UCL Department of Computer Science
CS M038/GZ06: Mobile and Cloud Computing
Spring 2014
Kyle Jamieson and Brad Karp

MapReduce: Simplified Data Processing on Large Clusters (Dean and Ghemawat, USENIX OSDI 2004, pdf)

MapReduce

- Background: Google engineers used to implement special-purpose programs to process large datasets
- Computation had to be distributed across many machines
  - Resulting details: how to parallelize the computation, distribute data, handle failures.
  - Details obscured the code, made it complex
- MapReduce is a simple abstraction to hide the messy details

Implemenetation overview

- Bisection bandwidth: divide the network into two parts. This is the bandwidth between the parts. Examples: ring has BB of 2, tree 1.

MapReduce execution overview

- Partition input data into M **splits**, each split gets mapped to a task on a different machine in parallel.
  - User optionally specifies M.
  - Input and output data resides in GFS
  - Locality: Master takes location information of input data into account, tries to assign map tasks to machines that already have a local copy of data, or machines on same LAN.
- Partition intermediate key space into R pieces (*e.g.* hash(key) mod R). This generates R reduce tasks (each responsible for different partition) that get mapped to different machines in parallel as well.
  - User specifies R and partitioning function.

Execution flow

1. Break file into M splits; fork many copies of program on different machines
2. Master machine assigns either map or reduce task to any idle worker machine
3. Map worker: Read split, parse (k, v) pairs, pass each pair to user's Map function, buffer result in memory.
4. Map worker periodically writes buffered pairs to local disk, partitioned into R regions and updates master with region locations and sizes; master remembers these.
5. Reduce worker sends a remote procedure call to map worker to read pairs from map workers' disks. Sort pairs by intermediate key and group by intermediate key (many different intermediate keys map to same reduce task). **If data is too large to fit into memory, reduce worker uses disk.**
6. Reduce worker: for each intermediate key, pass all intermediate values to user's Reduce function
7. Master wakes up user program when all reduce tasks finish; output is R files, one for each reduce task.
   - Master remembers (idle, in-progress, completed) state for each map or reduce task.

Fault tolerance

- Skipping bad records: Sometimes map/reduce functions crash deterministically on certain data. Sometimes not feasible to fix the bug (others' code).
- Solution: Store the sequence number of the data given to the map/reduce function and send it to master on crash.
- If master sees more than one failure on a particular record, it indicates that that record should be skipped when it re-issues execution of the corresponding map/reduce task.

Performance evaluation

- What's the high-level goal of the performance evaluation? To show how long a large MapReduce computation takes, and why. No head-to-head performance comparison here, however.
- Setup: 1,800 machine cluster, gig ethernet link

Data transfer rates for *sort* program

- Map function extracts sort key from text line and emits key, text line as key, value; Reduce is the identity function.
- Experiment run with 2-way replication in GFS
- $M = 15,000$ (64 MB input data chunks), $R = 4,000$
- Input rate: All map tasks finish before 200 seconds. Less peak input rate than grep, since sort expends I/O B/W on writing to disk, unlike grep.
- Shuffle (rate at which data flows over net from map to reduce tasks)
  - Begins as soon as first map task completes
  - First hump: first batch of 1,700 tasks
  - Second hump (at 300 seconds): some R tasks from first batch complete, and new R tasks start moving data
- Output (rate at which data written to final output files by R task)
  - Delay between end of first shuffle and start of write due to sorting data
- Input rate greater than shuffle, output rate b/c local reads
- Shuffle rate greater than output rate b/c output phase writes two copies

This is the sort benchmark, where the map function extracts a sorting key from each line of a document's text, and outputs (key, line) pairs. Of particular note here is that the intermediate key space is partitioned into is $R = 4,000$ pieces and the partitioning function uses the initial bytes of the key.

Three observations:

- Looking at the top graph of Figure 3 (a), we see that all the input data have been read by $t = 150$ seconds
- The accompanying text tells us that all map tasks finish before t = 200 seconds
- The location of the first "hump" in the middle graph of Figure 3 (a) and the accompanying text tells us that the first batch of approximately 1,700 reduce tasks begins executing soon after the first map task completes and finishes roughly at t = 300 seconds.

The master starts the shuffle work for the first 1,700 reduce tasks as the first map tasks finish executing around 30 seconds, and that work continues until about 200 seconds. Since all intermediate key, value pairs are available from the map workers before 200 seconds, the master can finish the first 1,700 reduce tasks' shuffle work soon after $t = 200$.

Since each reduce task "owns" its partition of the intermediate key space, MapReduce has all the data needed for that task once the shuffle work finishes, and so reduce workers can begin writing output data (bottom plot) soon after $t = 200$.

Implicit in the paper is that since we partition based on the first few bytes of the key, we can do the final grand sort by sorting the R = 4,000 output files by the corresponding first few bytes of the key.

Effect of disabling backup tasks

- This slide shows sort with backup tasks DISABLED
- Done after 1283 seconds (44% more than normal)
- At 960 seconds, waiting for just *five* slow reduce tasks

Effect of machine failures

- Kill 200 out of 1,700 worker tasks at about $t = 250$ seconds
- Negative input **and** shuffle rate since kill both M and R tasks, and some input and shuffle work needs to be redone
- Finish in 933 seconds, compared to 850 seconds normally

PageRank: Defined

- With probability α, user randomly enters new URL and lands at one of the N pages total
- With probability $1 - \alpha$, user follows link to x from one of the n pages $t_1, \ldots, t_n$ that link to x with PageRank $PR(t_i)$. Assume equal probability of following any link from a given page, so divide by the number of links from that page.