Parallel And Distributed Programming Paradigms

Module 3

Distributed computing system and Parallel computing

A distributed computing system is a set of computational engines connected by a network to achieve a common goal of running a job or an application.

A computer cluster or network of workstations is an example of a distributed computing system.

Parallel computing is the simultaneous use of more than one computational engine (not necessarily connected via a network) to run a job or an application.

Parallel computing may use either a distributed or a nondistributed computing system such as a multiprocessor platform

Advantages

Running a parallel program on a distributed computing system

Users' perspective, it decreases application response time; from the distributed computing systems'

Parallel and Distributed Programming Paradigms standpoint, it increases throughput and resource utilization.

Disadvantage

Running a parallel program on a distributed computing system, however, could be a very complicated process.

Factors

- **Computation partitioning**

This splits a given job or a program into smaller tasks.

Partitioning greatly depends on correctly identifying portions of the job or program that can be performed concurrently.

- **Data partitioning**

This splits the input or intermediate data into smaller pieces.

Data pieces may be processed by different parts of a program or a copy of the same program.

• **Mapping**

This assigns the either smaller parts of a program or the smaller pieces of data to underlying **resources.**

This process aims to appropriately assign such parts or pieces to be run simultaneously on different workers and is usually handled by **resource allocators** in the system.

• **Synchronization**

Different workers may perform different tasks, synchronization and coordination among workers is necessary so that **race conditions** are prevented and **data dependency** among different workers is properly managed.

Multiple accesses to a shared resource by different workers may raise race conditions

Data dependency happens when a worker needs the processed data of other workers.

- **Communication**

Data dependency is one of the main reasons for communication among workers, communication is always triggered when the intermediate data is sent to workers.

- **Scheduling**

For a job or program, when the number of computation parts (tasks) or data pieces is more than the number of available workers, a scheduler selects a sequence of tasks or data pieces to be assigned to the workers.

Motivation for Programming Paradigms

Handling the whole data flow of parallel and distributed programming is very time consuming and requires specialized knowledge of programming

Dealing with these issues may affect the productivity of the programmer and may even result in affecting the program's time to market.

It may detract the programmer from concentrating on the logic of the program itself.

Therefore, parallel and distributed programming paradigms or models are offered to abstract many parts of the data flow from users.

Summary

- To improve productivity of programmers
- To decrease programs' time to market
- To leverage underlying resources more efficiently
- To increase system throughput
- To support higher levels of abstraction

MapReduce, Hadoop, and Dryad are three of the most recently proposed parallel and distributed programming models.

	Google MapReduce [28]	Apache Hadoop [23]
Programming Model	MapReduce	MapReduce
Data Handling	GFS (Google File System)	HDFS (Hadoop Distributed File System)
Scheduling	Data locality	Data locality; rack-aware, dynamic task scheduling using global queue
Failure Handling	Reexecution of failed tasks; duplicated execution of slow tasks	Reexecution of failed tasks; duplicated execution of slow tasks
HLL Support	Sawzall [31]	Pig Latin [32,33]
Environment	Linux cluster	Linux clusters, Amazon Elastic MapReduce on EC2
Intermediate Data Transfer	File	File, HTTP

Comparison of MapReduce Type Systems

Parallel Programming in C

```
#include <omp.h>
main ( ) {
int var1, var2, var3;
Serial code
    .
Beginning of parallel section. Fork a team of threads.
Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
 {
 Parallel section executed by all threads
     .
 All threads join master thread and disband
 }
Resume serial code
    .
}
```

MapReduce

MapReduce, as introduced is a **software framework** which supports parallel and distributed computing on large data sets

It abstracts the data flow of running a parallel program on a distributed computing system by providing users with **two interfaces** in the form of two functions**: Map and Reduce.**

Users can **override** these two functions to interact with and manipulate the data flow of running their programs.

