MAP REDUCE (INTRODUCTION)

Master Program in Computer Science
University of Calabria

Prof. F. Ricca

Origin and motivations

- Google needed to process large amounts of raw data, such as crawled documents, web request logs, etc.
  - Computations are conceptually straightforward
  - But computations have to be distributed across hundreds or thousands of machines
  - How to parallelize the computation, distribute the data, and handle failures without large amounts of complex code?
- Google designed a new abstraction
  - To express the simple computations
  - To hide the messy details of parallelization, fault-tolerance, data distribution and load balancing
  - In-spired by the map and reduce primitives of functional languages
  - To achieve high performance on large clusters of commodity PCs

Map Reduce (1)

- Main Goal:

  Make programmers without any experience with parallel and distributed systems able to easily utilize the resources of a large distributed system

- MapReduce is
  - a programming model for processing and generating large data sets
  - a run-time system that takes care of
    - partitioning the input data
    - scheduling the program's execution in the cluster
    - handling machine failures
    - managing the required inter-machine communication

Map Reduce (2)

- The result:

  A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that runs on commodity PCs

- Check out the original paper:
  - http://research.google.com/arcHive/mapreduce.html

Main Components

- Two functions

  - *map (k1,v1) → list(k2,v2)*
    - **Input:** a key/value pairs
    - **Output:** a set of intermediate key/value pairs

  - *reduce* (k2,list(v2)) → list(v2)
    - **Input:** intermediate values having the same intermediate key
    - **Output:** a set of values*

- All intermediate values with the same key are passed to the *reduce* function for final aggregation

*can be key/value pairs

Programming Model

- The Process in brief
  - Given a set of input key/value pairs
  - Produce a set of output key/value pairs
  - The user provides two functions: *Map* and *Reduce*

1. Map (preprocesses)
   - Writes intermediate result in local machines
2. The library groups together all intermediate values associated with the same intermediate key
   - Intermediate values supplied to reduce via an iterator to handle lists of values that are too large to fit in memory
3. Reduce (merge, aggregate)
   - Writes the final result on GFS

Classic Example (Word count)

```
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));
```

