TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

# Elective exercise using Go and RPC

**Corso di Sistemi Distribuiti e Cloud Computing**
A.A. 2021/22

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

# Elective exercise using Go and RPC

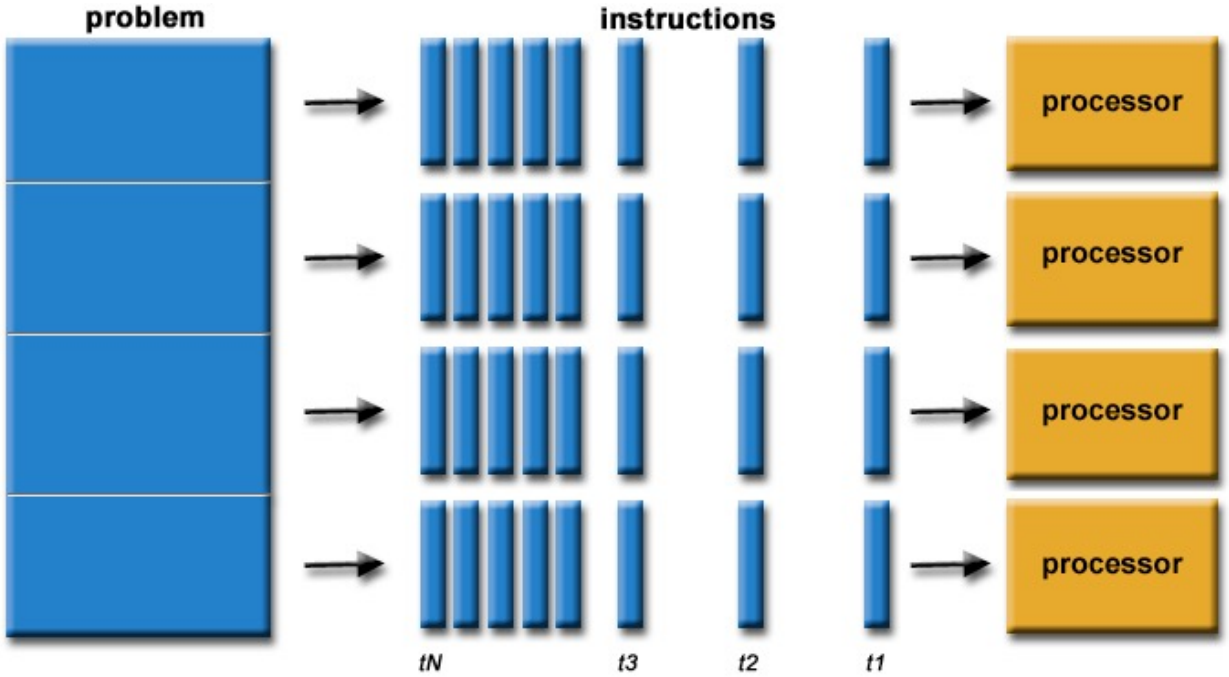
- Realize a distributed grep using the MapReduce paradigm:
  - **DistGrep** returns the lines of text of a large input file given in input that match a specific pattern (i.e., regular expression) specified

- Requirements: use Go and RPC (or gRPC)
- 1 or 2 students per group

# A brief introduction to MapReduce

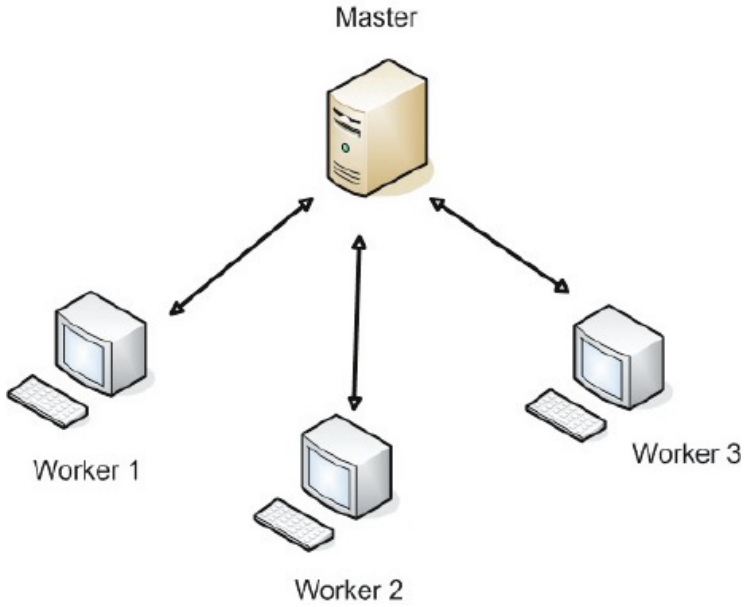
# Parallel programming: background

- Parallel programming
  - Simultaneous use of multiple computing resources (e.g., threads, cores, processors, machines) to solve a problem
  - How? Break processing into parts that can be executed concurrently on multiple computing resources



