MapReduce

Abstract

- A programming model and an associated implementation for **processing and generating large data sets**.
- Map process a k/v pair to generate a set of **intermediate k/v pair**. Reduce function that **merges all the intermediate values associated with the same intermediate key**.

Intro

- Issues
  - parallelize the computation.
  - distribute data.
  - handle failures.
  - Load balancing.

Programming Model

Input: k/v pairs.

Output: k/v pairs.

Map func takes an **input pair and produces a set of intermediate k/v pair**.

Then the mapreduce library **groups** the intermediate pair **according to the key**.

The reduce function accepts an **intermediate key** **and set of values for that key**. Then **merges** together these values to form a possibly smaller set of values.

Example

- Frequency Counting(Each word in text)

```
#<String word ,String frequency>
map(String key , String value):
  # key: doc name
  # value: contents of document
  for word in value:
    EmitIntermediate(word,"1")

reduce(String key , Iterator values):
  # key: a single word
  # values: set of values for the word, in this case, the frequency of each word.
  result = 0
  for v in values:
    result += int(v)
  Emit(str(result))
```

- Distributed Grep
- Count of URL Access Frequency.
- Reverse Web-Link Graph

Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the **environment**.

In paper's case: large clusters of commodity PCs connected together with switched Ethernet.

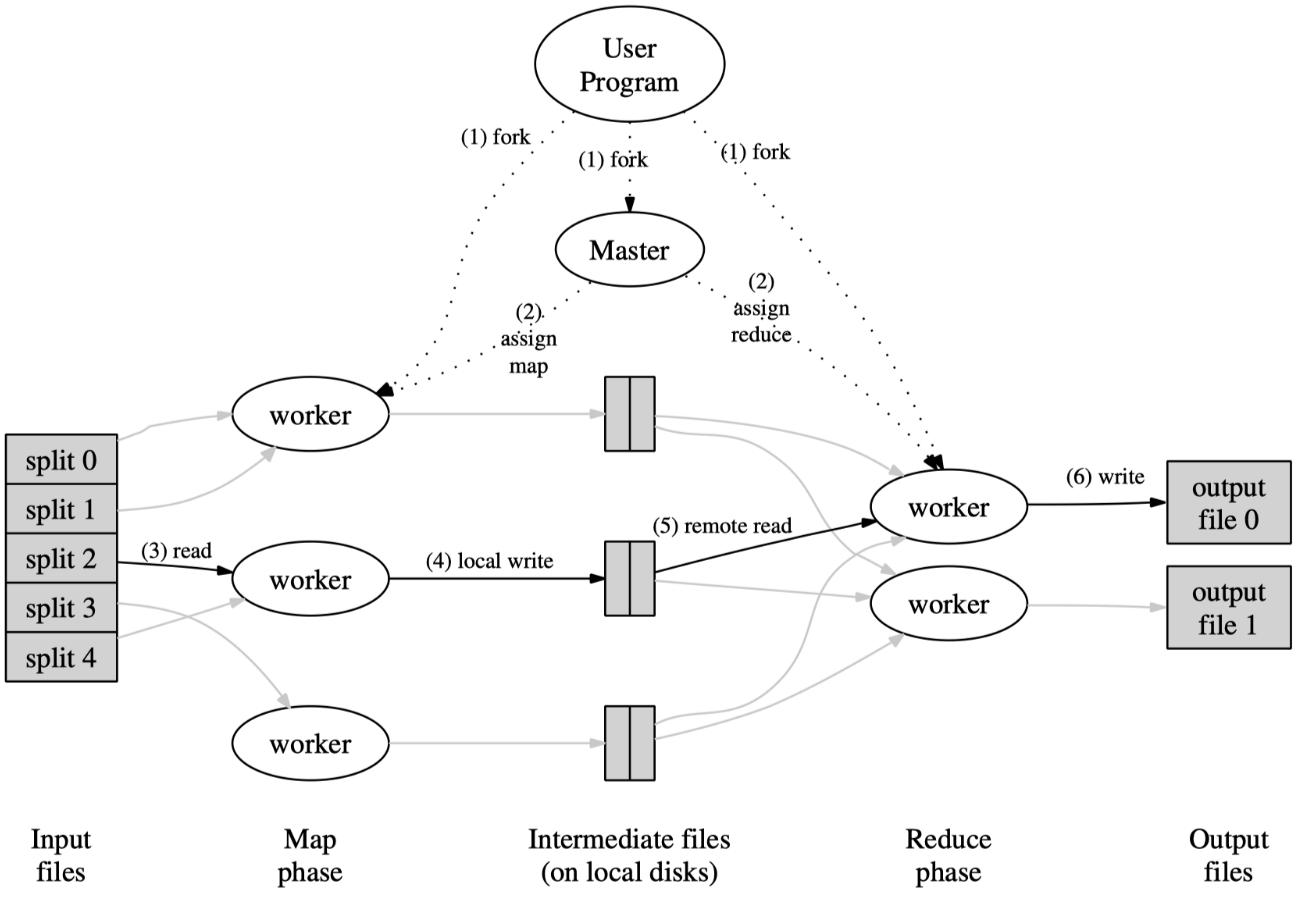

Figure 1: Execution overview

Map: Invokes on multiple workers parallelly by auto-partitioning the input file into M splits.

Reduce: Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key) mod R$).

Overview

1. The Mapreduce library in the user program *splits* the files into M pieces(typically 64 MB). Then call `fork()` to start up many copies(**Master, worker**) on a cluster of machines.
2. The master picks idle workers and assigns each one a map task or a reduce task. M map tasks and R reduce tasks.

3. Worker reads the split and parses the input data into K/V pairs and utilize user-definde Map function to produce intermediate K/V pairs **in memory**.
4. Buffered pairs splited into R regions are written to local disk Periodically.
5. When master assign reduce task to reduce workers, it uses RPC to read data. Then **sorts** it by the intermediate keys so that all occurrences of the **same key** are **grouped together**.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it **passes** the key and the corresponding **set of intermediate values** to the **user's Reduce function**. The output of the Reduce function is appended to a final output file for this reduce partition.
7. When all the work is done, the master wakes up user program.

- The final output is in R output files. If user want to combine these into one large file, then call another mapreduce task.

Fault Tolerance

- Worker Failure