In this framework the **"value"** part of the data, (key, value), is **the actual data,** and the **"key"** part is only used by the MapReduce controller **to control the data flow.**

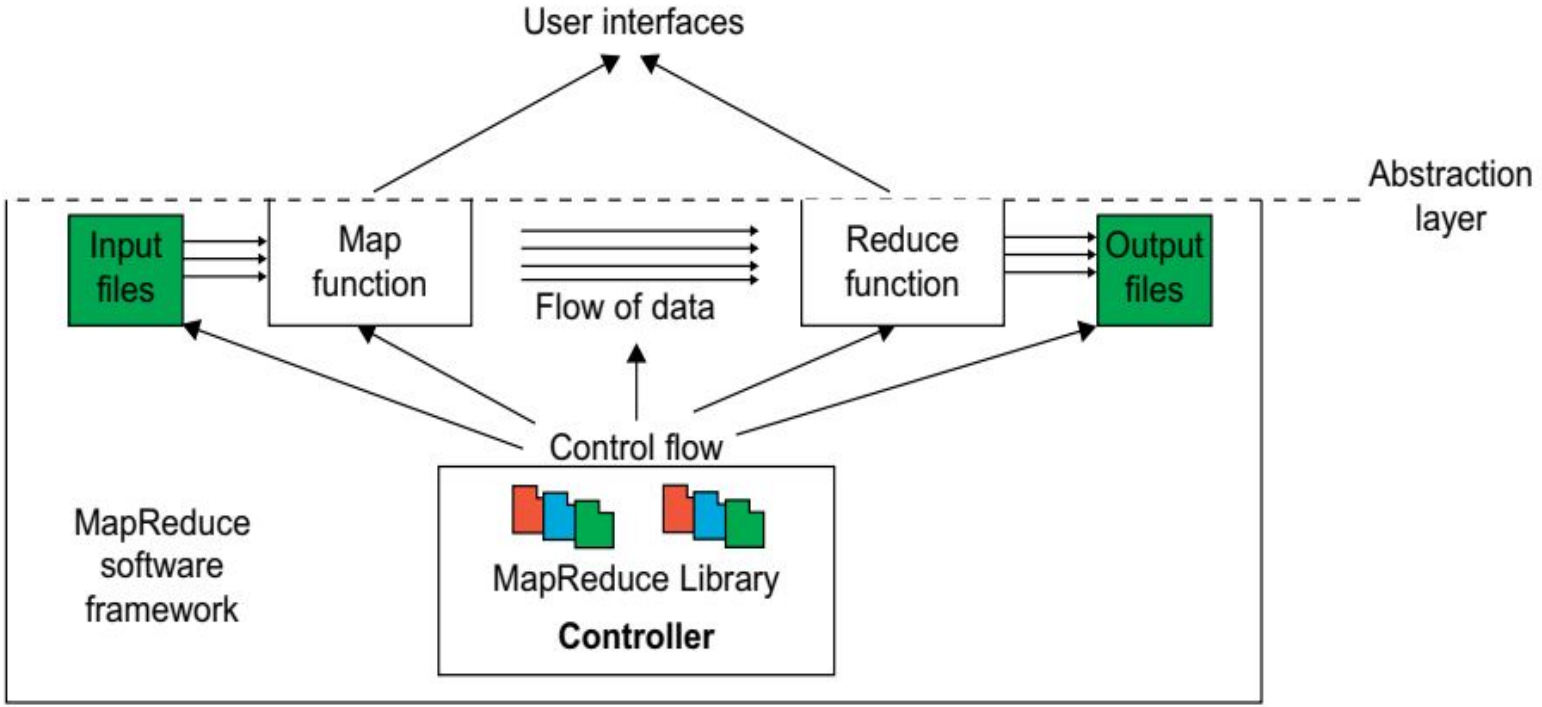

FIGURE 6.1

MapReduce framework: Input data flows through the Map and Reduce functions to generate the output result under the control flow using MapReduce software library. Special user interfaces are used to access the Map and Reduce resources.

