MapReduce

A framework for large-scale parallel processing

By Paul Krzyzanowski
November 2017

Introduction

Traditional programming tends to be serial in design and execution. We tackle many problems with a sequential, stepwise approach and this is reflected in the corresponding program. With *parallel programming*, we break up the processing workload into multiple parts, that can be executed concurrently on multiple processors. Not all problems can be parallelized. The challenge is to identify as many tasks as possible that can run concurrently. Alternatively, we can identify data groups that can be processed concurrently. This will allow us to divide the data among multiple concurrent tasks.

The most straightforward situation that lends itself to parallel programming is one where there is no dependency among data. Data can be split into chunks and each process can be assigned a chunk to work on. If we have lots of processors, we can split the data into lots of chunks. A **master/worker** approach is a design where a *master* process coordinates overall activity. It identifies the data, splits it up based on the number of available workers, and assigns a data segment to each worker. A **worker** receives the data segment from the master, performs whatever processing is needed on the data, and then sends results to the master. At the end, the master does whatever final processing (e.g., merging) is needed to the resultant data.

MapReduce Etymology

MapReduce is was created at Google in 2004 by Jeffrey Dean and Sanjay Ghemawat. The name is inspired from map and reduce functions in the LISP programming language. In LISP, the *map* function takes as parameters a function and a set of values. That function is then applied to each of the values. For example:

```
(map 'length '(() (a) (ab) (abc)))
```

applies the *length* funciton to each of the three items in the list. Since *length* returns the length of an item, the result of *map* is a list containing the length of each item:

```
(0 1 2 3)
```

The *reduce* function is given a binary function and a set of values as parameters. It combines all the values together using the binary function. If we use the + (add) function to reduce the list (0 1 2 3):

```
(reduce #'+ '(0 1 2 3))
```

we get:

```
6
```

If we think about how the *map* operation, we realize that each application of the function to a value can be performed in parallel (concurrently) since there is no dependence of one upon another. The *reduce* operation can take place only after the *map* is complete.

MapReduce is not an implementation of these LISP functions; they are merely an inspiration and etymological predecessor.

MapReduce

MapReduce is a framework for parallel computing. Programmers get a simple API and do not have to deal with issues of parallelization, remote execution, data distribution, load balancing, or fault tolerance. The framework makes it easy for one to use thousands of processors to process huge amounts of data (e.g., terabytes and petabytes).

From a user's perspective, there are two basic operations in MapReduce: *Map* and *Reduce*.

The **Map** function reads a stream of data and parses it into intermediate *(key, value)* pairs. When that is complete, the **Reduce** function is called once for each unique key that was generated by *Map* and is given the key and a list of all values that were generated for that key as a parameter. The keys are presented in sorted order.

As an example of using MapReduce, consider the task of counting the number of occurrences of each word in a large collection of documents. The user-written *Map* function reads the document data and parses out the words. For each word, it writes the (key, value) pair of (word, 1). That is, the word is treated as the key and the associated value of 1 means that we saw the word once. This intermediate data is then sorted by MapReduce by keys and the user's *Reduce* function is called for each unique key. Since the only values are the count of 1, *Reduce* is called with a list of a "1" for each occurence of the word that was parsed from the document. The function simply adds them up to generate a total word count for that word. Here's what the code looks like:

```
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));
```

Let us now look at what happens in greater detail.

MapReduce: More Detail

To the programmer, MapReduce is largely seen as an API: communication with the various machines that play a part in execution is hidden. MapReduce is implemented in a master/worker configuration, with one master serving as the coordinator of many workers. A worker may be assigned a role of either a *map worker* or a *reduce worker*.

Step 1. Split input

The first step, and the key to massive parallelization in the next step, is to split the input into multiple pieces. Each piece is called a **split**, or **shard**. For *M* map workers, we want to have *M* shards, so that each worker will have something to work on. The number of workers is mostly a function of the amount of machines we have at our disposal.

The MapReduce library of the user program performs this split. The actual form of the split may be specific to the location and form of the data. MapReduce allows the use of custom readers to split a collection of inputs into shards, based on specific format of the files.

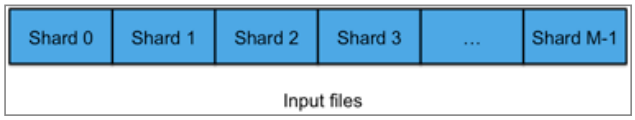

Figure 1. Split input into shards

Step 2. Fork processes

The next step is to create the master and the workers. The **master** is responsible for dispatching jobs to workers, keeping track of progress, and returning results. The master picks idle workers and assigns them either a **map task** or a **reduce task**. A map task works on a single shard of the original data. A reduce task works on intermediate data generated by the map tasks. In all, there will be *M* map tasks and *R* reduce tasks. The number of reduce tasks is the number of partitions defined by the user. A worker is sent a message by the master identifying the program (map or reduce) it has to load and the data it has to read.

