International Journal of Distributed and Parallel Systems (IJDPS) Vol.3, No.1, January 2012

Parallel Processing of cluster by Map Reduce

Madhavi Vaidya, Department of Computer Science

Vivekanand College, Chembur, Mumbai

vamadhavi04@yahoo.co.in

Abstract

MapReduce is a parallel programming model and an associated implementation introduced by Google. In the programming model, a user specifies the computation by two functions, Map and Reduce. The underlying MapReduce library automatically parallelizes the computation, and handles complicated issues like data distribution, load balancing and fault tolerance. Massive input, spread across many machines, need to parallelize. Moves the data, and provides scheduling, fault tolerance. The original MapReduce implementation by Google, as well as its open-source counterpart, Hadoop, is aimed for parallelizing computing in large clusters of commodity machines. Map Reduce has gained a great popularity as it gracefully and automatically achieves fault tolerance. It automatically handles the gathering of results across the multiple nodes and returns a single result or set.

This paper gives an overview of MapReduce programming model and its applications. The author has described here the workflow of MapReduce process. Some important issues, like fault tolerance, are studied in more detail. Even the illustration of working of Map Reduce is given.

The data locality issue in heterogeneous environments can noticeably reduce the Map Reduce performance. In this paper, the author has addressed the illustration of data across nodes in a way that each node has a balanced data processing load stored in a parallel manner. Given a data intensive application running on a Hadoop Map Reduce cluster, the auhor has exemplified how data placement is done in Hadoop architecture and the role of Map Reduce in the Hadoop Architecture. The amount of data stored in each node to achieve improved data-processing performance is explained here.

Keywords:

parallelization, Hadoop, Google File Systems, Map Reduce, Distributed File System

Introduction

In this paper author has made a study on Parallel Data Processing in context with Map Reduce Framework. MapReduce is an attractive model for parallel data processing in high-performance cluster computing environments. The scalability of MapReduce is proven to be high, because a job in the MapReduce model is partitioned into numerous small tasks running on multiple machines in a large-scale cluster. MapReduce is a widely used method of parallel computation on massive data. MapReduce was designed (by Google, Yahoo, and others) to marshal all the storage and computation resources of a dedicated cluster computer. The most recently published report indicates that, by 2008, Google was running over one hundred thousand MapReduce jobs per day and processing over 20 PB of data in the same period [6]. By 2010, Google had created over ten thousand distinct MapReduce programs performing a variety of functions, including large-scale graph processing, text processing etc.

DOI : 10.5121/ijdps.2012.3113

Google File System and Hadoop Distributed File System have common design goals. They are both targeted at data intensive computing applications where massive data files are common. Both are optimized in favor of high sustained bandwidths instead of low latency, to better support batch-processing style workloads. Both run on clusters built with commodity hardware components where failures are common, motivating the inclusion of built-in fault tolerance mechanisms through replication.

In both systems, the filesystem is implemented by user level processes running on top of a standard operating system (in the case of GFS, Linux). A single GFS master server running on a dedicated node is used to coordinate storage resources and manage metadata. Multiple slave servers (*chunkservers* in Google parlance) are used in the cluster to store data in the form of large blocks (*chunks*), each identified with a 64-bit ID. Files are saved by the chunkservers on local disk as native Linux files, and accessed by chunk ID and offset within the chunk. Both HDFS and GFS use the same default chunk size (64MB) to reduce the amount of metadata needed to describe massive files, and to allow clients to interact less often with the single master. Finally, both use a similar replica placement policy that saves copies of data in many locations—locally, to the same rack, and to a remote rack — to provide fault tolerance and improve performance.

Google File System

The Google File System (GFS) is a proprietary Distributed File System developed by Google.

It is designed (Figure 1) to provide efficient, reliable access to data using large clusters of commodity hardware. The files are huge and divided into chunks of 64 megabytes.[7] Most files are mutated by appending new data rather than overwriting existing data: once written, the files are only read and often only sequentially. This DFS is best suited for scenarios in which many large files are created once but read many times. The GFS is optimized to run on computing clusters where the nodes are cheap computers. Hence, there is a need for precautions against the high failure rate of individual nodes and data loss. In the Google file system there can be 100 to 1000 PCs in a cluster can be used.