Formal Definition of MapReduce

The overall structure of a user's program containing the Map, Reduce, and the Main functions is given below. The Map and Reduce are two major subroutines. They will be called to implement the desired function performed in the main program.

```
Map Function (. . . . )
  {
    ... ...
  }
Reduce Function (. . . . )
  {
    ... ...
  }
Main Function (. . . . )
  {
    Initialize Spec object
    ... ...
    MapReduce (Spec, & Results)
  }
```

MapReduce Logical Data Flow

The input data to the Map function is in the form of a (key, value) pair.

The key is the line offset within the input file and the value is the content of the line.

The output data from the Map function is structured as (key, value) pairs called **intermediate (key, value) pairs.**

The user-defined Map function processes each input (key, value) pair and produces a number of **(zero, one, or more) intermediate (key, value) pairs.**

The goal is to process all input (key, value) pairs to the Map function in parallel

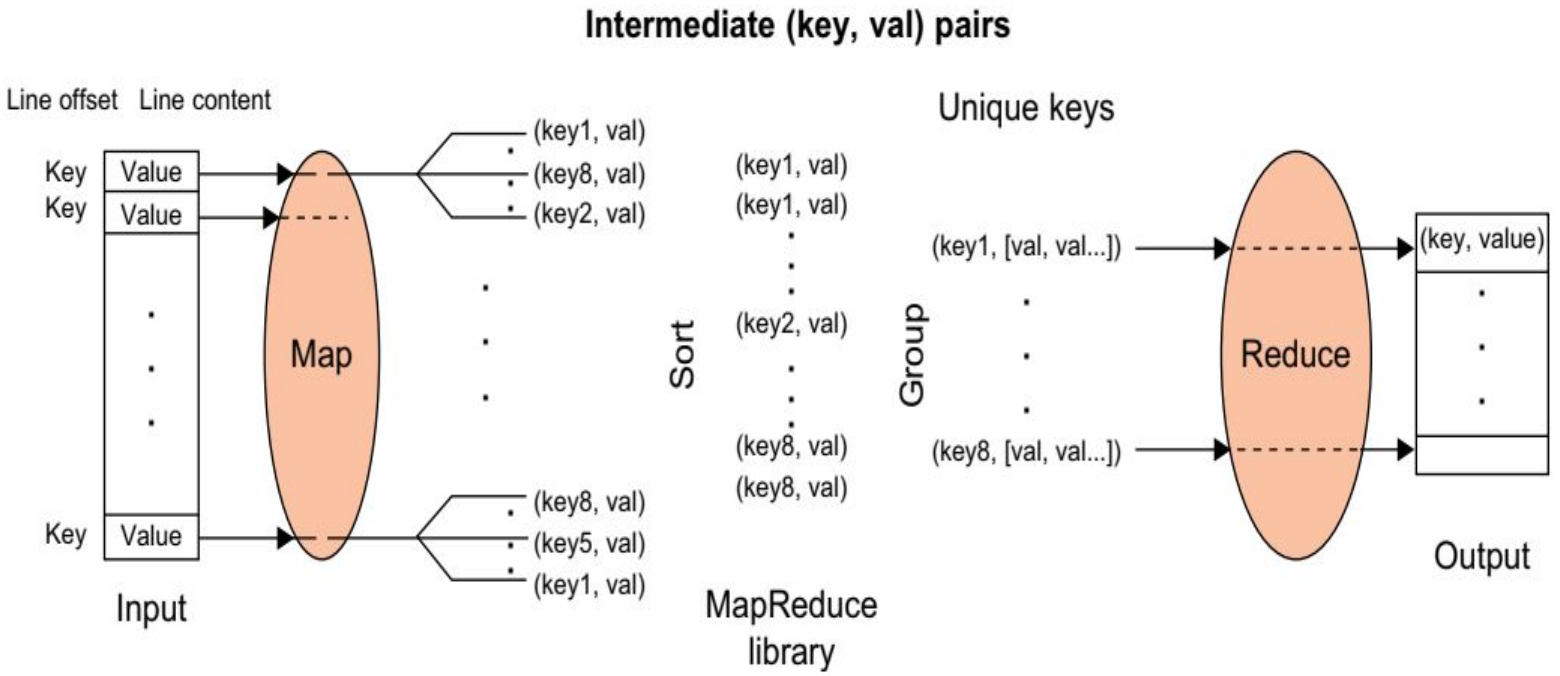

FIGURE 6.2

MapReduce logical data flow in 5 processing stages over successive (key, value) pairs.

MapReduce Actual Data and Control Flow

The main responsibility of the **MapReduce framework** is to efficiently run a user's program on a distributed computing system.

The MapReduce framework meticulously handles all partitioning, mapping, synchronization, communication, and scheduling details of such data flows.

The following distinct steps:

1. Data partitioning

The MapReduce library splits the input data (files), already stored in GFS, into **M pieces** that also correspond to the number of map tasks.

2. Computation partitioning

This is implicitly handled by obliging users to write their programs in the form of the Map and Reduce functions.

