UNIVERSITY OF CAMBRIDGE

Cloud Computing

MapReduce, Batch Processing

Eva Kalyvianaki
ek264@cam.ac.uk

Contents

- Batch processing: processing large amounts of data at-once, in one-go to deliver a result according to a query on the data.

- Material is from the paper:

  "MapReduce: Simplified Data Processing on Large Clusters",

  By Jeffrey Dean and Sanjay Ghemawat from Google

  published in Usenix OSDI conference, 2004

Motivation: Processing Large-sets of Data

- Need for many computations over large/huge sets of data:
  - Input data: crawled documents, web request logs
  - Output data: inverted indices, summary of pages crawled per host, the set of the most frequent queries in a given day, …
- Most of these computation are relatively straight-forward
- To speedup computation and shorten processing time, we can distribute data across 100s of machines and process them in parallel
- But, parallel computations are difficult and complex to manage:
  - Race conditions, debugging, data distribution, fault-tolerance, load balancing, etc
- **Ideally,** we would like to process data in parallel but not deal with the complexity of parallelisation and data distribution

MapReduce

"A new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library."

- Programming model:
  - Provides abstraction to express computation
- Library:
  - To take care the runtime parallelisation of the computation.

Example: Counting the number of occurrences of each work in the text below from Wikipedia

"*Cloud computing is a recently evolved computing terminology or metaphor based on utility and consumption of computing resources. Cloud computing involves deploying groups of remote servers and software networks that allow centralized data storage and online access to computer services or resources. Cloud can be classified as public, private or hybrid.*"

Word:	**Number of Occurrences**
Cloud	1
computing	**2**
is	1
a	1
recently	1
evolved	1
terminology	1

Programming Model

- Input: a set of key/value pairs
- Output: a set of key/value pairs

- Computation is expressed using the two functions:

1. Map task: a single pair → a list of intermediate pairs
   - map(input-key, input-value) → list(out-key, intermediate-value)
   - $<k_i, v_i> \rightarrow \{ < k_{int}, v_{int} > \}$

1. Reduce task: all intermediate pairs with the same k_{int} → a list of values
   - reduce(out-key, list(intermediate-value)) → list(out-values)
   - $< k_{int}, \{v_{int}\} > \rightarrow < k_o, v_o >$