  - Master **pings** every worker periodically, if no response, **reset** all the tasks by the worker, and let others to finish. For **completed map tasks** , it **need to re-execute** since the file is locally stored and notify reduce workers to read datas from the new map worker, for **reduce** tasks , **no need** to re-execute, since the output is reachable for user.
- Master Failure
  - Write periodic checkpoints, archive master status.

Task Granullarity

M, R should be much larger than # of workers.

In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data, make R a small multiple of the number of worker machines we expect to use

Backup Tasks

- Straggler:a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Lengthens the tot.

  Ways to alleviate: When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks. The task is marked as completed whenever either the primary or the backup execution completes.

Refinement

- Partition function
  - default: $hash(key)\%R$
  - URL case: $hash(Hostname(urlkey))\%R$, causes all URLs from the same host to end up in the same output file.
- **Ordering Guarantee**

    - within a given partition, the intermediate key/value pairs are processed in increasing key order.
- **Combiner Function**
    - Allow user to specify an **optional Combiner function** (basically the same as reduce function) that does **partial merging** of this data **before it is sent over the network**. Since there is **significant repletion** in the intermediate K/V pairs produced by map task, e.g.,`<the ,"1">`. If send them all, it would be a loss for network bandwidth.
- Input & output types.
    - Library offers serveral formats of input data.
        - "text mode": treats each line as a k/v pair. key: **offset** of the word, value: content of the **line**.
        - Users can add support for a new input type by providing an implementation of a simple **reader interface**.
- Skipping Bad Records

  Bugs that cause the Map or Reduce functions to crash deterministically on certain records.
    - Bugs in user code
        - fix
    - Bugs in third-party lib or okay to skip some record.
        - Optional mode of execution where the MapReduce library detects which records cause deterministic crashes and **skips** these records in order to make forward progress.
- Local Execution
    - To help facilitate debugging, testing. we have developed an **alternative** implementation of the MapReduce library that **sequentially** executes all of the work for a MapReduce operation on the local machine.

Conclusions

- Restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant.
- Network bandwidth is a **scarce** resource. A number of optimizations in our system are therefore targeted at **reducing the amount of data sent across the network**: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth.
- Third, **redundant execution** can be used to reduce the impact of slow machines, and to handle machine failures and data loss.