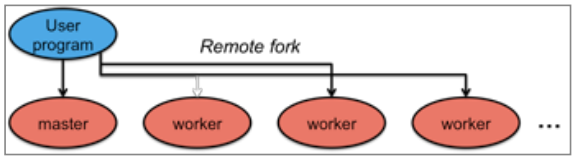

Figure 2. Remotely execute worker processes

Step 3. Map

Each *map* task reads from the input shard that is assigned to it. It parses the data and generates *(key, value)* pairs for data of interest. In parsing the input, the *map* function is likely to get rid of a lot of data that is of no interest. By having many map workers do this in parallel, we can linearly scale the performance of the task of extracting data.

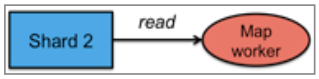

Figure 3. Map task

Step 4: Map worker: Partition

The stream of *(key, value)* pairs that each worker generates is buffered in memory and periodically stored on the local disk of the map worker. This data is partitioned into *R* regions by a partitioning function.

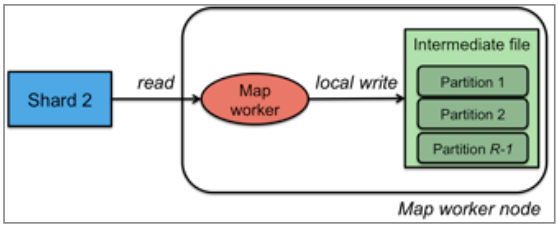

Figure 4. Create intermediate files

The **partitioning function** is responsible for deciding which of the *R* reduce workers will work on a specific key. The default partitioning function is simply a hash of *key* modulo *R* but a user can replace this with a custom partition function if there is a need to have certain keys processed by a specific reduce worker.

Step 5: Reduce: Sort (Shuffle)

When all the map workers have completed their work, the master notifies the **reduce workers** to start working. The first thing a reduce worker needs to is to get the data that it needs to present to the user's *reduce* function. The reduce worker contacts every map worker via remote procedure calls to get the *(key, value)* data that was targeted for its partition. This data

is then sorted by the keys. Sorting is needed since it will usually be the case that there are many occurrences of the same key and many keys will map to the same reduce worker (same partition). After sorting, all occurrences of the same key are grouped together so that it is easy to grab all the data that is associated with a single key.

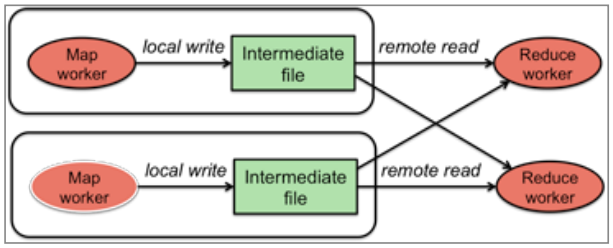

Figure 5. Sort and merge partitioned data

This phase is sometimes called the **shuffle** phase.

Step 6: Reduce function

With data sorted by keys, the user's *Reduce* function can now be called. The reduce worker calls the *Reduce* function once for each unique key. The function is passed two parameters: the key and the list of intermediate values that are associated with the key.

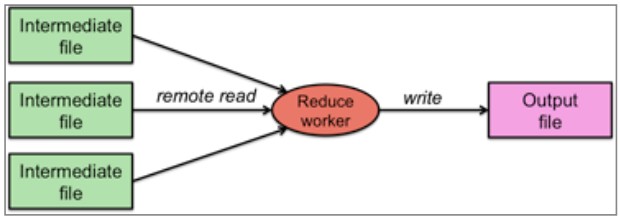

Figure 6. Reduce function writes output

The *Reduce* function writes output sent to file.

Step 7: Done!

When all the reduce workers have completed execution, the master passes control back to the user program. Output of MapReduce is stored in the *R* output files that the *R* reduce workers created.

The big picture

Figure 7 illustrates the entire MapReduce process. The client library initializes the shards and creates map workers, reduce workers, and a master. Map workers are assigned a shard to process. If there are more shards than map workers, a map worker will be assigned another shard when it is done. Map workers invoke the user's *Map* function to parse the data and write intermediate *(key, value)* results onto their local disks. This intermediate data is partitioned into *R* partitions according to a partioning function. Each of *R* reduce workers contacts all of the map workers and gets the set of *(key, value)* intermediate data that was targeted to its partition. It then calls the user's *Reduce* function once for each unique key and gives it a list of all values that were generated for that key. The *Reduce* function writes its final output to a file that the user's program can access once MapReduce has completed.

