S1 INTRO: MAPREDUCE

S2 MOTIVATION:

- Large datasets: thousands of terabytes
- Allows users to focus on their computation
- Have hundreds or thousands of commodity computers
- Want abstract details such as handling of data distribution, machine faults, computation distribution and load balancing?

S3 GOOGLE'S MAP REDUCE

S4 WHAT IS IT?

S5 PROGRAMMING MODEL

- Functional programming model & supporting framework
- Allows you to easily utilize hundreds or thousands of PCs by defining map and reduce

S6 MAPREDUCE MODEL

- Map program reads "records" from input file, filtering and transformations, outputs a set of intermediate pairs
- Reduces, aggregates and summarizes for particular key
- Shuffle/sort function, hash function
  - (*hash(key) mod* R where *R* is *user defined*, usually 4K), though any deterministic function will suffice
  - When bucket fills, written to disk. The map program terminates with M output files, one for each bucket.
- SQL: map like group-by, reduce like aggregate

S7 WORD COUNT EXAMPLE

- Map just emits word and count of occurrences, 1 in simple example Reduce sums together counts emitted for each word. Full code in appendix.
- Names of input/output files and tuning parameters. Then invoke MapReduce function

S8 APPLICATIONS

S9 WHY USE?

- Tasks such as shuffling data from one representation to another or extracting data
- Automatic parallelization, load balancing, network and disk transfers, handling of machine faults, robustness
- Google uses it internally for thousands of jobs, has respawned many OSS alternatives

S10 HOW DOES IT WORK

Let's look internally at how the model is implemented by Google and what happens when the MapReduce routine is called on word counting example

S11 EXECUTION OVERVIEW

1. Split input file- start many copies of program – 1 copy's special "master"
2. Once split, each split of input file is a task and idle workers must be assigned
   - Tries to pick workers that have the data on disk (GFS also stores redundantly): locality
3. Buffered pairs written to local disk, partitioned into R regions by shuffle function
4. Reduce worker notified about intermediate file location, reads data
5. Passes k/vs to user Reduce function, output of Reduce appended to final output file for reduce partition

S12 WORD COUNT REVISED

- As many buckets as reduce task, hashing has thus been used for shuffle function

S13 PIPELINING

- Want many more map tasks than workers, with 2k machines, often 200k map, 5k reduce tasks
- Minimizes time for fault recovery
- Can pipeline shuffling with map execution, usually they're sequential

S14 FAULT TOLERANCE

- Dead workers. Workers send periodic heartbeats to master. 3 cases:
  - Map failure: reexecute completed and in-progress tasks
  - Reduce failure: Re-execute in-progress tasks
  - Master: Google's implementation aborts and leaves it up to user code to handle
  - Google lost 1.6k of 1.8k machines once, and still finished computation

- Slow workers
  - Redundant execution, because as we shall see, slow workers significantly lengthen completion time
  - Solution: near end of phase, assign backup copies of map/reduce tasks to idle workers

S15 EFFECTIVE?

S16 GREP

- Scans through 1 TB data for rare (92k in (10^10) 10 billion records) 3-character pattern
- Locality helpful, as peak data 31 GB/s with ~1800 workers assigned, without 10 GB/s limited by rack switches
- Entire computation takes about 80s
- Startup delay due to program propagation to workers, interacting with GFS to open 1k input files

S17 SORTING

- Sorts 10 billion 100-byte records, 10 TB data
- Map: extract 10-byte sorting key from text line and emits the key and original text line as intermediate k, v
- Reduce: identity
- a: 839s (13 min)
  - Top: Input rate less than grep, as half time and bidwidth writing intermediat output. Grep much smaller
  - Mid: Shuffle starts right after first map. Hump from first 1700 reduce tasks, only 1700 machines, thus it goes down
  - Bot: Delay between end of first shuffle and start of writing due to processing intermediate data
  - Input rate > shuffle+output due to locality optimization, most data is read from disk

    - Shuffle rate > output because output writes twice (GFS)
- b: 1235s (20 min)
    - Similar to a, but without backup tasks. Except with long tail
    - At end, all but 5 tasks completed
    - 44% longer
- c: 886s (14 min)
    - Intentionally kill 200/1746 worker processes minutes into the computation
    - Immediately restart new worker processes on machines
    - Only 5% longer
    - Negative input since previously completed map disappears and needs to be redone. Re-execution happens quickly.

S18 USEFUL?

S19 IMPLEMENTATION

- Implementation
    - MapReduce used to be Google's term, and the implementation discussed in paper is proprietary and not accessible, GFS proprietary too
    - Apache has the Hadoop ecosystem, with OSS HDFS and OSS implementation of MapReduce

S20 PROS/CONS

- Cons
    - It's batch oriented in nature
    - Not every algorithm can boil down to MapReduce, such as algorithms that require global state
- Frequent I/Os required for fault tolerance, reduces efficiency a lot
- Efficiency, map/reduce are blocking, thus can't use pipelining between map/reduce operations

S21 CONCLUSIONS

- High level, get to focus on problem and library deals with details