Università degli Studi di Roma "Tor Vergata"
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Elective exercise using Go and RPC

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2018/19

Valeria Cardellini

Elective exercise using Go and RPC

- Solve one of the following problems using the MapReduce paradigm:
  - **WordCount**: it should return the counts of each word in a collection of documents (assume either N files or a large file divided into N chunks)
  - **WordLengthCount**: it should return how many words of certain lengths exist in a collection of documents
- Requirements: use Go and RPC
- 1 or 2 students per group

A brief introduction to MapReduce

Parallel programming: background

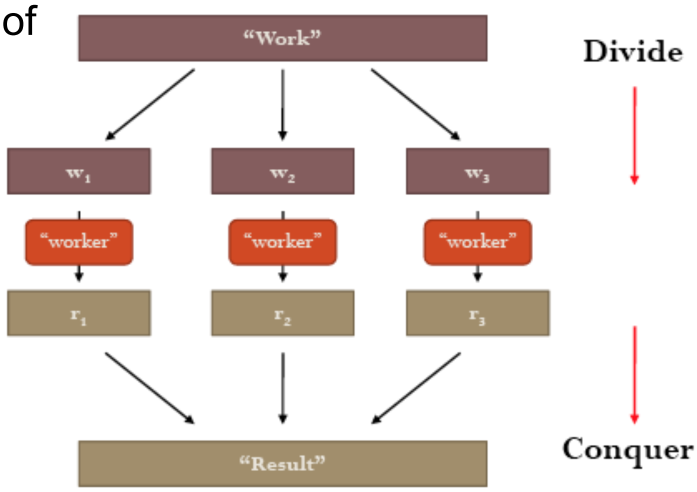
- Parallel programming
  - Break processing into parts that can be executed concurrently on multiple workers
    - Workers can be:
      - Threads in a processor core
      - Cores in a multi-core processor
      - Multiple processors in a machine
      - Many machines in a cluster
- Challenge
  - Identify tasks that can run concurrently and/or groups of data that can be processed concurrently

Parallel programming: background

- Which is the simplest environment for parallel programming?
  - No dependency among data
    - Data can be split into equal-size chunks
  - Each worker can work on a chunk
  - Master/worker approach
    - Master
      - Initializes array and splits it according to the number 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
- Single Program, Multiple Data (**SPMD**): technique to achieve parallelism
  - The most common style of parallel programming

Key idea behind MapReduce: Divide and conquer

- A feasible approach to tackling large-data problems
  - Partition a large problem into smaller sub-problems
  - Independent sub-problems executed in parallel
  - Combine intermediate results from each individual worker
- Implementation details of divide and conquer are complex



Divide and conquer: how?

- Decompose the original problem into smaller, parallel tasks
- Schedule tasks on workers distributed in a cluster, keeping into account:
  - Data locality
  - Resource availability
- Ensure workers get the data they need
- Coordinate synchronization among workers
- Share partial results
- Handle failures

Key idea behind MapReduce: Scale out, not up!

- For data-intensive workloads, a large number of commodity servers is preferred over a small number of high-end servers
  - Cost of super-computers is not linear
  - Datacenter efficiency is a difficult problem to solve, but recent improvements
- Processing data is quick, I/O is very slow
- Sharing vs. shared nothing:
  - Sharing: manage a common/global state
  - Shared nothing: independent entities, no common state
- Sharing is difficult:
  - Synchronization, deadlocks
  - Finite bandwidth to access data from SAN
  - Temporal dependencies are complicated (restarts)

MapReduce

- **Programming model** for processing huge amounts of data sets over thousands of servers
  - Originally proposed by Google in 2004: “MapReduce: simplified data processing on large clusters” http://bit.ly/2iq7jlY
  - Based on a shared nothing approach
- Also an associated **implementation** (framework) of the distributed system that runs the corresponding programs
- Some examples of applications for Google:
  - Web indexing
  - Reverse Web-link graph
  - Distributed sort
  - Web access statistics

Typical Big Data problem

Map

- Iterate over a large number of records
- Extract something of interest from each record
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Reduce

Key idea: provide a functional abstraction of the two Map and Reduce operations

MapReduce: model

- Processing occurs in two phases: **Map** and **Reduce**
  - Functional programming roots (e.g., Lisp)
- Map and Reduce are defined by the programmer
- Input and output: sets of key-value pairs
- Programmers specify two functions: map and reduce
- **map**$(k_1, v_1) \rightarrow [(k_2, v_2)]$
- **reduce**$(k_2, [v_2]) \rightarrow [(k_3, v_3)]$
  - (k, v) denotes a (key, value) pair
  - […] denotes a list
  - Keys do not have to be unique: different pairs can have the same key
  - Normally the keys k_1 of input elements are not relevant

