To Orchestrate a Job in a Cluster
A job comprises many a task
What could be so hard, you ask?
A job’s done, when every task wraps up
Deal you must, with every hiccup
Machines may slowdown or go bust
For no reason nor rhyme
Try to complete, you must
All tasks, at roughly the same time

Topics covered in this lecture
- Map Reduce

Programming model
- Computation takes a set of input key/value pairs
- Produces a set of output key/value pairs
- Express the computation as two functions:
  - Map
  - Reduce

Map
- Takes an input pair
- Produces a set of intermediate key/value pairs
Mappers

- If map operations are **independent** of each other they can be performed in parallel
  - **Shared nothing**
- This is usually the case

MapReduce library

- **Groups** all intermediate values with the same intermediate key
- **Passes** them to the Reduce function

Reduce function

- Accepts intermediate key \( I \) and
  - Set of values for that key
- **Merge** these values together to get
  - Smaller set of values

Counting number occurrences of each word in a large collection of documents

```java
map (String key, String value)
//key: a word
//value: document contents
for each word \( w \) in value
 EmitIntermediate(w, "1")
```

Counting number occurrences of each word in a large collection of documents

```java
reduce (String key, Iterator values)
//key: a word
//values: a list of counts
int result = 0;
for each \( v \) in values
 result += ParseInt(v);
Emit(AsString(result));
```

**Some together all counts emitted for a particular word**

MapReduce specification object contains

- **Names of**
  - Input
  - Output
- **Tuning parameters**
Map and reduce functions have associated types drawn from different domains

\[
\text{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2) \\
\text{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)
\]

What's passed to-and-from user-defined functions?
- Strings
- User code converts between
  - String
  - Appropriate types

Programs expressed as MapReduce computations:
Distributed Grep
- Map
  - Emit line if it matches specified pattern
- Reduce
  - Just copy intermediate data to the output
  - The reducer here is an identity function

Counts of URL access frequency
- Map
  - Process logs of webpage requests
  - Output \(<URL, 1>\)
- Reduce
  - Add together all values for a particular URL
  - Output \(<URL, \text{total count}>\)

Reverse Web-link Graph
- Map
  - Outputs \(<\text{target}, \text{source}>\) pair for each target URL found in page source
- Reduce
  - Concatenate list of all sources for a target URL
  - Output \(<\text{target}, \text{list(source)}>\)
**Term-Vector per Host**

- Summarizes important terms that occur in a set of documents: `<word, frequency>`
- For each input document, the Map
  - Emits `<hostname, term vector>`
- Reduce function
  - Has all per-document vectors for a given host
  - Add term vectors; discard away infrequent terms
  - `<hostname, term vector>`

**Inverted Index**

- **Map**
  - Parse each document
  - Emit `<word, document ID>`
- **Reduce**
  - Accept all pairs for a given word
  - Sort document IDs
  - Emit `<word, list(document ID) pair`

**Implementation**

- Machines are *commodity* machines
- GFS is used to manage data stored on the disks

**Execution Overview – Part I**

- Maps distributed across multiple machines
- Automatic partitioning of data into M splits
- Splits are processed *concurrently* on different machines

**Execution Overview – Part II**

- Partition *intermediate* key space into R pieces
- E.g. `hash(key) mod R`
- User specified parameters
  - **Partitioning function**
  - **Number** of partitions (R)
## Execution Overview

**Step I**
- The MapReduce library
  - Splits input files into M pieces
    - 16-64 MB per piece
  - Starts up copies of the program on a cluster of machines

**Step II**
- Program copies
  - One of the copies is a Master
  - There are M map tasks and R reduce tasks to assign
  - Master
    - Picks idle workers
    - Assigns each worker a map or reduce task

**Step III**
- Workers that are assigned a map task
  - Read contents of their input split
  - Parses <key, value> pairs out of the input data
  - Pass each pair to user-defined Map function
  - Intermediate <key, value> pairs from Maps
    - Buffered in Memory