# Parallel programming: background

- Simplest environment for parallel programming: **master/worker** architecture
  - Master
    - Gets data and splits it into chunks according to the number of workers
    - Sends each worker equal number of chunks
    - Receives the results from each worker
  - Workers
    - Receive some chunks of data from master
    - Perform processing
    - Send back results to master

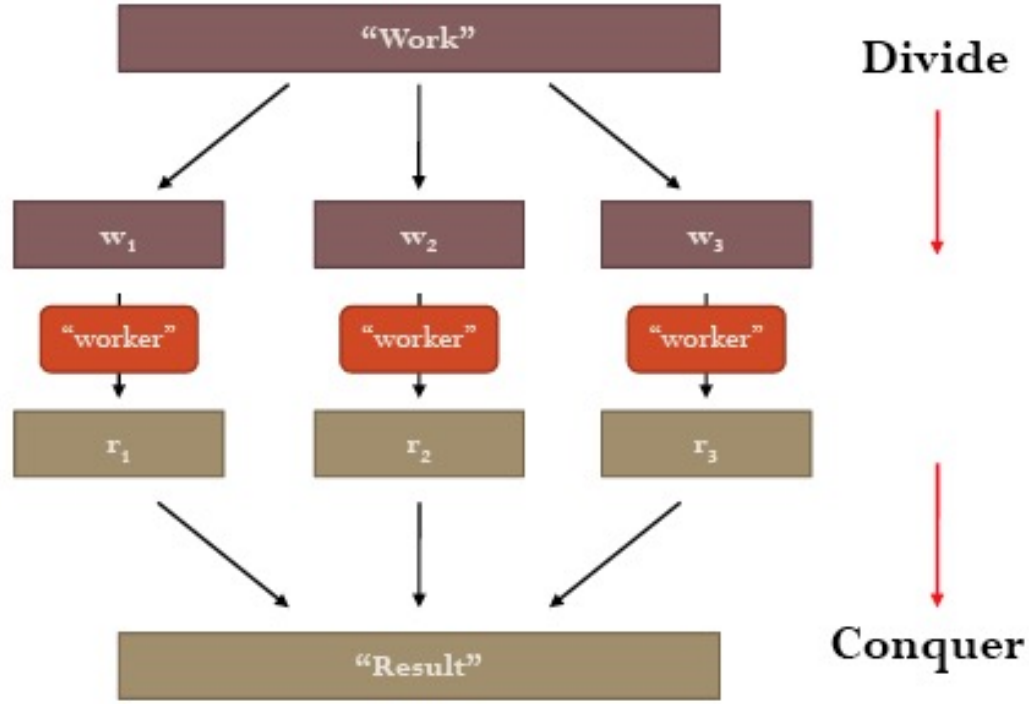


# Parallel programming: background

- Several styles of parallel programming
- Single Program, Multiple Data (**SPMD**) is the most commonly used
  - **Single Program**: all computing resources execute the same program simultaneously
  - **Multiple Data**: all computing resources may use different data

# Key idea behind MapReduce and Spark: Divide et impera (aka divide and conquer)

- Feasible approach to tackle Big data problems
  - Partition a large problem into smaller sub-problems
  - Solve independent sub-problems in parallel
  - Combine intermediate results from each individual worker
- Implementation details are complex
  - But are mostly managed by Big Data processing frameworks!



# Divide et impera: how?

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

# Other key ideas behind MapReduce and Spark: Scale out + Shared nothing

- **Scale out**: a large number of commodity servers is preferred over a small number of high-end servers
  - Cost of commodity servers is linear

- **Shared nothing** is preferable over sharing
  - Shared nothing: each node is completely independent of other nodes in the system, no shared memory or shared disks
    - Pros: scalability and fault tolerance
  - Sharing: nodes share some common/global state
    - Cons: requires synchronization, deadlocks can occur, shared resources can become bottlenecks

# 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

- Also an associated **implementation** (framework) of the distributed system that runs the corresponding programs

- Examples of applications at Google:
  - Web indexing
  - Reverse Web-link graph
  - Distributed sort
  - Web access statistics