Map

- Execute a function on a set of key-value pairs (input shard) to create a new list of key-value pairs

  map (in_key, in_value) → list(out_key, intermediate_value)

- Map calls are distributed across machines by automatically partitioning the input data into *shards*
  - Parallelism is achieved as keys can be processed by different machines
- MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function

Reduce

- Combine values in sets to create a new value

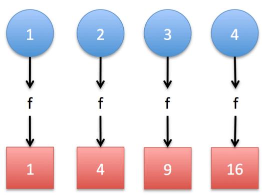
  reduce (out_key, list(intermediate_value)) → list(out_key, out_value)

  - The key in output is often identical to the key in the input
  - Parallelism is achieved as reducers operating on different keys can be executed simultaneously

Your first MapReduce example (in Lisp)

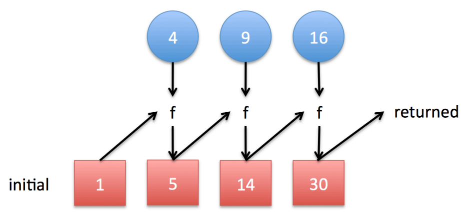
- *Example*: sum-of-squares (sum the square of numbers from 1 to n) in MapReduce fashion
- Map function:

  map square [1,2,3,4]
  returns [1,4,9,16]



- Reduce function:

  reduce [1,4,9,16]
  returns 30 (the sum of the square elements)

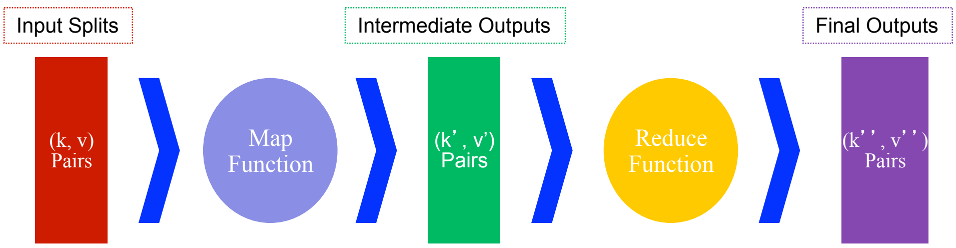


MapReduce computation

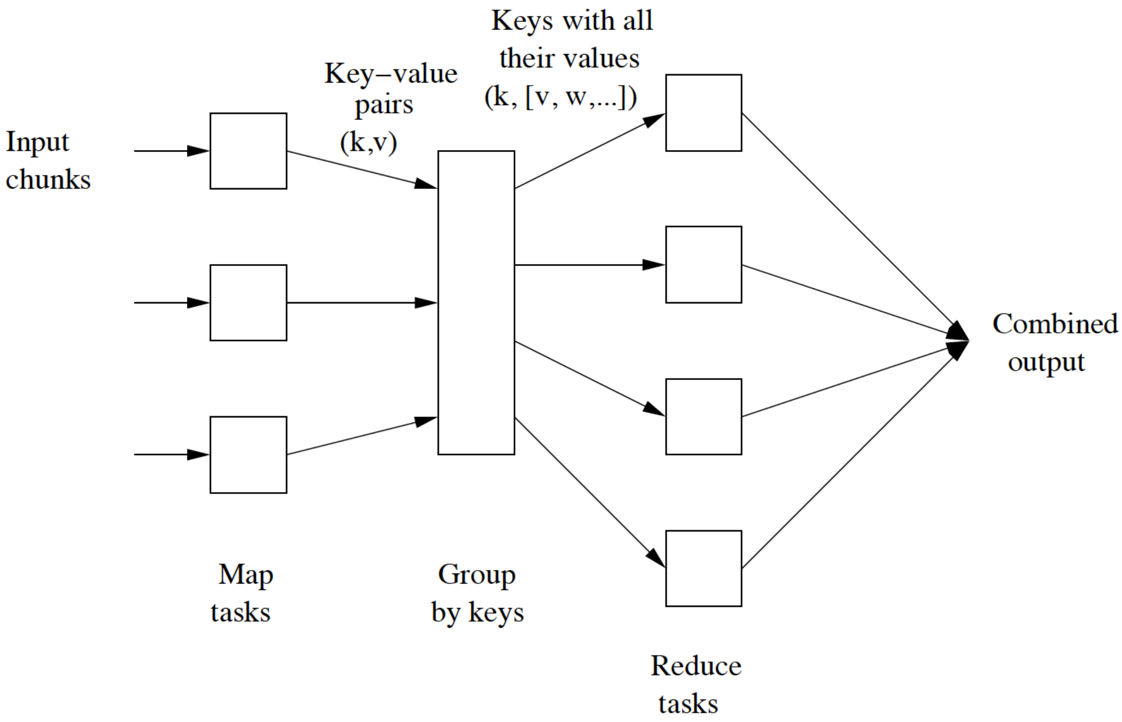
1. Some number of Map tasks each are given one or more chunks of data from a distributed file system
2. These Map tasks turn the chunk into a sequence of key-value pairs
   – The way key-value pairs are produced from the input data is determined by the code written by the user for the Map function
3. The key-value pairs from each Map task are collected by a master controller and sorted by key
4. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task
5. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way
   – The manner of combination of values is determined by the code written by the user for the Reduce function
6. Output key-value pairs from each reducer are written persistently back onto the distributed file system
7. The output ends up in r files, where r is the number of reducers
   – Such output may be the input to a subsequent MapReduce phase

