MC855 - Projeto em Sistemas de Computação

MapReduce

Islene Calciolari Garcia

Instituto de Computação - Unicamp

Segundo Semestre de 2015

Motivação

- Exemplo retirado do livro do Tom White: *Hadoop: The Definitive Guide*
- Achar a temperatura máxima por ano em um conjunto de arquivos texto
- Fazer todo o trabalho duro em Unix...
- Entender a importância de um framework

Weather dataset

Dados crus

Example 2-1. Format of a National Climate Data Center record

```
0057
332130   # USAF weather station identifier
99999    # WBAN weather station identifier
19500101 # observation date
0300     # observation time
4
+51317   # latitude (degrees x 1000)
+028783  # longitude (degrees x 1000)
FM-12
+0171    # elevation (meters)
99999
V020
320      # wind direction (degrees)
1        # quality code
N
0072
1
00450    # sky ceiling height (meters)
1        # quality code
C
N
010000   # visibility distance (meters)
1        # quality code
```

Fonte: Tom White

Weather dataset

Organização dos arquivos

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

Fonte: Tom White

Example 2-2. A program for finding the maximum recorded temperature by year from NCDC w records

```
#!/usr/bin/env bash
for year in all/*
do
  echo -ne `basename $year .gz`"\t"
  gunzip -c $year | \
    awk '{ temp = substr($0, 88, 5) + 0;
           q = substr($0, 93, 1);
           if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
         END { print max }'
done
```

Fonte: Tom White

Weather dataset

Como paralelizar?

- Múltiplas threads?
- Um computador por ano?
- Como atribuir trabalho igual para todos?
- Como juntar os resultados parcias?
- Como lidar com as falhas?

Weather dataset

How the data is represented in the actual file

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

How the lines in the file are presented to the map function by the framework

keys: Line offsets within the file

```
(0,   0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

The lines are presented to the map function as key-value pairs

Map function

- **Extract** year and temperature from each record and **emit** output

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

Weather dataset

The output from the map function

- Processed by the MapReduce framework *before* being sent to the reduce function
  - **Sort** and **group** <*key, value*> pairs by key
- In our example, each year appears with a list of all its temperature readings

```
(1949, [111, 78])
(1950, [0, 22, -11])
...
```

What about the reduce function?

- All it has to do now is iterate through the list supplied by the maps and pick the max reading
- Example output at the reducer?