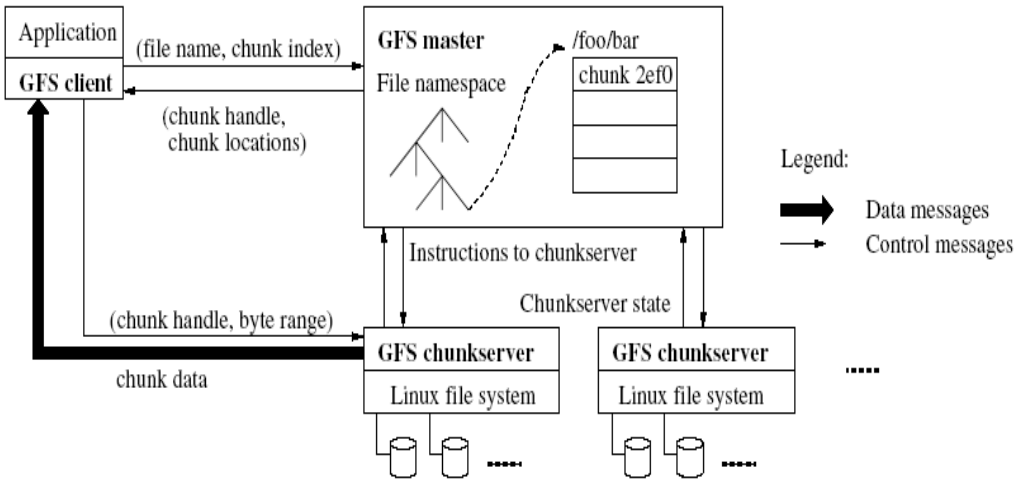


Figure 1:Google File System

Chunkserver Architecture

Server

- Stores 64 MB file chunks on local disk using standard Linux filesystem, each with version number and checksum
- Read/write requests specify chunk handle and byte range
- Chunks replicated on configurable number of chunkservers (default: 3)
- No caching of file data

Client

- Issues control (metadata) requests to master server
- Issues data requests directly to chunkservers
- Caches metadata
- Does no caching of data
  - No consistency hence difficulties among clients
  - Streaming reads (read once) and append writes (write once) don't benefit much from caching at client

Hadoop Distributed File system

HDFS, the Hadoop Distributed File System, is a distributed file system designed (Figure 2) to hold very large amounts of data (terabytes or even petabytes), and provide high-throughput access to this information. Files are stored in a redundant fashion across multiple machines to ensure their durability to failure and high availability to very parallel applications.

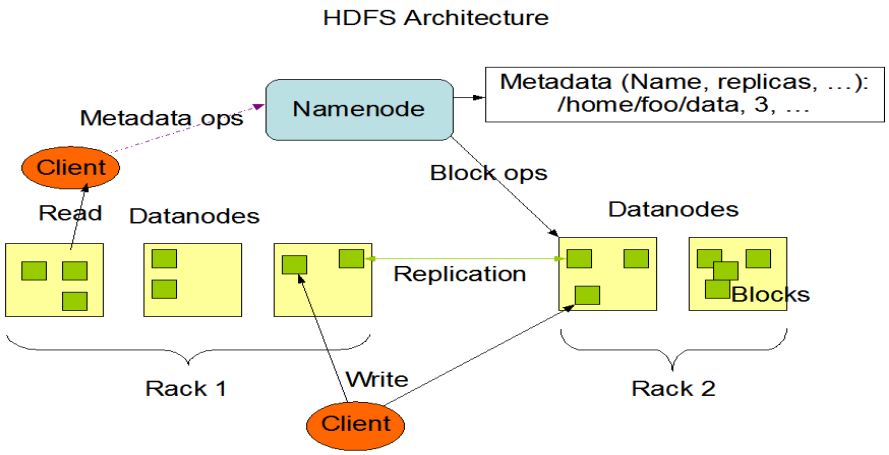


Figure 2 : Hadoop Distributed File System

HDFS has a master /slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manages storage attached to the nodes that they run on. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.[3]

MapReduce is also a data processing model. Its greatest advantage is the easy scaling of data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. In the mapping phase, MapReduce takes the input data and feeds each data element to the mapper. In the reducing phase, the reducer processes all the outputs from the mapper and arrives at a final result. In simple terms, the mapper is meant to filter and transform the input into something that the reducer can aggregate over.[4]

Before developing the MapReduce framework, Google used hundreds of separate implementations to process and compute large datasets. Most of the computations were relatively simple, but the input data was often very large. Hence the computations needed to be distributed across hundreds of computers in order to finish calculations in a reasonable time. MapReduce is highly efficient and scalable, and thus can be used to process huge datasets. When the MapReduce framework was introduced, Google completely rewrote its web search indexing system to use the new programming model. The indexing system produces the data structures used by Google web search. There is more than 20 Terabytes of input data for this operation. At first the indexing system ran as a sequence of eight MapReduce operations, but several new phases have been added since then. Overall, an average of hundred thousand MapReduce jobs is run daily on Google's clusters, processing more than twenty Petabytes of data every day. The idea of MapReduce is to hide the complex details of parallelization, fault tolerance, data distribution and load balancing in a simple library. In addition to the computational problem, the programmer only needs to define parameters for controlling data distribution and parallelism. Like Google's MapReduce, Hadoop uses many machines in a cluster to distribute data processing. The parallelization doesn't necessarily have to be performed over many machines in a network. There are different implementations of MapReduce for parallelizing computing in different environments.