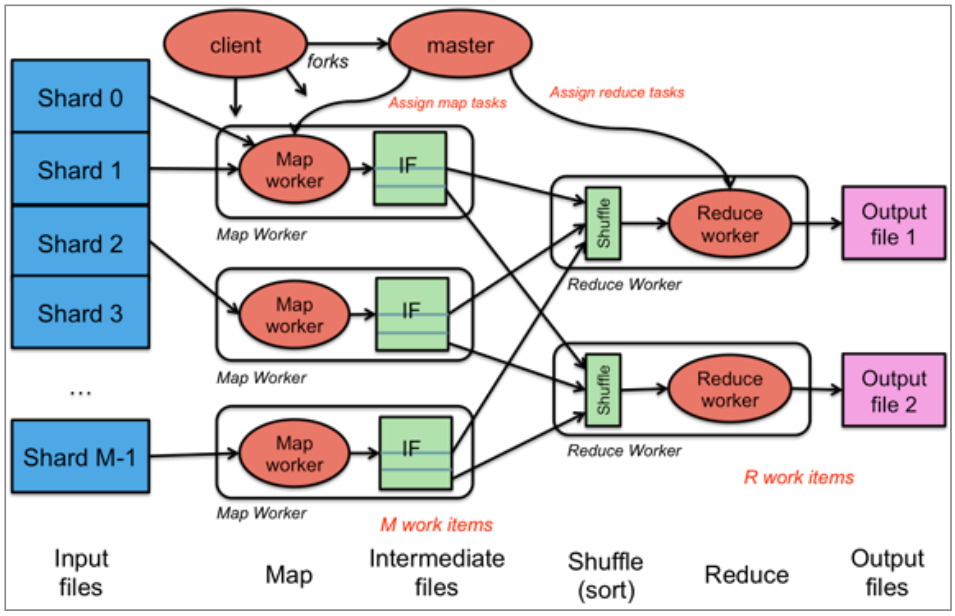

Figure 7. MapReduce

* * *

Dealing with failure

The master pings each worker periodically. If no response is received within a certain time, the worker is marked as *failed*. Any map or reduce tasks that have been assigned to this worker are reset back to the inital state and rescheduled on other workers.

Locality

MapReduce is built on top of GFS, the Google File System. Input and output files are stored on GFS. The MapReduce workers run on GFS chunkservers. The MapReduce master attempts to schedule a *map* worker onto one of the machines that holds a copy of the input chunk that it needs for processing. Alternatively, MapReduce may read from or write to Bigtable.

What is it good for and who uses it?

MapReduce is clearly not a general-purpose framework for all forms of parallel programming. Rather, it is designed specifically for problems that can be broken up into the the map-reduce paradigm. Perhaps surprisingly, there are a lot of data analysis tasks that fit nicely into this model. While MapReduce is heavily used within Google, it also found use in companies such as Yahoo, Facebook, and Amazon.

The original, and proprietary, implementation was done by Google. It is used internally for a large number of Google services. The Apache Hadoop project built a clone to specs defined by Google. Amazon, in turn, uses Hadoop MapReduce running on their EC2 (elastic cloud) computing-on-demand service to offer the Amazon Elastic MapReduce service.

Some problems it has been used for include:

Distributed grep (search for words)

Map: emit a line if it matches a given pattern

Reduce: just copy the intermediate data to the output

Count URL access frequency

Map: process logs of web page access; output <URL, 1>

Reduce: add all values for the same URL

Reverse web-link graph

Map: output <target, source> for each link to target in a page source

Reduce: concatenate the list of all source URLs associated with a target. Output <target, list(source)>

Inverted index

Map: parse document, emit <word, document-ID> pairs

Reduce: for each word, sort the corresponding document IDs; emits a <word, list(document-ID)> pair. The set of all output pairs is an inverted index

References

Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

The introductory and definitive paper on MapReduce

Jerry Zhao, Jelena Pjesivac-Grbovic, MapReduce: The programming model and practice, SIGMETRICS'09 Tutorial, 2009.

Tutorial of the MapReduce programming model.

MapReduce.org,

Amazon Elastic MapReduce,

Amazon's MapReduce service

Google: MapReduce in a Week , Google Code University

MapReduce course, a part of 2008 Independent Activities Period at MIT