The **MapReduce library only generates copies of a user program containing the Map and the Reduce functions**, distributes them, and starts them up on a number of available **computation engines.**

3. Determining the master and workers

The MapReduce architecture is based on a master worker model.

One of the copies of the user program becomes the **master** and the rest become **workers**.

The master picks idle workers, and assigns the map and reduce tasks to them.

A map/reduce worker is typically a computation engine such as a cluster node to run map/reduce tasks by executing Map/Reduce functions.

Steps 4–7 describe the map workers

4. Reading the input data (data distribution)

Each map worker reads its corresponding portion of the input data, namely the input data split, and sends it to its Map function.

Although a map worker may run more than one Map function, which means it has been assigned more than one input data split, each worker is usually assigned one input split only.

5. Map function

Each Map function receives the input data split as a set of (key, value) pairs to process and produce the intermediated (key, value) pairs.

6. Combiner function

An optional local function within the map worker which applies to intermediate (key, value) pairs.

The user can invoke the **Combiner function** inside the user program.

The Combiner function runs the same code written by users for the Reduce function as its functionality is identical to it.

The **Combiner function merges the local data of each map worker** before sending it over the network to effectively reduce its communication costs.

The MapReduce framework will also **sort and group the local data on each map worker** if the user invokes the Combiner function

The figure shows the data flow implementation of all data flow steps.