Hadoop is a distributed file system that can run on clusters ranging from a single computer up to many thousands of computers. Hadoop was inspired by two systems from Google, MapReduce and Google File System.

Hadoop is good at processing large amount of data in parallel. The idea is to breakdown the large input into smaller chunks and each can be processed separately on different machines. That way, we can alleviate the IO bottleneck across many machines to achieve better overall performance. The infrastructure has abstracted you out from the complexity of distributed computing. So, the user has no worry of machine failure, data availability and coordination.

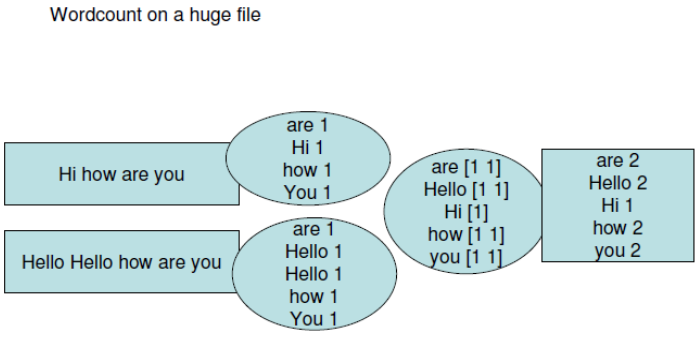


Figure 3:Map Reduce

GFS is a scalable distributed file system for data-intensive applications. GFS provides a fault tolerant way to store data on commodity hardware and deliver high aggregate performance to a large number of clients.

MapReduce is a toolkit (Figure 3) for parallel computing and is executed on a large cluster of commodity machines. There is quite a lot of complexity involved when dealing with parallelizing the computation, distributing data and handling failures. Using MapReduce allows users to create programs that run on multiple machines but hides the messy details of parallelization, fault-tolerance, data distribution, and load balancing. MapReduce conserves network bandwidth by taking advantage of how the input data is managed by GFS and is stored on the local storage device of the computers in the cluster.

MapReduce and the Hadoop File System (HDFS), which is based on GFS, what defines the core system of Hadoop. MapReduce provides the computation model while HDFS provides the distributed storage. MapReduce and HDFS are designed to work together. While MapReduce is taking care of the computation, HDFS is providing high throughput of data. Hadoop has one machine acting as a NameNode server, which manages the file system namespace. All data in HDFS are split up into block sized chunks and distributed over the Hadoop cluster. The NameNode manages the block replication. If a node has not answered for some time, the NameNode replicates the blocks that were on that node to other nodes to keep up the replication level.

Most nodes in a Hadoop cluster are called DataNodes; the NameNode is typically not used as a DataNode, except for some small clusters. The DataNodes serve read/write requests from clients and perform replication tasks upon instruction by NameNode. DataNodes also run a TaskTracker to get map or reduce jobs from JobTracker. The JobTracker runs on the same machine as NameNode and is responsible for accepting jobs submitted by users. The JobTracker also assigns Map and Reduce tasks to Trackers, monitors the tasks and restarts the tasks on other nodes if they fail.

MapReduce

MapReduce, based on the LISP map and reduce primitives, was created as a way to implement parallel processing without having to deal with all the communication between nodes,and distribution of tasks[2], like, for example, MPI. MapReduce programming consists of writing two functions, a map function, and a reduce function. The map function takes a key,

value pair and outputs a list of intermediate values with the key. The map function is written in such a way that multiple map functions can be executed at once, so it's the part of the program that divides up tasks. The reduce function then takes the output of the map functions, and does some process on them, usually combining values, to generate the desired result in an output file.