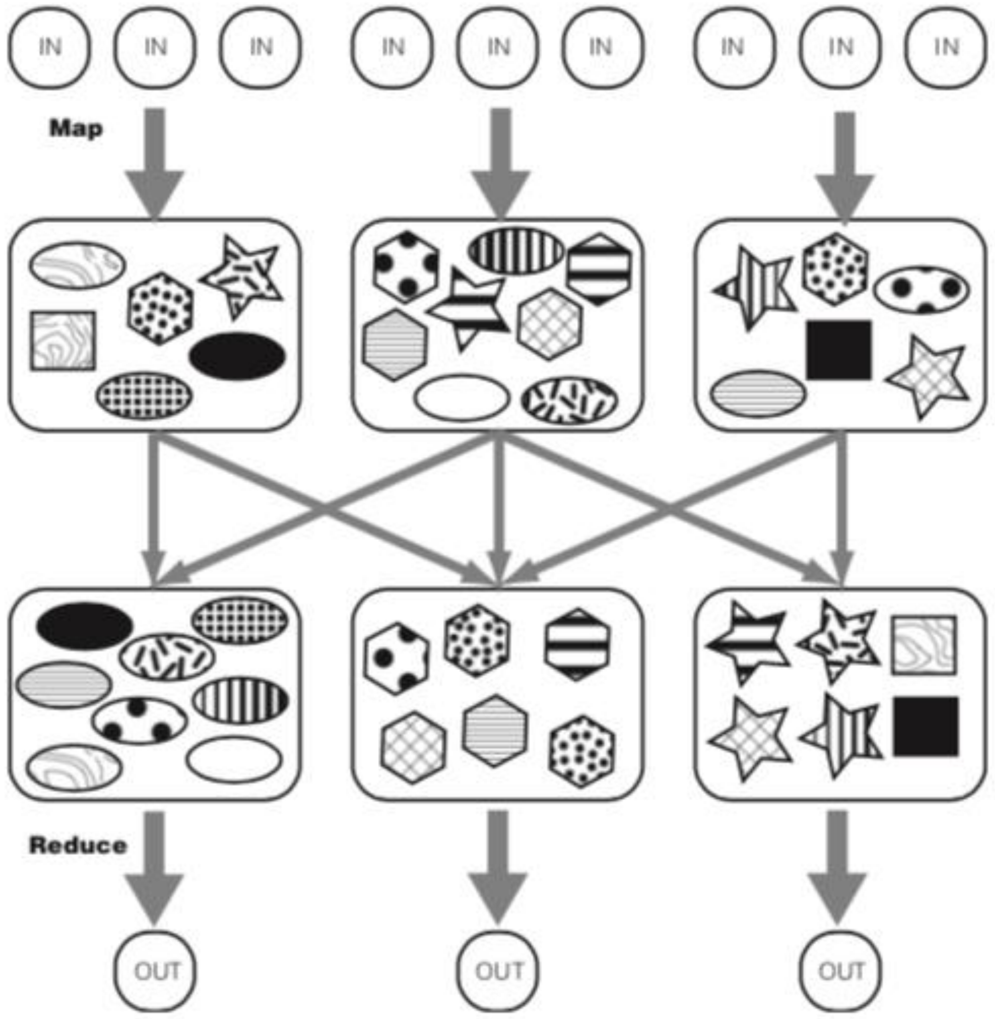

Figure 3.2 The MapReduce data flow, with an emphasis on partitioning and shuffling. Each icon is a key/value pair. The shapes represents keys, whereas the inner patterns represent values. After shuffling, all icons of the same shape (key) are in the same reducer. Different keys can go to the same reducer, as seen in the rightmost reducer. The partitioner decides which key goes where. Note that the leftmost reducer has more load due to more data under the "ellipse" key.

Execution on a Cluster (1)

- Google's original setting

  (1) Machines dual-processor x86 running Linux, with 2-4 GB of memory

  (2) Commodity networking hardware 100 M/s / 1GB/s

  (3) Hundreds or thousands of machines (*machine failures are common*)

  (4) Inexpensive IDE disks + a distributed file system GFS

  (5) Users submit jobs to a scheduling system,
  jobs divided in tasks mapped by the scheduler to available machines

Execution on a Cluster (2)

- The Map invocations distributed
  - Input data partitioned into a set of *M splits*
  - Splits can be processed in parallel by different machines

- Reduce invocations distributed
  - The key space (hash-)partitioned into *R pieces* (e.g., hash(key) mod R)
  - R and the partitioning function are specified by the user

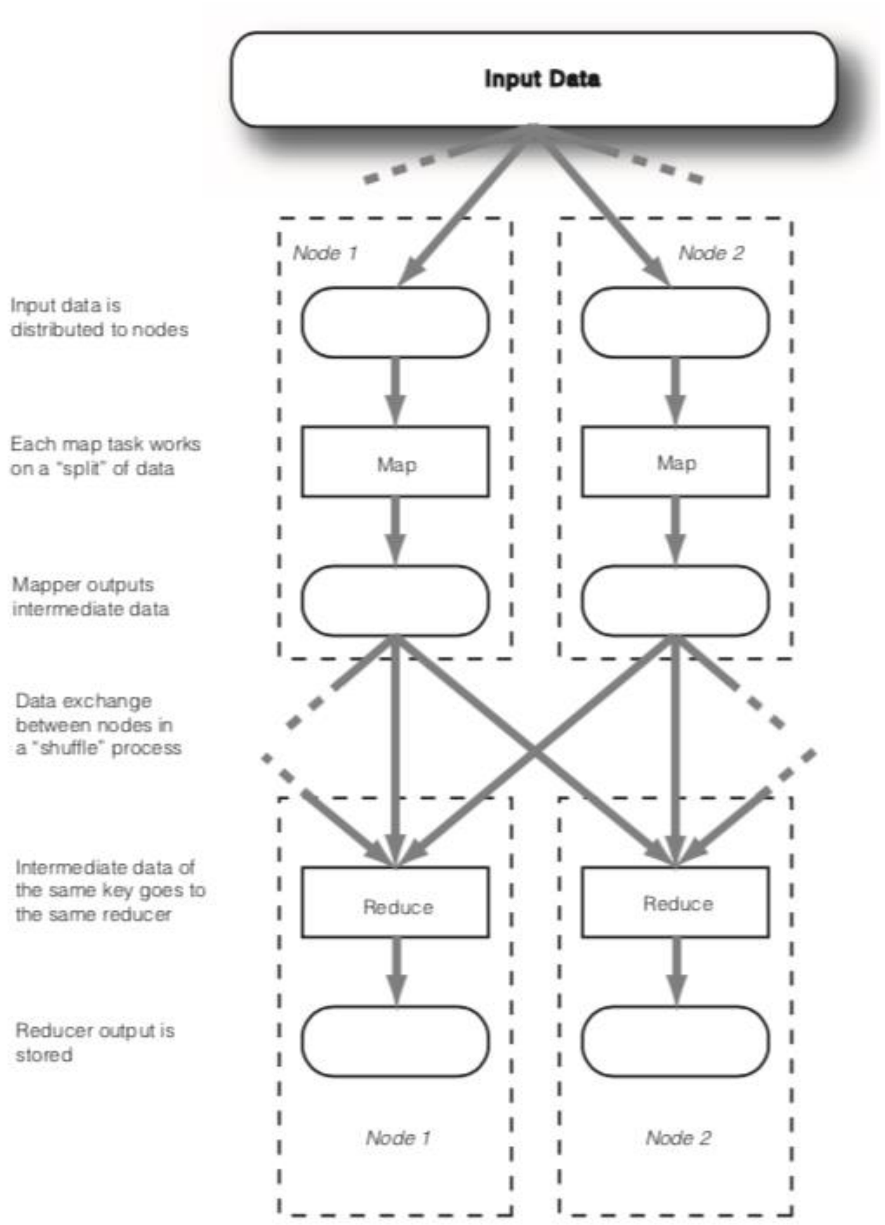
Input Data
Node 1
Node 2
Input data is distributed to nodes
Each map task works on a "split" of data
Map
Map
Mapper outputs intermediate data
Data exchange between nodes in a "shuffle" process
Intermediate data of the same key goes to the same reducer
Reduce
Reduce
Reducer output is stored
Node 1
Node 2