# Typical Big Data problem

- 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

Map

Reduce

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

# MapReduce: model

- Processing occurs in two phases: **Map** and **Reduce**
- Input and output: sets of key-value pairs
- Programmer specifies map and reduce functions
- **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
  - Keys of input elements are usually not relevant

# Map

- Execute a function on a set of key-value pairs to create a new list of key-value pairs

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

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

# Reduce

- Combine values in sets to create a new value

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

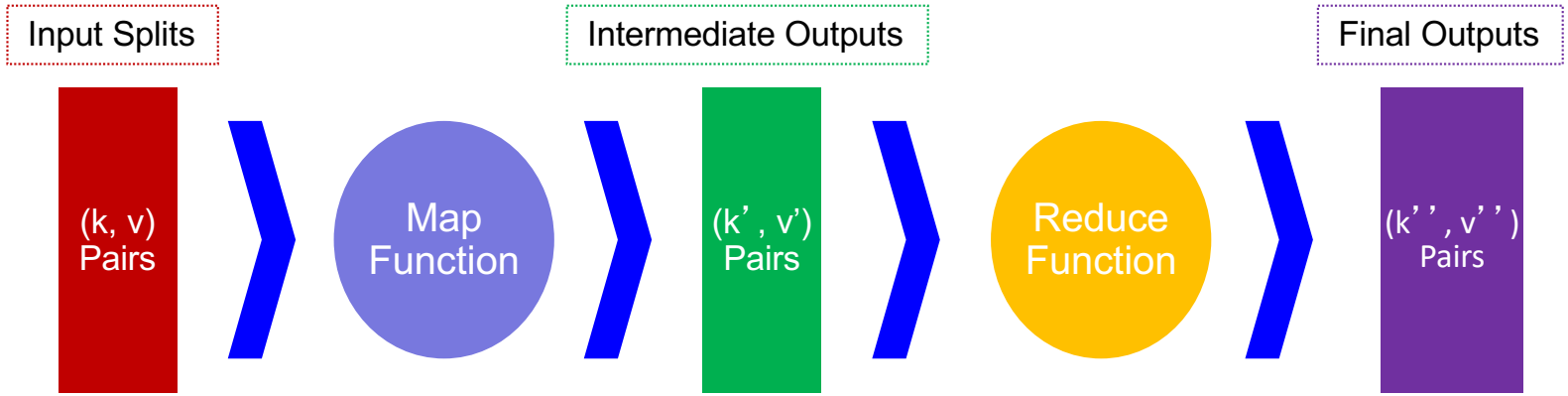
- Reduce output key is often identical to its input key
- Parallelism is achieved as Reduce functions operating on different keys can be executed simultaneously

# MapReduce computation

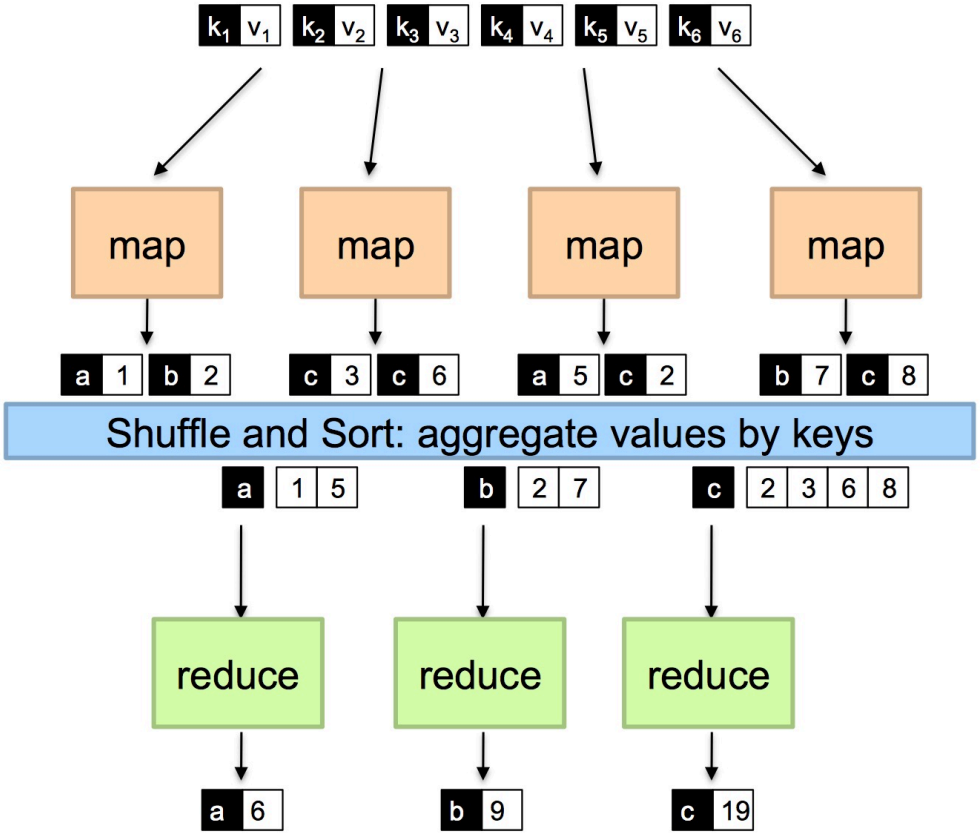
1. Some number of Map tasks each are given one or more data blocks stored in a distributed file system
2. Map tasks turn the chunk into a sequence of key-value pairs
   - The way key-value pairs are produced from the input data is determined by Map function
3. Key-value pairs from each Map task are grouped by and sorted by key
4. Keys are divided among Reduce tasks, so that all key-value pairs with the same key wind up at the same Reduce task
5. Reduce tasks work on one key at a time, and combine all the values associated with that key in some way
   - The way values are combined is determined by Reduce function
6. Output key-value pairs from each reducer are written 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