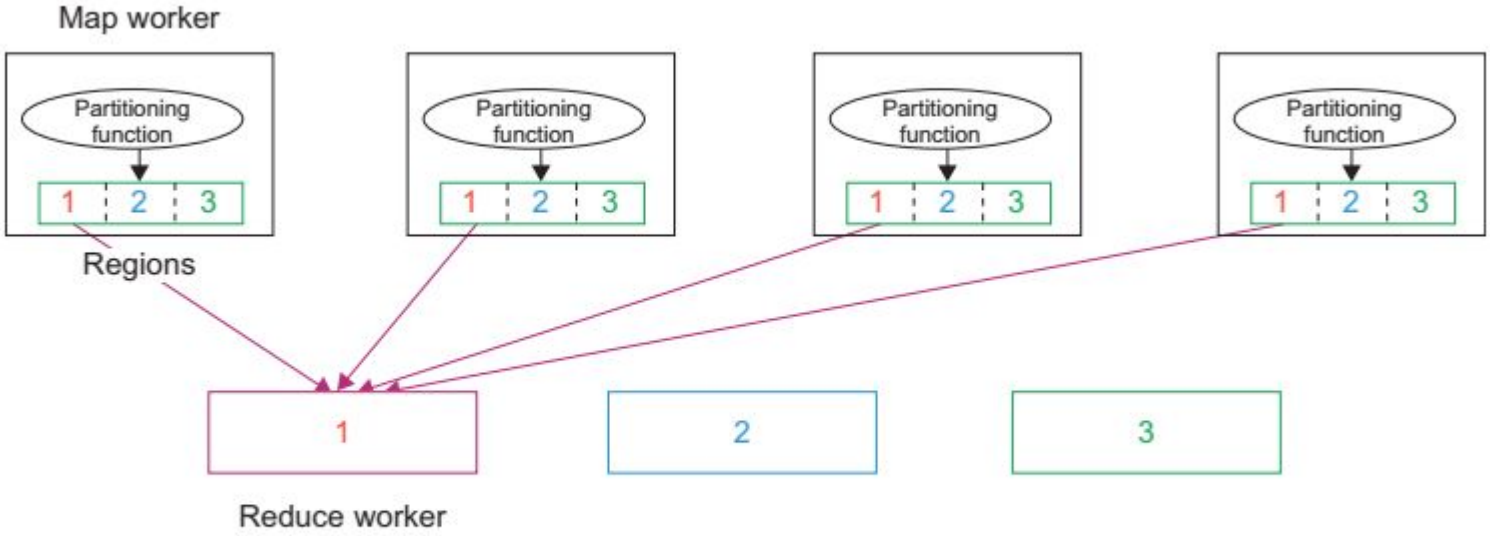

FIGURE 6.4

Use of MapReduce *partitioning* function to link the Map and Reduce workers.

7. Partitioning function

The MapReduce data flow, the intermediate (key, value) pairs with **identical keys are grouped together** because all values inside each group should be **processed by only one Reduce function to generate the final result.**

There are **M map and R reduce tasks**, intermediate(key, value) pairs with the same key might be produced by different map tasks, although they should be grouped and processed together by one Reduce function only.

The intermediate (key, value) pairs produced by each map worker are partitioned into R regions, equal to the number of reduce tasks, by the Partitioning function to guarantee that all (key, value) pairs with identical keys are stored in the same region.

Reduce worker i reads the data of region i of all map workers, all (key, value) pairs with the same key will be gathered by reduce worker i accordingly

To implement this technique, a Partitioning function could simply be a hash function (e.g., Hash(key) mod R) that forwards the data into particular regions. It is also worth noting that the locations of the buffered data in these R partitions are sent to the master for later forwarding of data to the reduce workers.

The following are two networking steps:

8. Synchronization

MapReduce applies a simple synchronization policy to coordinate map workers with reduce workers, in which the **communication between them starts when all map tasks finish.**

9. Communication

Reduce worker i, already notified of the location of region i of all map workers, uses a remote procedure call to read the data from the respective region of all map workers.

All reduce workers read the data from all map workers, all-to-all communication among all map and reduce workers, which incurs network congestion, occurs in the network.

A data transfer module was proposed to schedule data transfers independently.

10. Sorting and Grouping

When the process of reading the input data is finalized by a reduce worker, the data is initially **buffered in the local disk of the reduce worker.**

Then the reduce worker **groups intermediate (key, value) pairs by sorting the data based on their keys**, followed by grouping all occurrences of identical keys.

The buffered data is sorted and grouped because the number of unique keys produced by a map worker may be more than R regions in which more than one key exists in each region of a map worker

11. Reduce function

The reduce worker iterates over the grouped (key, value) pairs, and for each unique key, it sends the key and corresponding values to the Reduce function.

Then this function processes its input data and stores the output results in predetermined files in the user's program.