Example: Counting the number of occurrences of each work in a collection of documents

```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");
```

```
reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

MapReduce Example Applications

- The MapReduce model can be applied to many applications:
  - Distributed grep:
    - map: emits a line, if line matched the pattern
    - reduce: identity function
  - Count of URL access Frequency
  - Reverse Web-Link Graph
  - Inverted Index
  - Distributed Sort
  - ….

MapReduce Implementation

- MapReduce implementation presented in the paper matched Google infrastructure at-the-time:
  1. Large cluster of commodity PCs connected via switched Ethernet
  2. Machines are typically dual-processor x86, running Linux, 2-4GB of mem! (slow machines for today's standards)
  3. A cluster of machines, so failures are anticipated
  4. Storage with (GFS) Google File System (2003) on IDE disks attached to PCs. GFS is a distributed file system, uses replication for availability and reliability.
- Scheduling system:
  1. Users submit jobs
  2. Each job consists of tasks; scheduler assigns tasks to machines

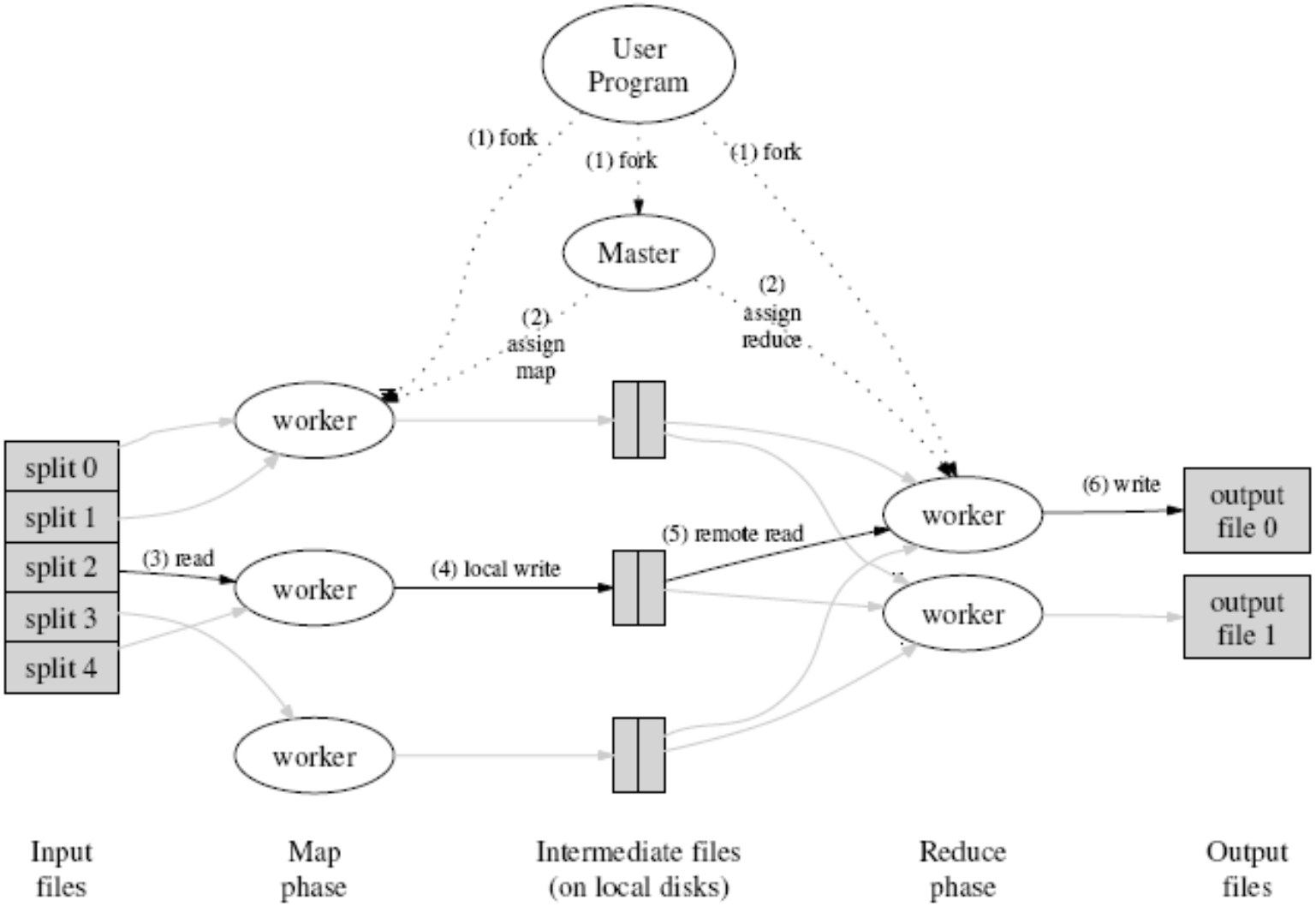
Google File System (GFS)

- File is divided into several chunks of predefined size;
  - Typically, 16-64 MB
- The system replicates each chunk by a number:
  - Usually three replicas
  - To achieve fault-tolerance, availability and reliability

Parallel Execution

- User specifies:
  - **M**: number of map tasks
  - **R**: number of reduce tasks
- Map:
  - MapReduce library splits the input file into **M** pieces
  - Typically 16-64MB per piece
  - Map tasks are distributed across the machines
- Reduce:
  - Partitioning the intermediate key space into **R** pieces
  - `hash(intermediate_key) mod R`
- Typical setting:
  - 2,000 machines
  - M = 200,000
  - R = 5,000

Execution Flow



Master Data Structures

- For each map/reduce task:
  - State status {*idle, in-progress, completed*}
  - Identity of the worker machine (for non-idle tasks)

- The location of intermediate file regions is passed from maps to reducers tasks through the master.
- This information is pushed incrementally (as map tasks finish) to workers that have *in-progress* reduce tasks.

Fault-Tolerance

Two types of failures:

1. worker failures:
   - Identified by sending heartbeat messages by the master. If no response within a certain amount of time, then the worker is dead.
   - *In-progress* and *completed* map tasks are re-scheduled → *idle*
   - *In-progress* reduce tasks are re-scheduled → *idle*
   - Workers executing reduce tasks affected from failed map/workers are notified of re-scheduling
   - Question: Why *completed* tasks have to be re-scheduler?
   - Answer: Map output is stored on local fs, while reduce output is stored on GFS
2. master failure:
   1. Rare
   2. Can be recovered from checkpoints
   3. Solution: aborts the MapReduce computation and starts again

Disk Locality

- Network bandwidth is a relatively scarce resource and also increases latency
- The goal is to save network bandwidth

- Use of GFS that stores typically three copies of the data block on different machines
- Map tasks are scheduled "close" to data
  - On nodes that have input data (local disk)
  - If not, on nodes that are nearer to input data (e.g., same switch)

Task Granularity

- Number of map tasks > number of worker nodes
  - Better load balancing
  - Better recovery

- But, this, increases load on the master
  - More scheduling
  - More states to be saved

- **M** could be chosen with respect to the block size of the file system
  - For locality properties
- **R** is usually specified by users
  - Each reduce tasks produces one output file

Stragglers

- Slow workers delay overall completion time → **stragglers**
  - Bad disks with soft errors
  - Other tasks using up resources
  - Machine configuration problems, etc

- Very close to end of MapReduce operation, master schedules backup execution of the remaining *in-progress* tasks.
- A task is marked as complete whenever either the primary or the backup execution completes.

- Example: sort operation takes 44% longer to complete when the backup task mechanism is disabled.

Refinements: Partitioning Function

- Partitioning function identifies the reduce task
  - Users specify the desired output files they want, R
  - But, there may be more keys than R
  - Uses the intermediate key and R
  - Default: `hash(key) mod R`

- Important to choose well-balanced partitioning functions:
  - `hash(hostname(urlkey)) mod R`
  - For output keys that are URLs

Refinements: Combiner Function

- Introduce a mini-reduce phase before intermediate data is sent to reduce