```
(1949, 111)
(1950,  22)
...
```

Credit

Much of this information is from the Google Code University:

http://code.google.com/edu/parallel/mapreduce-tutorial.html

See also: http://hadoop.apache.org/common/docs/current/

for the Apache Hadoop version

Read this (the definitive paper):

http://labs.google.com/papers/mapreduce.html

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Background

- Traditional programming is serial

- Parallel programming
  - Break processing into parts that can be executed concurrently on multiple processors

- Challenge
  - Identify tasks that can run concurrently
    and/or groups of data that can be processed concurrently
  - Not all problems can be parallelized

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Simplest environment for parallel processing

- No dependency among data
- Data can be split into equal-size chunks - **shards**
- Each process can work on a chunk
- Master/worker approach
  - Master:
    - Initializes array and splits it according to # of workers
    - Sends each worker the sub-array
    - Receives the results from each worker
  - Worker:
    - Receives a sub-array from master
    - Performs processing
    - Sends results to master

 © 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

MapReduce

- Created by Google in 2004
  – Jeffrey Dean and Sanjay Ghemawat

- Inspired by LISP
  – Map(function, set of values)
    - Applies function to each value in the set

      (map ‘length ‘(() (a) (a b) (a b c))) ⇒ (0 1 2 3)

  – Reduce(function, set of values)
    - Combines all the values using a binary function (e.g., +)

      (reduce #'+ ‘(1 2 3 4 5)) ⇒ 15

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

MapReduce

- MapReduce
  - Framework for parallel computing
  - Programmers get simple API
  - Don't have to worry about handling
    - parallelization
    - data distribution
    - load balancing
    - fault tolerance

- Allows one to process huge amounts of data (terabytes and petabytes) on thousands of processors

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Who has it?

- Google
  - Original proprietary implementation
- Apache Hadoop MapReduce
  - Most common (open-source) implementation
  - Built to specs defined by Google
- Amazon Elastic MapReduce
  - Uses Hadoop MapReduce running on Amazon EC2

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

MapReduce

- Map: (input shard) → intermediate(key/value pairs)
  - Map calls are distributed across machines by automatically partitioning the input data into M "shards".
  - MapReduce library groups together all intermediate values associated with the same intermediate key & passes them to the *Reduce* function
- Reduce: intermediate(key/value pairs) → result files
  - Accepts an intermediate key & a set of values for the key
  - It merges these values together to form a smaller set of values
  - Reduce calls are distributed by partitioning the intermediate key space into R pieces using a partitioning function
    (e.g., *hash(key) mod R*).The user specifies the # of partitions (R) and the partitioning function.

 © 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

MapReduce

- Map
  - Grab the relevant data from the source
  - User function gets called for each chunk of input
  - Spits out (key, value) pairs

- Reduce
  - Aggregate the results
  - User function gets called for each unique key

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

MapReduce: what happens in between?

- Map
  - Grab the relevant data from the source (parse into key, value)
  - Write it to an intermediate file
- Partition
  - Partitioning: identify which of *R* reducers will handle which keys
  - Map partitions data to target it to one of *R* Reduce workers based on a partitioning function (both *R* and partitioning function user defined)

Map Worker

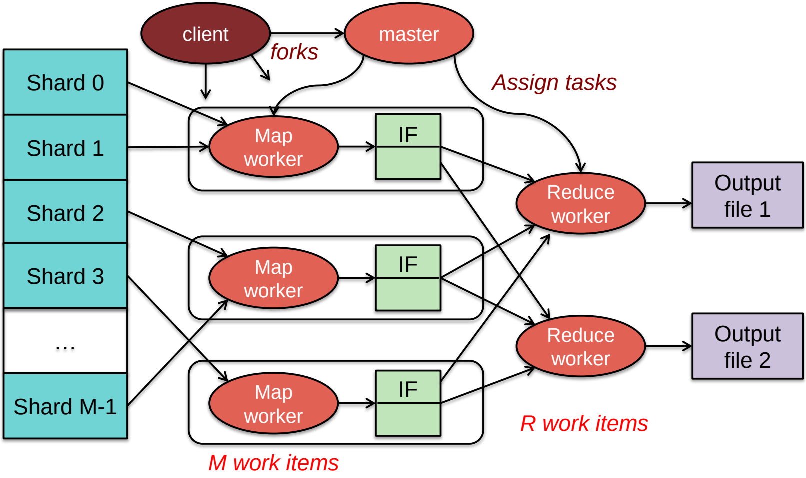
- Shuffle (Sort)
  - Fetch the relevant partition of the output from <u>all</u> mappers
