  similar to DSM vision but much bigger, no desire for compatibilty

Example: inverted index
  e.g. index terabytes of web pages for a search engine
  Input:
    A collection of documents, e.g. crawled copy of entire web
    doc 31: i am alex
    doc 32: alex at 8 am
  Output:
    alex: 31/3 32/1 ...
    am: 31/2 32/4 ...
  Map(document file i):
    split into words
    for each offset j
      emit key=word[j] value=i/j
  Reduce(word, list of d/o)
    emit word, sorted list of d/o

Diagram:
  * input partitioned into M splits on GFS: A, B, C, ...
  * Maps read local split, produce R local intermediate files (A0, A1 .. AR)
  * Reduce # = hash(key) % R
  * Reduce task i fetches Ai, Bi, Ci -- from every Map worker
  * Sort the fetched files to bring same key together
  * Call Reduce function on each key's values
  * Write output to GFS
  * Master controls all:
    Map task list
    Reduce task list
    Location of intermediate data (which Map worker ran which Map task)

Notice:
  Input is huge -- terabytes
  Info from all parts of input contributes to each output index entry
    So terabytes must be communicated between machines
  Output is huge -- terabytes

The main challenge: communication bottleneck
  Three kinds of data movement needed:
    Read huge input
    Move huge intermediate data
    Store huge output
  How fast can one move data?
    RAM: 1000*1 GB/sec =   1000 GB/sec
    disk: 1000*0.1 GB/sec = 100 GB/sec
    net cross-section: 10 GB/sec

```
  Explain host link b/w vs net cross-section b/w

What are the crucial design decisions in MapReduce?
  Contrast to DSM (TreadMarks or IVY, or even put/get in key/value store):
    They allow arbitrary random interaction among threads.
    But: latency sensitive, poor throughput efficiency.
    But: recovering from crashed compute server very hard;
         probably need global checkpoint, since states are
         intertwined.
  Maps and Reduces work on local data -> reduced network communication.
    For Map, split storage and computation in the same way, use local disk.
  Maps and Reduces work on big batches of data -> no small latency-sensitive network
messages.
  Very little interaction:
    Maps and Reduces can't interact with each other directly.
    No interaction across phase boundaries.
    -> Can re-execute single Map/Reduce independently, no need for e.g. global checkpoint.
  Programmer can't directly cause network communication,
    but has indirect control since Map specifies key.

Where does MapReduce input come from?
  Input is striped+replicated over GFS in 64 MB chunks
  But in fact Map always reads from a local disk
    They run the Maps on the GFS server that holds the data
  Tradeoff:
    Good: Map reads at disk speed, much faster than over net from GFS server
    Bad: only two or three choices of where a given Map can run
         potential problem for load balance, stragglers

Where does MapReduce store intermediate data?
  On the local disk of the Map server (not in GFS)
  Tradeoff:
    Good: local disk write is faster than writing over network to GFS server
    Bad: only one copy, potential problem for fault-tolerance and load-balance

Where does MapReduce store output?
  In GFS, replicated, separate file per Reduce task
  So output requires network communication -- slow
  The reason: output can then be used as input for subsequent MapReduce

The Question: How soon after it receives the first file of
  intermediate data can a reduce worker start calling the application's
  Reduce function?

Why does MapReduce postpone choice of which worker runs a Reduce?
  After all, might run faster if Map output directly streamed to reduce worker
  Dynamic load balance!
  If fixed in advance, one machine 2x slower -> 2x delay for whole computation
    and maybe the rest of the cluster idle/wasted half the time

Will MR scale?
  Will buying 2x machines yield 1/2 the run-time, indefinitely?
  Map calls probably scale
    2x machines -> each Map's input 1/2 as big -> done in 1/2 the time
    but: input may not be infinitely partitionable
    but: tiny input and intermediate files have high overhead
  Reduce calls probably scale
    2x machines -> each handles 1/2 as many keys -> done in 1/2 the time
    but: can't have more workers than keys
    but: limited if some keys have more values than others
      e.g. "the" has vast number of values for inverted index
```

```
      so 2x machines -> no faster, since limited by key w/ most values
  Network may limit scaling, if large intermediate data
    Must spend money on faster core switches as well as more machines
    Not easy -- a hot R+D area now
  Stragglers are a problem, if one machine is slow, or load imbalance
    Can't solve imbalance w/ more machines
  Start-up time is about a minute!!!
    Can't reduce w/ more machines (probably makes it worse)
  More machines -> more failures

Now let's talk about fault tolerance
  The challenge: paper says one server failure per job!
  Too frequent for whole-job restart to be attractive

The main idea: Map and Reduce are deterministic, functional, and independent,
    so MapReduce can deal with failures by re-executing
  Often a choice:
    Re-execute big tasks, or
    Save output, replicate, use small tasks
  Best tradeoff depends on frequency of failures and expense of communication

What if a worker fails while running Map?