- When there is significant repetition of intermediate keys
  - Merge values of intermediate keys before sending to reduce tasks
  - Example: word count, many records of the form `<word_name, 1>`. Merge records with the same `word_name`
  - Similar to reduce function
- Saves network bandwidth

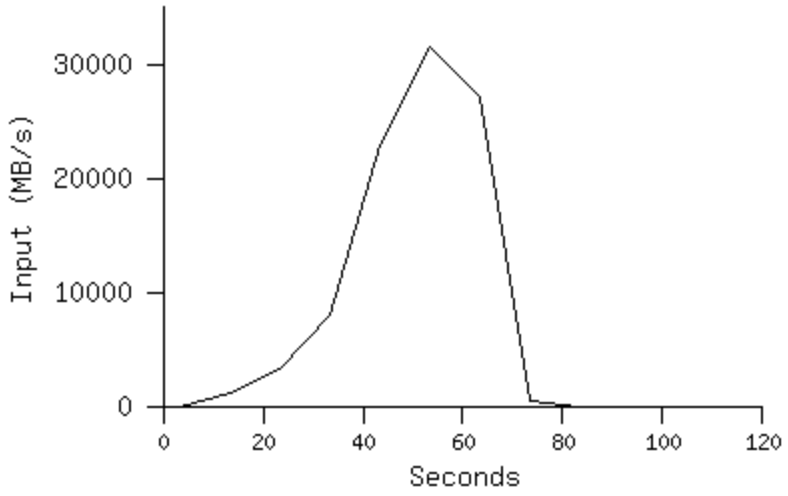
Evaluation - Setup

- Evaluation on two programs running on a large cluster and processing 1 TB of data:
  1. **grep:** search over 10^{10} 100-byte records looking for a rare 3-character pattern
  2. **sort:** sorts 10^{10} 100-byte records

- Cluster configuration:
  - 1,800 machines
  - Each machine has 2 GHz Intel Xeon proc., 4GB mem, 2 160GB IDE disks
  - Gigabit Ethernet link
  - Hosted in the same facility

Grep

- M = 15,000 of 64MB each split
- R = 1
- Entire computation finishes at 150s
- Startup overhead ~60s
  - Propagation of program to workers
  - Delays to interact with GFS to open 1,000 files
  - …
- Picks at 30GB/s with 1,764 workers

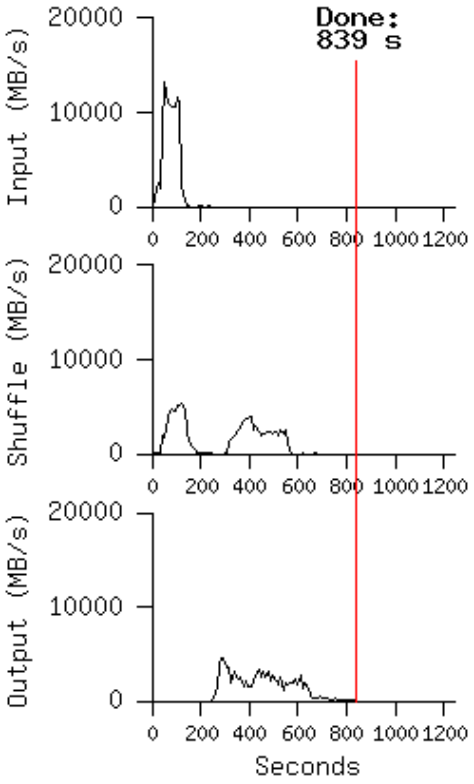


Sort

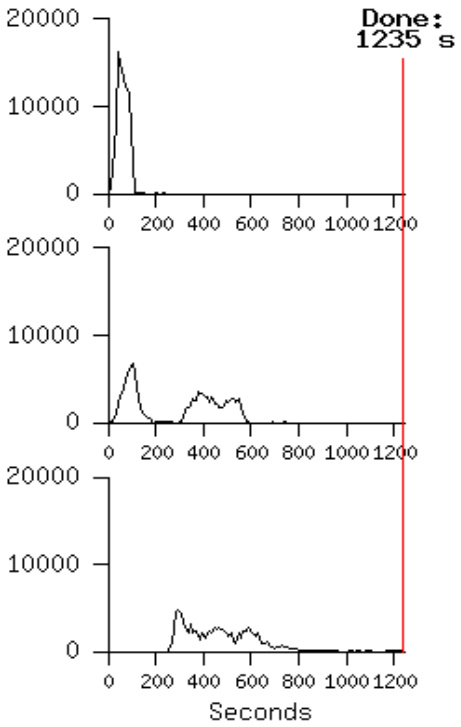
- M = 15,000 splits, 64MB each
- R = 4,000 files
- Workers = 1,700
- Evaluated on three executions:
  - With backup tasks
  - Without backup tasks
  - With machine failures

Sort Results

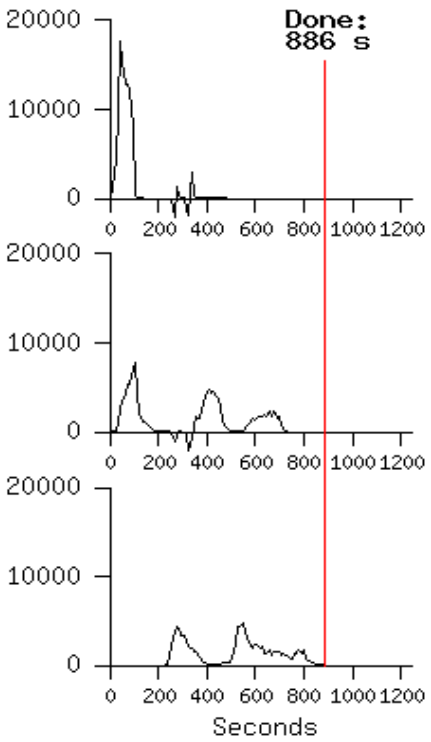
Top: rate at which input is read
Middle: rate at which data is sent from mappers to reducers
Bottom: rate at which sorted data is written to output file by reducers



Normal execution with backup



Without backup tasks, 5 reduce tasks stragglers, 44% increase



With machine failures, 200 out of 1746 workers,

- a 5% increase over normal execution time

Implementation

- First MapReduce library in 02/2003
- Use cases (back then):
  - Large-scale machine learning problems
  - Clustering problems for the Google News
  - Extraction of data for reports Google zeitgeist
  - Large-scale graph computations

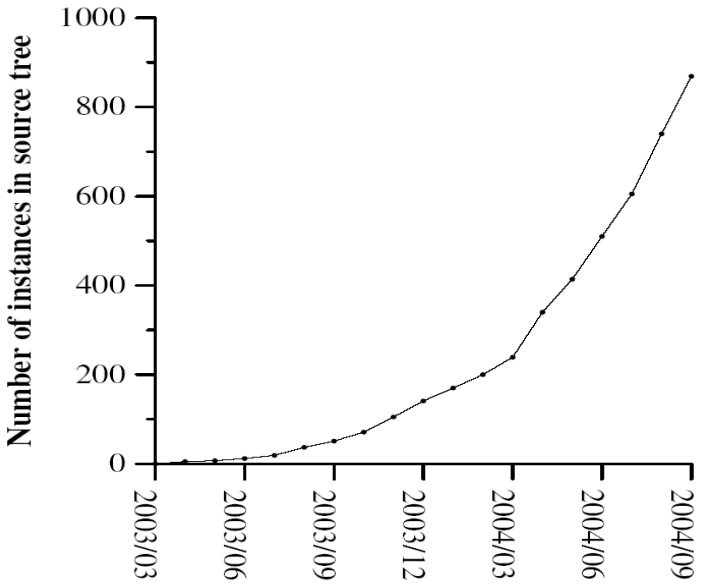


Figure 4: MapReduce instances over time

MapReduce jobs run in 8/2004

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

Summary

- MapReduce is a very powerful and expressive model
- Performance depends a lot on implementation details

- Material is from the paper:

  "MapReduce: Simplified Data Processing on Large Clusters",

  By Jeffrey Dean and Sanjay Ghemawat from Google

  published in Usenix OSDI conference, 2004