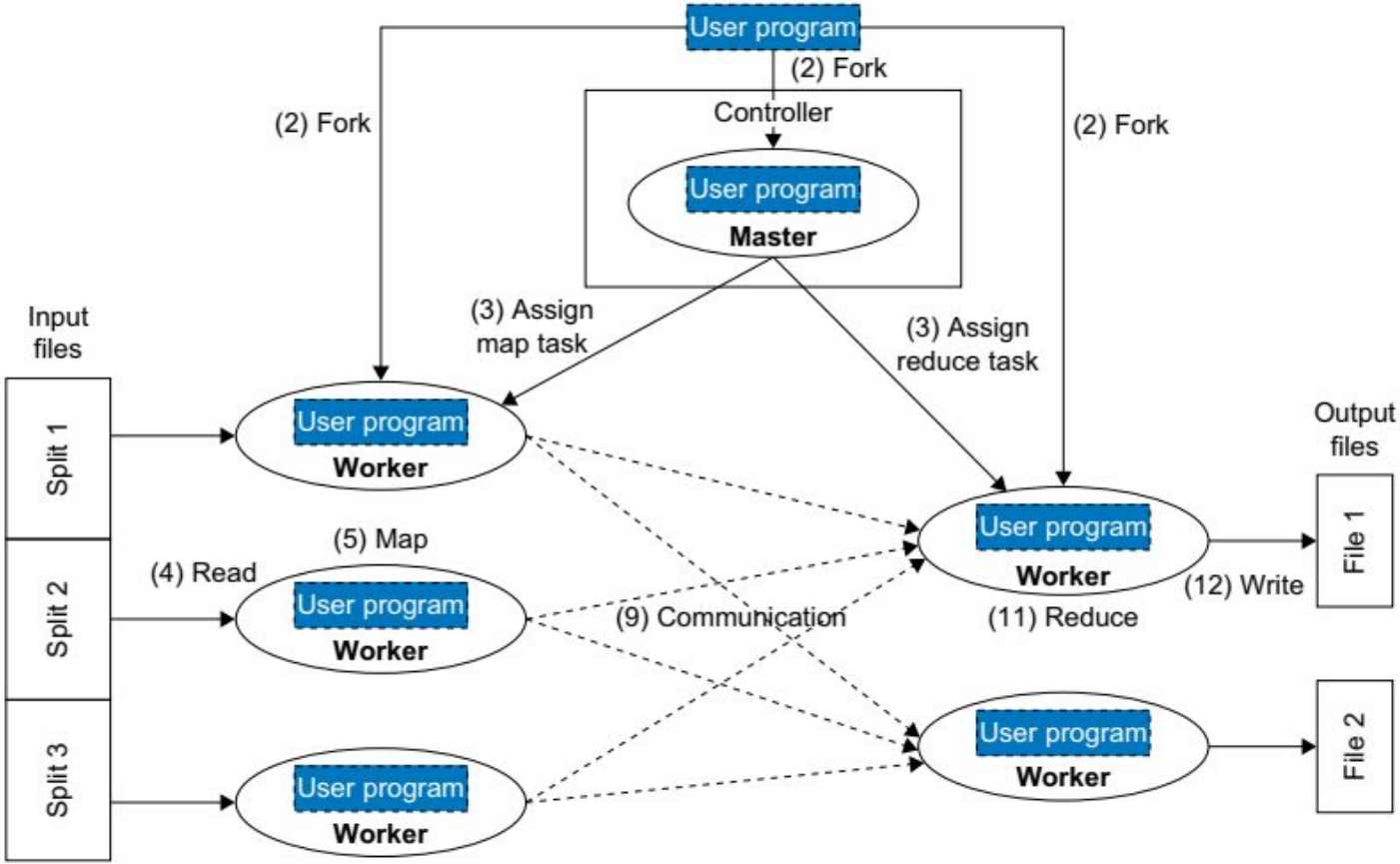

FIGURE 6.6

Control flow implementation of the MapReduce functionalities in Map workers and Reduce workers (running user programs) from input files to the output files under the control of the master user program.