**Step IV**
- Writing to disk
  - Periodically, buffered pairs are written to disk
  - These writes are partitioned
    - By the partitioning function
  - Locations of buffered pairs on local disk
    - Reported back to Master
    - Master forwards these locations to reduce workers

**Step V**
- Reading intermediate data
  - Master notifies Reduce worker about locations
  - Reduce worker reads buffered data from the local disks of Maps
  - Read all intermediate data; sort by intermediate key
    - All occurrences of the same key are grouped together
    - Many different keys map to the same Reduce task
Execution Overview: Step VI
Processing data at the Reduce worker
- Iterate over sorted intermediate data
- For each unique key pass
  - Key + set of intermediate values to Reduce function
- Output of the Reduce function is appended
  - To output file of the reduce partition

Execution Overview: Step VII
Waking up the user
- After all Map & Reduce tasks have been completed
- Control returns to the user code

Master Data Structures
- For each Map and Reduce task
  - State: {idle, in-progress, completed}
  - Worker machine identity
- For each completed Map task store
  - Location and sizes of R intermediate file regions
  - Information pushed incrementally to in-progress Reduce tasks

Fault Tolerance
Worker failures
- Master pings worker periodically
- After a certain number of failed pings
  - Master marks worker as having failed
- Any Map task completed by failed worker?
  - Reset to initial idle state
  - Eligible for rescheduling

Why completed Map tasks are reexecuted
- Output is stored on local disk of failed machine
  - Inaccessible
- All reduce workers are notified about reexecution
- Reduce tasks do not need to be reexecuted
  - Output stored in GFS
Master Failures

- Could checkpoint at the Master
  - Data structures are well-defined
- However, since there is only one Master
  - Assumption is that failure is unlikely
- If there is a Master failure?
  - MapReduce computation is aborted
  - Client must check and retry MapReduce operation

Semantics in the presence of failures:
If map and reduce operators are deterministic

- Distributed execution output is identical to
  - Non-faulting, sequential execution
- Atomic commits of map and reduce task outputs help achieve this

Each in-progress task writes output to private temporary files

- Map task produces $R$ such files
  - When task completes, Map sends this info to the Master
- Reduce task produces one such file
  - When reduce completes, worker atomically:
    - Renames temporary file to final output file
    - Uses GFS to do this

Locality

- Conserve network bandwidth
- Input files managed by GFS
- MapReduce master takes location of input files into account
- Schedule task on machine that contains a replica of the input slice

Locality and its impact when running large MapReduce tasks

- Most input data is read locally
- Consumes no network bandwidth

Task Granularity
### Task Granularity
- Subdivide map phase into \( M \) pieces
- Subdivide reduce phase into \( R \) pieces
- \( M, R >> \) number of worker machines
- Each worker performing many different tasks:
  - Improves dynamic load balancing
  - Speeds up recovery during failures

### Practical bounds on how large \( M \) and \( R \) can be
- Master must make \( O(M + R) \) scheduling decisions
- Keep \( O(MR) \) state in memory

### Typical values used at Google
- \( M = 200,000 \)
- \( R = 5,000 \)
- \( W = 2,000 \) worker machines

### Backup Tasks

### Stragglers
- Machine that takes an unusually long time to complete a map or reduce operation
- Can slow down entire computation
How stragglers arise

- Machine with a bad disk
  - Frequent, correctable errors
  - Read performance drops from 30 MB/s to 1 MB/s
- Over scheduling
  - Many tasks executing on the same machine
  - Competition for CPU, memory, disk or network cycles
- Bug in machine initialization code
  - Processor caches may be disabled

Alleviating the problem of stragglers

- When a MapReduce operation is close to completion
- Schedule backup executions of remaining in-progress tasks
- Task completed when
  - Primary or backup finishes execution
- Significantly reduces time to complete large MapReduce operations

The contents of this slide set are based on the following references

- Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004: 137-150