Paper Summary of *MapReduce: Simplified Data Processing on Large Clusters*[*]

Li, Pengda
10225101460

1 Background

In early 2000s, with the rapid development of the Internet and distributed systems, processing large-scale data sets has become a common problem. Developers in Google and other companies implemented hundreds of special-purpose computations that process large amounts of raw data. However, the input data is usually large that it must be calculated on a cluster of thousands of machines to complete in a reasonable time. How to calculate in parallel, distribute the data and handle failures make the problem more complex. Traditional methods struggled to handle this situation efficiently. This hilighted the need for a new distributed computing model to solve these problems.

2 Main Problems

Parallelization. As the data is too large to be processed on a single machine, the computation must be parallelized to run on clusters. The model should provide a simple or automatic way to parallelize the computation.

Distribution. To process efficiently, the data should be distributed across the machines. The model should handle the distribution of data and the scheduling of tasks.

Fault Tolerance. When error occurs on a machine, the model should handle the error—either by trying to restart or by scheduling the task on another machine.

These problems undeniably exists and are important in distributed computing. Parallelization is essential to break down computations and distribute them across machines. However, efficient parallelization relies on proper distribution. Fault tolerance is also important to ensure the robustness of the system. All of the three problems are important and should be addressed in a good designed distributed computing model.

3 Model

The authors proposed a new model called **MapReduce** to solve the problems above. Their inspiration came from the *map* and *reduce* functions in functional programming languages such as Lisp. They use these two functions to describe a computation.

[*]Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

3.1 `map`

`map` function handles the input data and produces a set of intermediate key/value pairs. Each mapper processes a subset of the input data independently, which makes it easy to parallelize the computation. Type of the `map` function can be described as follows:

```
map (k1, v1) → list(k2, v2)
```

The process of `map` is simple. First, the input data is divided into splits. Then each mapper processes one split with a user-defined logic, producing a set of intermediate key/value pairs. In the end, the output is a combined set of all the key/value pairs produced by all mappers.

Take the program of counting words as an example. The program reads several documents as input data and counts the frequency of each word. The `map` function reads each document and emits a key/value pair for each word in the document. Figure 1 shows the pseudocode of it.

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

Figure 1: Pseudocode of the `map` function in word count program.

The `map` function will produce a set of key/value pairs, such as `("apple", "1")`, `("banana", "1")`, etc.

3.2 `reduce`

The `reduce` function accept a key and a set of values for that key, which are produced by the `map` function. It merges the values. The type of the `reduce` function can be described as follows:

```
reduce (k2, list(v2)) → list(v2)
```

The process of reduce is also simple. First, shuffle and sort the intermediate key/value pairs by key. Then each reducer receives a key and all values associated with it. Last, the reducer processes the key and values to generated the final output with a user-defined logic.

Take the word count example again. The `reduce` function sums the values for each word to get the final count. Figure 2 shows the pseudocode of it.

```
1 reduce(String key, Iterator values):
2     // key: a word
3     // values: a list of counts
4     int result = 0;
5     for each v in values:
6         result += ParseInt(v);
7     Emit(AsString(result));
```

Figure 2: Pseudocode of the `reduce` function in word count program.

The data it receives is like `("apple", ["1", "1", "1"])`, `("banana", ["1", "1"])`, etc. And the output is like `("apple", "3")`, `("banana", "2")`, etc.

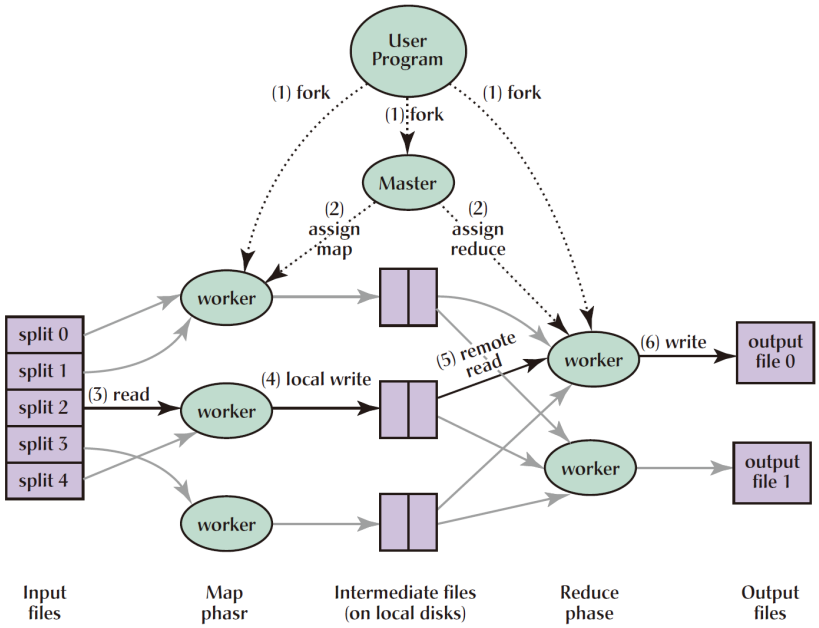

Figure 3: Execution overview of MapReduce.

4 Implement

There can be various ways to implement the MapReduce model. The authors provided a implementation of it that is widely used in Google.

4.1 Execution Overview

Figure 3 shows the overall flow of a MapReduce function. An execution process is as follows:

1. Split the input files into M pieces. Start up copies of the program on the machines in the cluster.
2. Master, a special copy of the program, assigns M map tasks and R reduce tasks to the workers.
3. A map worker reads the contents of the corresponding input split and applies the user-defined `map` function to the data. The intermediate key/value pairs are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into R regions and then it's location will be sent to master to forward to reduce workers.
5. When a reduce worker is notified of the location of the intermediate data, it reads the data from the local disk and sorts it by key.
6. For each unique key, the reduce worker applies the user-defined `reduce` function to it. The final output is written to the output file.
7. When all map and reduce tasks are completed, the master wakes up the user program and returns.

4.2 Fault Tolerance

The MapReduce model is designed to handle machine failures. The master node periodically monitors the workers. If a worker fails, the master will reassign the tasks to other workers. It also notifies relevant nodes to update task statuses accordingly. This implements an efficient fault tolerance mechanism for large-scale worker node failures.

5 Innovation

The introduction of the MapReduce model is a significant innovation in the field of distributed computing since it greatly simplifies it. Inspired by functional programming languages, MapReduce introduces two simple function abstractions, `map` and `reduce`, hiding the complexity of parallelization, distribution, and fault tolerance under the model. Developers only need to focus on the `map` and `reduce` functions, instead of worrying about the underlying details of distributed computing.

From today's perspective, MapReduce is often regarded as one of the first widely used distributed computing frameworks, especially in the context of big data processing. It also inspired many other distributed computing models, such as Apache Hadoop, Apache Spark, etc.

6 Advantages

- **Simplicity**. MapReduce is simple and easy to understand. It makes it come true to parallelize and distribute computations on large clusters simply by defining two functions.
- **Scalability**. The model can be easily scaled to handle large-scale data of PB level. And we can expand it by just adding more machines to the cluster.
- **Fault Tolerance**. MapReduce can automatically detect failures and handle them. It ensures the robustness of the system.

7 Disadvantages

- **High Latency**. Since the process of mapping, shuffling, sorting, and reducing takes time, the latency of the system is relatively high. It may not be suitable for real-time processing.
- **High Disk I/O**. As it shown in Figure 3, the intermediate data is written to disk and read from disk. This may cause high disk I/O. Since data is written and read from disk, it can not use the memory efficiently, which may cause performance issues.