Where the magic happens

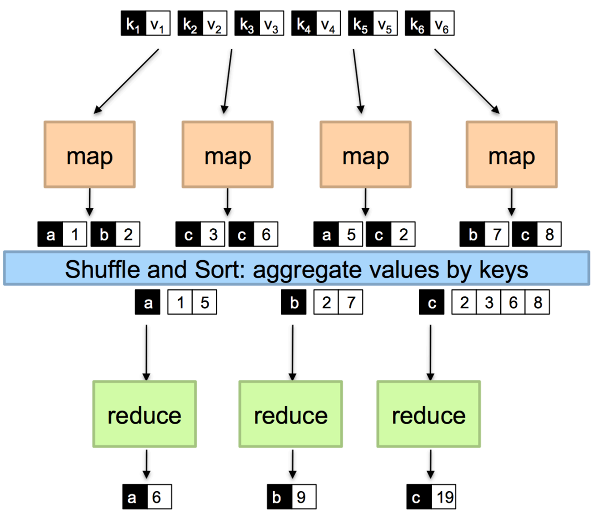
- Implicit between the map and reduce phases is a distributed "group by" operation on intermediate keys, called **shuffle and sort**
  – Transfer mappers output to reducers, merging and sorting the output
  – Intermediate data arrive at every reducer sorted by key
- Intermediate keys are transient
  – They are not stored on the distributed file system
  – They are "spilled" to the local disk of each machine in the cluster



MapReduce computation: the complete picture



A simplified view of MapReduce: example



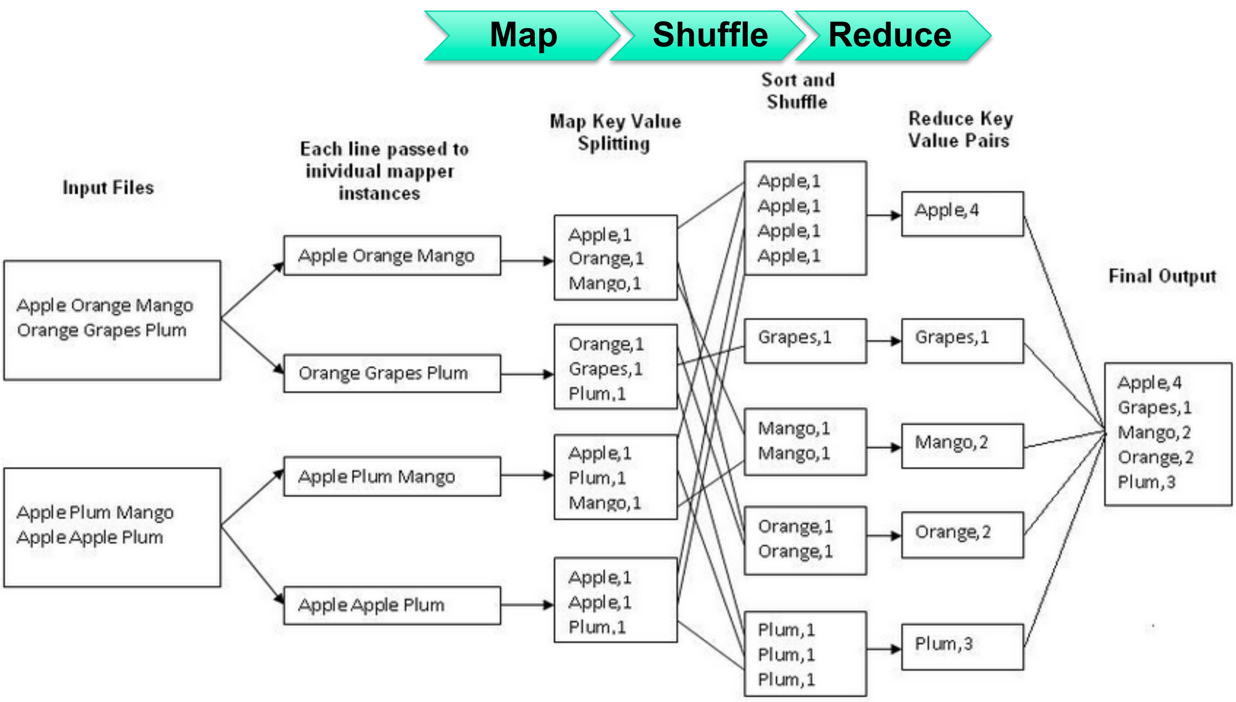
- Mappers are applied to all input key-value pairs, to generate an arbitrary number of intermediate pairs
- Reducers are applied to all intermediate values associated with the same intermediate key
- Between the map and reduce phase lies a barrier that involves a large distributed sort and group by