Figure 3.1 The general MapReduce data flow. Note that after distributing input data to different nodes, the only time nodes communicate with each other is at the "shuffle" step. This restriction on communication greatly helps scalability.

Data types

- Map
  - The input keys and values are drawn from a different domain than the output keys and values

- Reduce
  - The intermediate keys and values are from the same domain as the output keys and values

- Keys and Values can be of different types
  - Must be "serializable"
  - Keys must be "comparable"

Overall flow (1)

- The library first splits the input files into M pieces
  - From 16 megabytes to 64 MB each
  - Starts up many copies of the program on a cluster of machines
- One of the copies of the program is the master
  - Workers that are assigned work by the master
  - There are M map tasks and R reduce tasks to assign
- A worker who is assigned a map task reads the contents of the corresponding input split
  - It parses key/value pairs
  - Calls the user-defined Map function
  - The intermediate key/value pairs are buffered in memory

Overall flow (2)

- Periodically, the buffered pairs are written to local disk
  - Partitioned into R regions by the partitioning function
  - The master is informed of the buffered pairs locations
  - The master forwards these locations to the reduce workers

- A reduce worker is notified by the master about locations
  - It uses remote procedure calls to read the buffered data from the local disks of the workers
  - It sorts it by the intermediate keys so that all occurrences of the same key are grouped together
    (needed because many different keys map to the same reduce task)

Overall flow (3)

- The reduce worker iterates over the sorted intermediate data
  - Calls reduce function for each unique intermediate key encountered
  - The output of the Reduce function is appended to a final output file for this reduce partition

- When all map tasks and reduce tasks have been completed
  - The master wakes up the user program
  - The program returns back to the user code

- The output is available in the R output files (one per reduce task)
  - Not need to combine these R output files into one file