# Shuffle and sort

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



# A simplified view of MapReduce



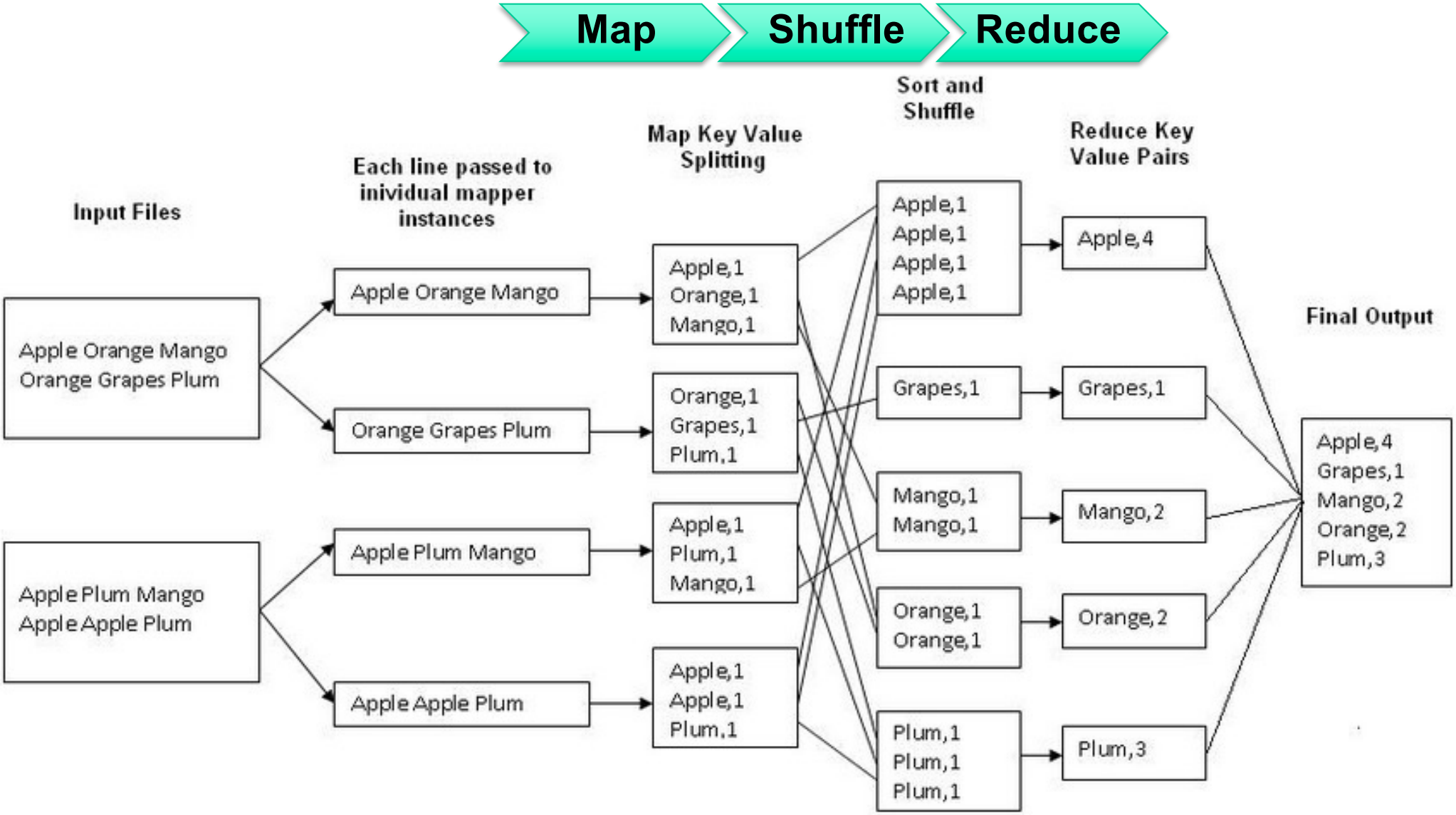

- Mappers are applied to all input key-value pairs, to generate an arbitrary number of intermediate pairs
- Reducers are applied to all intermediate values associated with the same intermediate key
- Between map and reduce phases lies a barrier that involves a large distributed sort and group by

# “Hello World” in MapReduce: 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:

    (w1, 1), (w2, 1), … , (wn, 1)
- **Shuffle and sort**: group by key and generates pairs of the form (w1, [1, 1, … , 1]) , . . . , (wn, [1, 1, … , 1])
- **Reduce**: add up all the values and emits (w1, k) ,…, (wn, 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

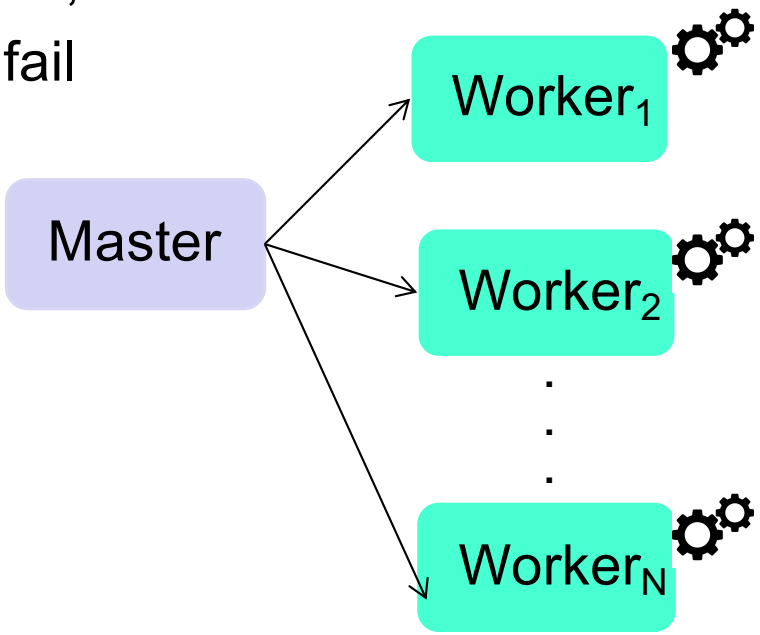
# WordCount in practice



# Back to our exercise

# Architecture overview

- Exploit master-worker architecture
  - Distribute work among workers
  - Use RPC for client-master and master-workers communication
- Implement a master that assigns map and reduce tasks to workers (let's assume N workers)
- Do not consider failures of master and workers
  - Assume that set of workers is known and does not change during computation; no worker fails
  - Master does not fail



# Architecture overview

- 3 phases
  - **Map**
  - **Shuffle and sort**
  - **Reduce**
- Master distributes 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
  - Master must wait all the reduce tasks before merging their results

# Main ideas

- **Map phase**: process the N files/blocks in parallel on workers, applying the map function (i.e., local grep) to each file/block
  - Master assigns the N files/blocks to workers, that execute the map task
  - To implement the local grep, you can use the package `strings` and the functions it provides
  - Each map task can either write its results to (some number of) intermediate file(s) 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 output of map tasks in such a way that input data received by reduce tasks is grouped by key

- **Reduce phase**: each reduce task processes its input and sends it to the master
  - In our case, the reduce phase uses the identity function
  - Master merges all outputs from reduce tasks and produces final result

# Delivery

- When
  - By December 20, 2021
- What
  - Your code, including instructions to run it
  - Optional: very short report describing the application architecture and main ideas
- How
  - By email
  - Use as mail subject: [SDCC] consegna esercizio