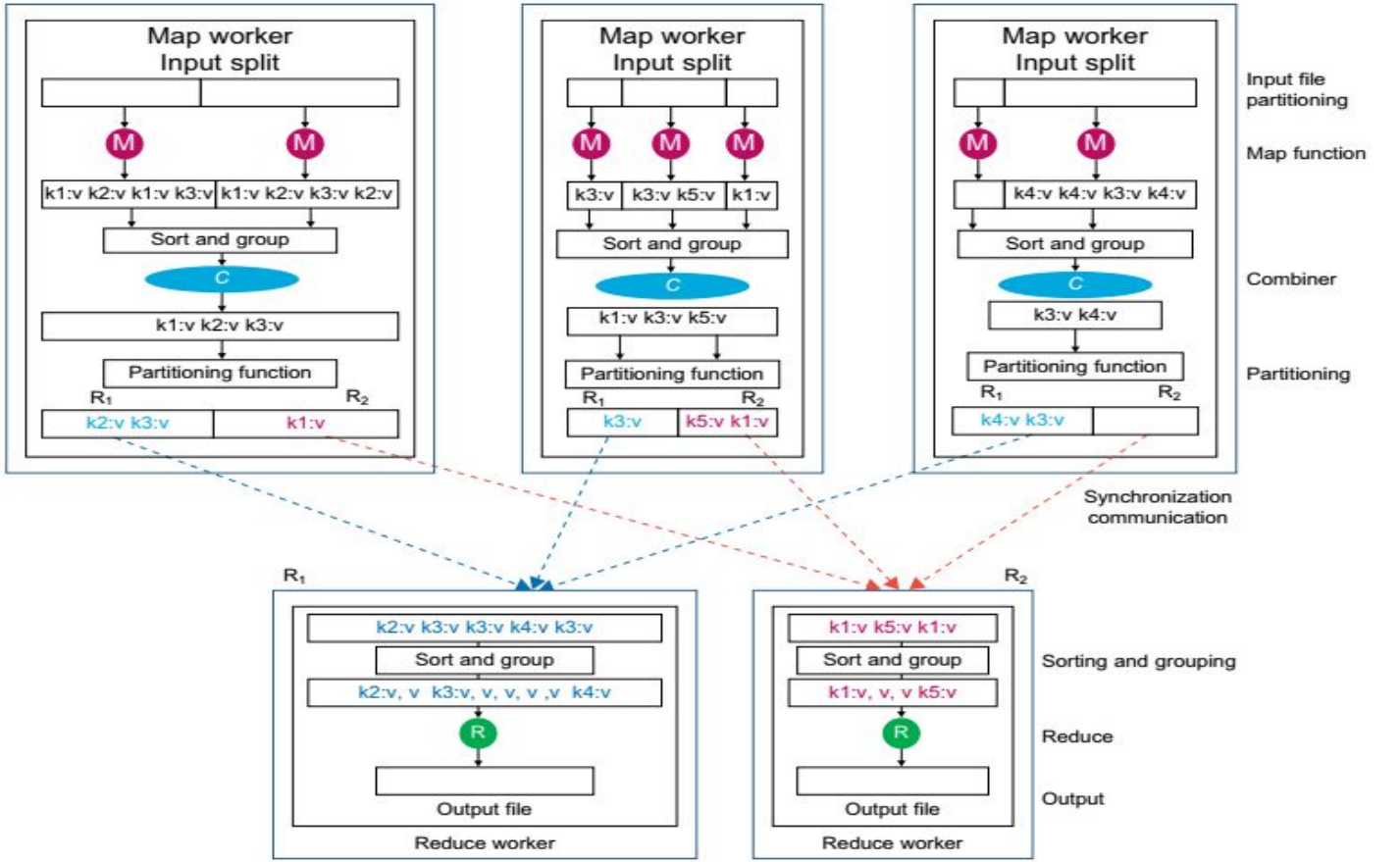

FIGURE 6.5

Data flow implementation of many functions in the Map workers and in the Reduce workers through multiple sequences of partitioning, combining, synchronization and communication, sorting and grouping, and reduce operations.