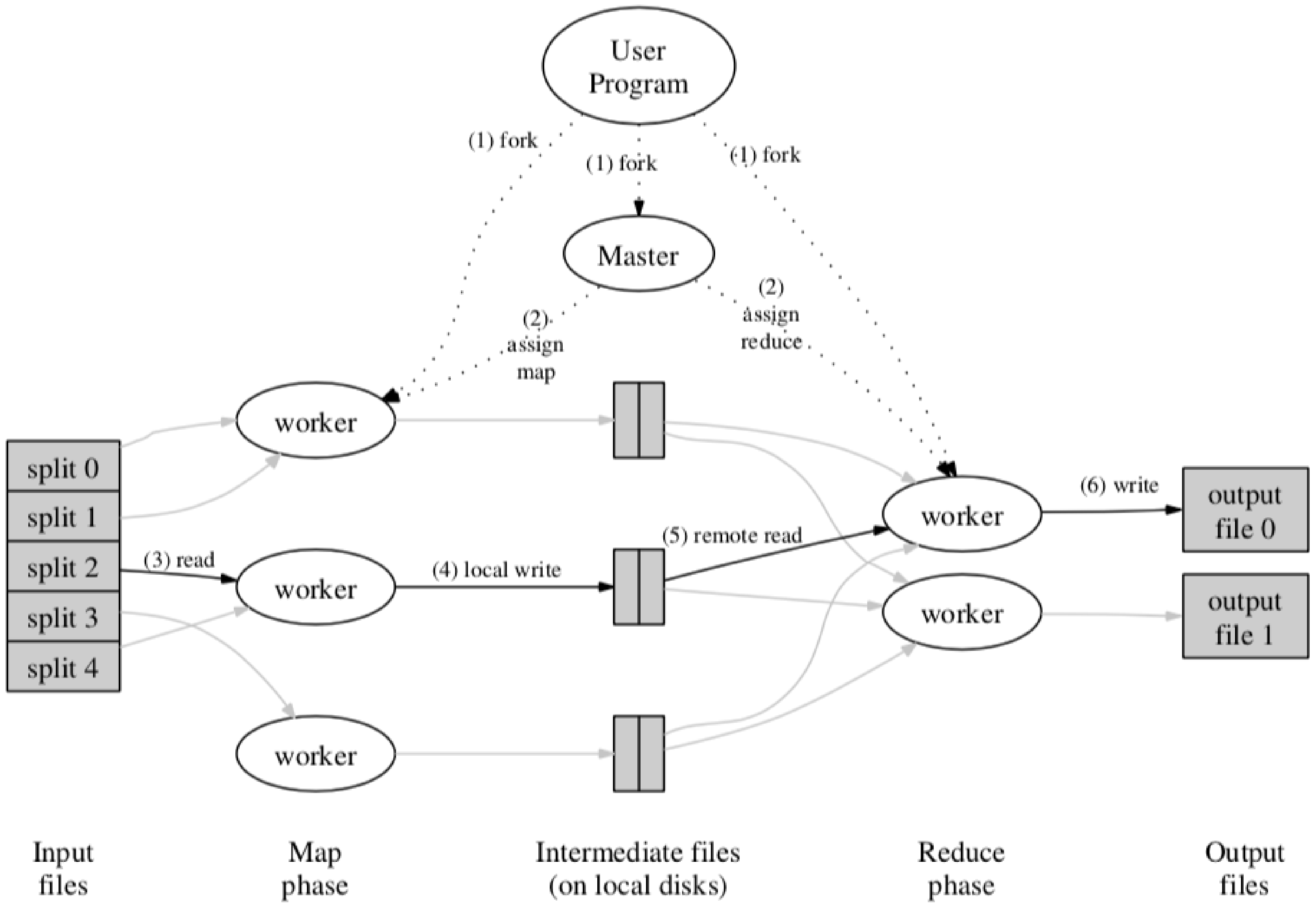

Figure 1: Execution overview

Fault Tolerance (1)

- MapReduce is designed tolerate machine failures
  - Distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program
  - Atomic commits of map and reduce task outputs to achieve this property

- Two "kind" of failures
  - Worker Failure
  - Master Failure

Fault Tolerance (2)

- **Worker Failure**
  - The master pings every worker periodically
  - If no response is received in a certain amount of time
    - the master marks the worker as failed
    - any map tasks completed by the worker are reset to idle state and become eligible for scheduling on other workers
  - Completed map tasks are re-executed
    - Their output is lost in the local filesystem
  - Completed reduce tasks do not need to be re-executed
    - Their output is stored in a global file system!
- **Master Failure**
  - Its failure is unlikely
  - Aborts the MapReduce computation if the master fails

Other issues (1)

- **Locality**
  - Network bandwidth is a relatively scarce resource
  - Input data (managed by GFS) is stored on the local disks
  - GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines
  - The MapReduce master attempts to schedule a map task on a machine that contains a replica of the corresponding input data, or schedules a map task near a replica of that task's input data
  - Most input data is read locally and consumes no network bandwidth

Other issues (2)

- **Task Granularity**
  - Choose M so that each task works roughly 16 MB to 64 MB of input
  - Make R a small multiple of the number of worker machines
  - M and R should be much larger than the number of worker machines

- **Backup Tasks**
  - Alleviate the problem of "stragglers"
  - Master schedules backup executions of *in-progress* tasks
  - The task is marked as completed whenever either the primary or the backup execution completes

Refinements (1)

- **Partitioning Function**
  - Used-defined: to partition data by some other function of the key
  -
- **Ordering Guarantees**
  - Intermediate key/value pairs are processed in increasing key order
  - Easy to generate a sorted output file per partition
  - Support efficient random access lookups by key

Refinements (2)

- **Combiner Function**
  - If there is significant repetition in the intermediate keys produced by map tasks
  - *Reduce* function is commutative and associative
  - *Combiner* function is executed in the same machine as a map task

- **Skipping Bad Records**
  - Bugs in user code cause the *Map* or *Reduce* functions to crash deterministically on certain records
  - Inform the master to be resilient to failures

- **Counters**
  - A counter facility to count occurrences of various events

```
public class WordCount2 {
    public static void main(String[] args) {
        JobClient client = new JobClient();
        JobConf conf = new JobConf(WordCount2.class);

        FileInputFormat.addInputPath(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(LongWritable.class);
        conf.setMapperClass(TokenCountMapper.class);
        conf.setCombinerClass(LongSumReducer.class);
        conf.setReducerClass(LongSumReducer.class);

        client.setConf(conf);
        try {
            JobClient.runJob(conf);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

❶ Hadoop's own TokenCountMapper

❷ Hadoop's own LongSumReducer