  - Sort by keys (different mappers may have output the same key)
- Reduce
  - Input is the sorted output of mappers
  - Call the user *Reduce* function per key with the list of values for that key to aggregate the results

Reduce Worker

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

MapReduce: the complete picture

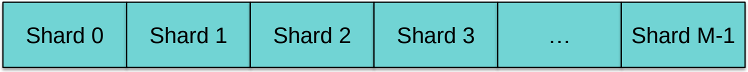


© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Step 1: Split input files into chunks (shards)

- Break up the input data into *M* pieces (typically 64 MB)



Input files

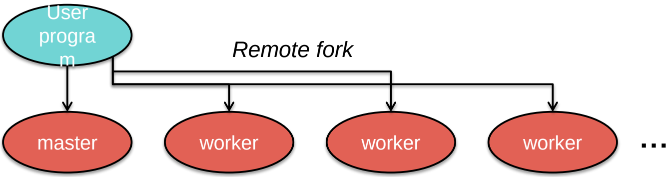
Divided into *M* shards

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Step 2: Fork processes

- Start up many copies of the program on a cluster of machines
  - 1 master: scheduler & coordinator
  - Lots of workers
- Idle workers are assigned either:
  - map tasks (each works on a shard) – there are *M* map tasks
  - reduce tasks (each works on intermediate files) – there are *R*
    - *R* = # partitions, defined by the user

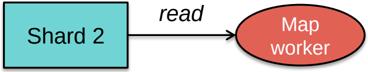


© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Step 3: Run Map Tasks

- Reads contents of the input shard assigned to it
- Parses key/value pairs out of the input data
- Passes each pair to a user-defined *map* function
  - Produces intermediate key/value pairs
  - These are buffered in memory

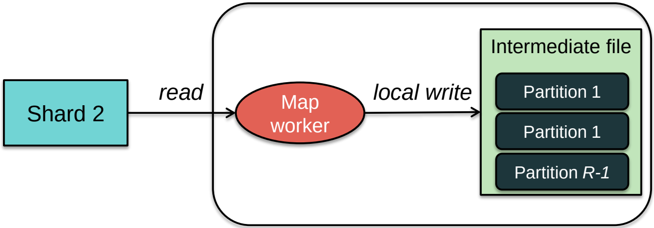


© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Step 4: Create intermediate files

- Intermediate key/value pairs produced by the user's *map* function buffered in memory and are periodically written to the local disk
  - Partitioned into *R* regions by a partitioning function



© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Step 4a. Partitioning

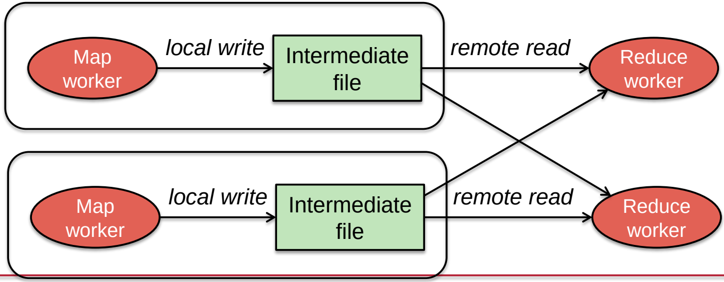
- Map data will be processed by Reduce workers
  - The user's *Reduce* function will be called once per unique key generated by *Map*.
- This means we will need to sort all the (key, value) data by keys and decide which Reduce worker processes which keys – the Reduce worker will do this
- **Partition function**: decides which of *R* reduce workers will work on which key
  - Default function: *hash(key) mod R*
  - Map worker partitions the data by keys
- Each Reduce worker will read their partition from every Map worker

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Step 5: Reduce Task: sorting

- Reduce worker gets notified by the master about the location of intermediate files for its partition
- Uses RPCs to read the data from the local disks of the map workers
- When the *reduce* worker reads intermediate data for its partition
  - It sorts the data by the intermediate keys
  - All occurrences of the same key are grouped together

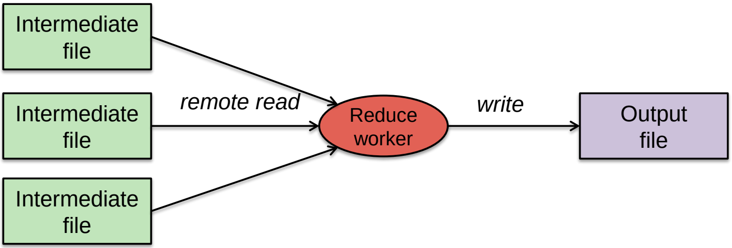


© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Step 6: Reduce Task: *Reduce*

- The sort phase grouped data with a unique intermediate key
- User's *Reduce* function is given the key and the set of intermediate values for that key
  - < key, (value1, value2, value3, value4, …) >
- The output of the *Reduce* function is appended to an output file



© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Step 7: Return to user

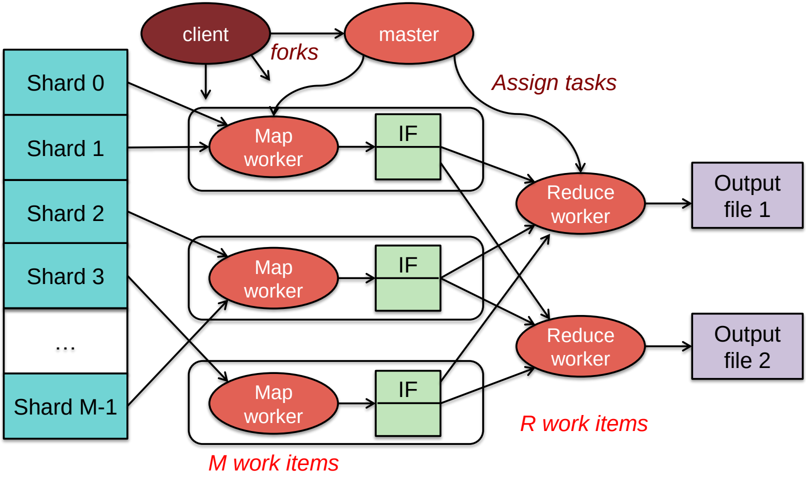
- When all *map* and *reduce* tasks have completed, the master wakes up the user program

- The *MapReduce* call in the user program returns and the program can resume execution.
  - Output of *MapReduce* is available in *R* output files

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

MapReduce: the complete picture



 © 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Example

- Count # occurrences of each word in a collection of documents
- Map:
  - Parse data; output each word and a count (1)
- Reduce:
  - Sort: sort by keys (words)
  - Reduce: Sum together counts each key (word)

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

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));
```


© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Locality

- Input and Output files are on GFS (Google File System)
- MapReduce runs on GFS chunkservers
- Master tries to schedule *map* worker on one of the machines that has a copy of the input chunk it needs.

Fonte: Prof. Paul Krzyzanowski

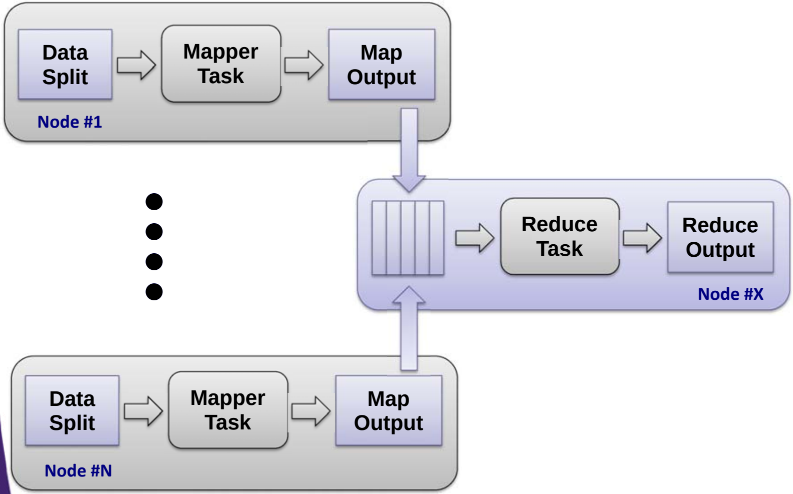
MapReduce

- **Divided in two phases**
  - Map phase
  - Reduce phase
- **Both phases use key-value pairs as input and output**
- **The implementer provides map and reduce functions**
- **MapReduce framework orchestrates splitting, and distributing of Map and Reduce phases**
  - Most of the pieces can be easily overridden

MapReduce

- **Job – execution of map and reduce functions to accomplish a task**
  - Equal to Java's main
- **Task – single Mapper or Reducer**
  - Performs work on a fragment of data

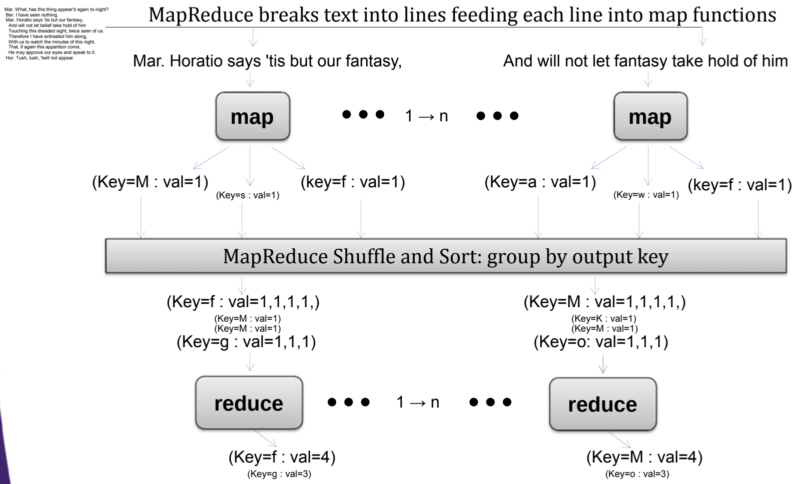
Map Reduce Flow of Data



First Map Reduce Job

- **<u>StartsWithCount</u> Job**
  - Input is a body of text from HDFS
    - In this case hamlet.txt
  - Split text into tokens
  - For each first letter sum up all occurrences
  - Output to HDFS