WordCount

- **Problem**: count the number of occurrences for each word in a large collection of documents
- **Input**: repository of documents, each document is an element
- **Map**: read a document and emit a sequence of key-value pairs where:
  - Keys are words of the documents and values are equal to 1:

    $(w_1, 1), (w_2, 1), \ldots , (w_n, 1)$
- **Shuffle and sort**: group by key and generates pairs of the form $(w_1, [1, 1, \ldots , 1]) , \ldots , (w_n, [1, 1, \ldots , 1])$
- **Reduce**: add up all the values and emits $(w_1, k) ,\ldots, (w_n, l)$
- **Output**: (w,m) pairs where:
  - w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents

WordCount in practice



WordLengthCount

- **Problem**: count how many words of certain lengths exist in a collection of documents
- **Input**: a repository of documents, each document is an element
- **Map**: read a document and emit a sequence of key-value pairs where the key is the length of a word and the value is the word itself:

  (i, w1), … , (j, wn)
- **Shuffle and sort**: group by key and generate pairs of the form

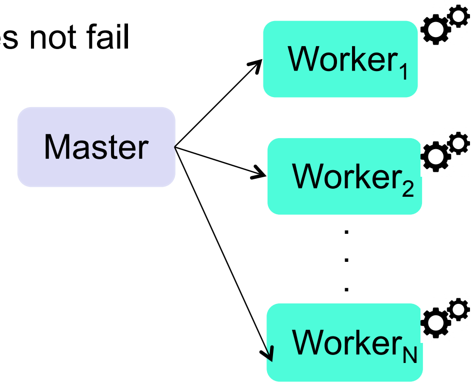
  (1, [w1, … , wk]) , … , (n, [wr, … , ws])
- **Reduce**: count the number of words in each list and emit:

  (1, l) , … , (p, m)
- **Output**: (l,n) pairs, where l is a length and n is the total number of words of length l in the input documents

Back to the exercise

Overview on architecture

- Exploit master-worker architecture
  - Distribute the work among workers using channels via RPC
- Need to implement a master that assigns map and reduce tasks to workers
- Do not consider failures of master and workers
  - The set of workers is known and does not change during the computation; no worker fails
  - The master does not fail



Overview on architecture

- 3 phases
  - **Map**
  - **Shuffle and sort**
  - **Reduce**
- Distribute the work among parallel workers
- Need a synchronization point (i.e., barrier) between map and reduce phases
  - No reduce task can start until all the map tasks have finished their processing
- Need a synchronization point after reduce phase
  - The master must wait all the reducetasks before merging their results

Main ideas

- **Map phase**: process the N files/chunks in parallel on the workers, applying the map function to each file/chunk
  - The master should assign the N files/chunks to the workers that execute the map task
  - Each map task can either write its results to some number of intermediate files or send its results to the master or the reduce tasks
    - You can choose to realize the shuffle and sort phase either in a centralized or decentralized way

Main ideas

- **Shuffle and sort phase**: organize the output of the map tasks in such a way that the reduce tasks receive in input data grouped by key

- **Reduce phase**: each reduce task processes its input and writes its output to a file or send its output to the master
  - The master merges all outputs from the reduce tasks and produces the final result

Delivery

- When to deliver
  - By January 18, 2019
- What to deliver
  - Your code, including the instructions to run it
  - Optional: short report describing the application architecture and main ideas
- How to deliver
  - By email
  - Use as mail subject: [SDCC] consegna esercizio