  Can we restart just that Map on another machine?
    Yes: GFS keeps copy of each input split on 3 machines
  Master knows, tells Reduce workers where to find intermediate files

If a Map finishes, then that worker fails, do we need to re-run that Map?
  Intermediate output now inaccessible on worker's local disk.
  Thus need to re-run Map elsewhere *unless* all Reduce workers have
    already fetched that Map's output.

What if Map had started to produce output, then crashed:
  Will some Reduces see Map's output twice?
  And thus produce e.g. word counts that are too high?

What if a worker fails while running Reduce?
  Where can a replacement worker find Reduce input?
  If a Reduce finishes, then worker fails, do we need to re-run?
    No: Reduce output is stored+replicated in GFS.

Load balance
  What if some Map machines are faster than others?
    Or some input splits take longer to process?
  Don't want lots of idle machines and lots of work left to do!
  Solution: many more input splits than machines
  Master hands out more Map tasks as machines finish
  Thus faster machines do bigger share of work
  But there's a constraint:
    Want to run Map task on machine that stores input data
    GFS keeps 3 replicas of each input data split
    So only three efficient choices of where to run each Map task

Stragglers
  Often one machine is slow at finishing very last task
    h/w or s/w wedged, overloaded with some other work
  Load balance only balances newly assigned tasks
  Solution: always schedule multiple copies of very last tasks!

How many Map/Reduce tasks vs workers should we have?
  They use M = 10x number of workers, R = 2x.
  More => finer grained load balance.
```

```
  More => less redundant work for straggler reduction.
  More => spread tasks of failed worker over more machines, re-execute faster.
  More => overlap Map and shuffle, shuffle and Reduce.
  Less => big intermediate files w/ less overhead.
  M and R also maybe constrained by how data is striped in GFS.
    e.g. 64 MByte GFS chunks means M needs to total data size / 64 MBytes

Let's look at paper's performance evaluation

Figure 2 / Section 5.2
  Text search for rare 3-char pattern, just Map, no shuffle or reduce
  One terabyte of input
  1800 machines
  Figure 2 x-axis is time, y-axis is input read rate
  60 seconds start-up time *omitted*! (copying program, opening input files)
  Why does it take so long (60 seconds) to reach the peak rate?
  Why does it go up to 30,000 MB/s? Why not 3,000 or 300,000?
    That's 17 MB/sec per server.
    What limits the peak rate?

Figure 3(a) / Section 5.3
  sorting a terabyte
  Should we be impressed by 800 seconds?
  Top graph -- Input rate
    Why peak of 10,000 MB/s?
    Why less than Figure 2's 30,000 MB/s? (writes disk)
    Why does read phase last abt 100 seconds?
  Middle graph -- Shuffle rate
    How is shuffle able to start before Map phase finishes?
      more map tasks than workers
    Why does it peak at 5,000 MB/s?
      net cross-sec b/w abt 18 GB/s
    Why a gap, then starts again?
      runs some Reduce tasks, then fetches more
    Why is the 2nd bump lower than first?
      maybe competing w/ overlapped output writes
  Lower graph -- Reduce output rate
    How can reduces start before shuffle has finished?
      again, shuffle gets all files for some tasks
    Why is output rate so much lower than input rate?
      net rather than disk; writes twice to GFS
    Why the gap between apparent end of output and vertical "Done" line?
      stragglers?

What should we buy if we wanted sort to run faster?
  Let's guess how much each resource limits performance.
  Reading input from disk: 30 GB/sec = 33 seconds (Figure 2)
  Map computation: between zero and 150 seconds (Figure 3(a) top)
  Writing intermediate to disk: ? (maybe 30 Gb/sec = 33 seconds)
  Map->Reduce across net: 5 GB/sec = 200 seconds
  Local sort: 2*100 seconds (gap in Figure 3(a) middle)
  Writing output to GFS twice: 2.5 GB/sec = 400 seconds
  Stragglers: 150 seconds? (Figure 3(a) bottom tail)
  The answer: the network accounts for 600 of 850 seconds

Is it disappointing that sort uses only a small fraction of cluster CPU power?
  After all, only 200 of 800 seconds were spent sorting.
  Alternate view: they made good use of RAM and network.
    Probably critical that 1800 machines had more then a terabyte of RAM.
    And sorting is perhaps inherently about movement, not CPU.
  If all they did was sort, they should sell CPUs/disks and buy a faster network.
```

Modern data centers have relatively faster networks
  e.g. FDS's 5.5 terabits/sec cross-section b/w vs MR paper's 150 gigabits/sec
  while CPUs are only modestly faster than in MR paper
  so today bottleneck might have shifted away from net, towards CPU

For what applications *doesn't* MapReduce work well?
  Small updates (re-run whole computation?)
  Small unpredictable reads (neither Map nor Reduce can choose input)
  Multiple shuffles (can use multiple MR but not very efficient)
    In general, data-flow graphs with more than two stages
  Iteration (e.g. page-rank)

MapReduce retrospective
  Single-handedly made big cluster computation popular
    (tho coincident w/ big datacenters, cheap servers, data-oriented companies)
  Hadoop is still very popular
  Inspired better successors (Spark, DryadLINQ, &c)