Word Count Job



StartsWithCount Job

1. **Configure the Job**
   – Specify Input, Output, Mapper, Reducer and Combiner
2. **Implement Mapper**
   – Input is text – a line from hamlet.txt
   – Tokenize the text and emit first character with a count of 1 - <token, 1>
3. **Implement Reducer**
   – Sum up counts for each letter
   – Write out the result to HDFS
4. **Run the job**

Other Examples

- Distributed grep (search for words)
  - *Search for words in lots of documents*
  - Map: emit a line if it matches a given pattern
  - Reduce: just copy the intermediate data to the output

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Other Examples

- Count URL access frequency
  - *Find the frequency of each URL in web logs*
  - Map: process logs of web page access; output <URL, 1>
  - Reduce: add all values for the same URL

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Other Examples

- Reverse web-link graph
  - *Find where page links come from*
  - Map: output <target, source>for each link to *target* in a page *source*

  - Reduce: concatenate the list of all source URLs associated with a target.

    Output <target, list(source)>

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Other Examples

- Inverted index
  - *Find what documents contain a specific word*
  - Map: parse document, emit <word, document-ID> pairs
  - Reduce: for each word, sort the corresponding document IDs

    Emit a <word, list(document-ID)> pair
    The set of all output pairs is an inverted index

 © 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

MapReduce Summary

- Get a lot of data
- Map
  - Parse & extract items of interest
- Sort (shuffle) & partition
- Reduce
  - Aggregate results
- Write to output files

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Fault tolerance

- Master pings each worker periodically
  - If no response is received within a certain time, the worker is marked as *failed*
  - *Map* or *reduce* tasks given to this worker are reset back to the initial state and rescheduled for other workers.

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

Locality

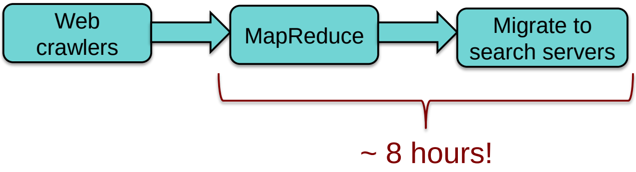
- Input and Output files are on GFS (Google File System)
- MapReduce runs on GFS chunkservers
- Master tries to schedule *map* worker on one of the machines that has a copy of the input chunk it needs.

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

All is not perfect

- MapReduce was used to process webpage data collected by Google's crawlers.
  - It would extract the links and metadata needed to search the pages
  - Determine the site's PageRank
- The process took around eight hours.
  - Results were moved to search servers.
  - This was done continuously.



Fonte: Prof. Paul Krzyzanowski

All is not perfect

- Web has become more dynamic
  - an 8+ hour delay is a lot for some sites
- Goal: refresh certain pages within seconds
- MapReduce
  - Batch-oriented
  - Not suited for near-real-time processes
  - Cannot start a new phase until the previous has completed
    - Reduce cannot start until all Map workers have completed
  - Suffers from “stragglers” – workers that take too long (or fail)
  - This was done continuously
- MapReduce is still used for many Google services
- Search framework updated in 2009-2010: Caffeine
  - Index updated by making direct changes to data stored in BigTable
  - Data resides in Colossus (GFS2) instead of GFS

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

In Practice

- Most data not simple files
  - B-trees, tables, SQL databases, memory-mapped key-values
- Hardly ever use textual data: slow & hard to parse
  - Most I/O encoded with Protocol Buffers

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski

More info

- Good tutorial presentation & examples at:
  http://research.google.com/pubs/pub36249.html

- The definitive paper:
  http://labs.google.com/papers/mapreduce.html

© 2014 Paul Krzyzanowski

Fonte: Prof. Paul Krzyzanowski