Figure 4 below shows a picture representing the execution of a MapReduce job.[8]

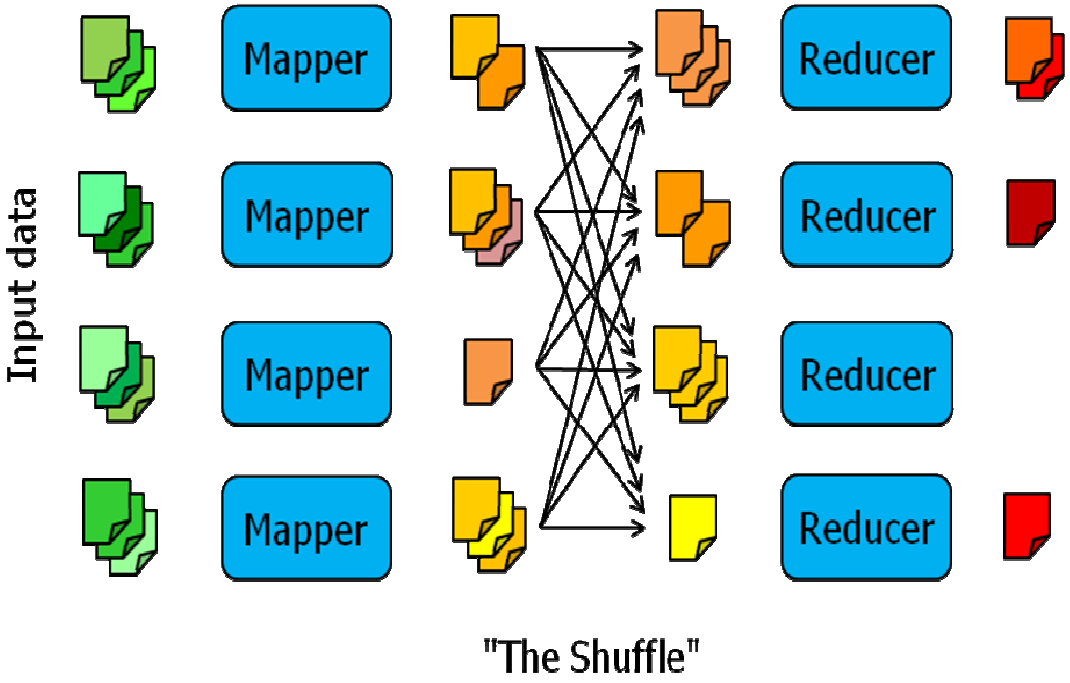


Figure 4 : Map Reduce Job

When a MapReduce program is run by Hadoop, the job is sent to a master node, the jobtracker, which has multiple "slave" nodes, or tasktrackers that report to it and ask for new work whenever they are idle. Using this process, the jobtracker divides the map tasks (and quite often the reduce tasks as well) amongst the tasktrackers, so that they all work in parallel. Also, the jobtracker keeps track of which tasktrackers fail, so their tasks are redistributed to other task trackers, only causing a slight increase in execution time. Furthermore, in case of slower workers slowing down the whole cluster, any tasks still running once there are no more new tasks left are given to machines that have finished their tasks already. Not every process nodes have a small piece of a larger file, so that when a file is accessed, the bandwidth of a large number of hard disks is able to be utilized in parallel. In this way, the performance of Hadoop may be able to be improved by having the I/O of nodes work more concurrently, providing more throughput.

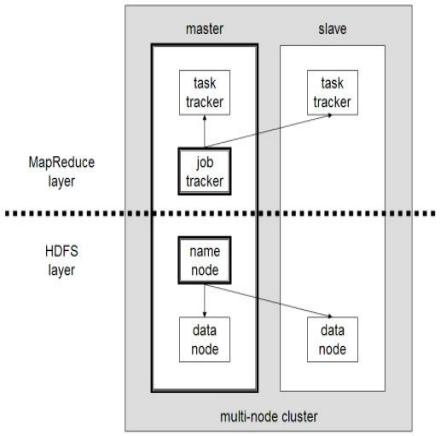


Figure 5: Hadoop Cluster Architecture

Map Reduce works in the following manner in below 7 tasks:-

1. The Map-Reduce library in the user program first splits the input into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece. It then starts up many copies of the program on a cluster of machines.(Refer Figure 5)

2. One of the copies of the program is special- the master copy. The rest are workers that are assigned work by the master. There are M map task and R reduce tasks to assign; the master picks idle workers and assign each one a task

3. A worker who is assigned a map task reads the contents of the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk partitioned into R regions by the partitioning function. The location of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers

5. When a reduce worker is modified by the master about these locations.it uses remote procedure calls to read buffered data from the local disk of map workers. When a reduce worker has read all intermediate data,it sorts it by the intermediate keys. The sorting is needed because typically many different key map to the same reuce task.

6. The reduce worker iterate over the sorted intermediate data and for each unique key encountered, it passes the key and the correspnding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to the final output file for this reduce partition

7. When all map task and reduce task have been completed, the master wakes up the user program. At this point, the Amp-Reduce call in the user program returns back to the user code.

Diagrammatic Representation of Map Reduce

One round of MapReduce computation consists of the following 3 steps:-

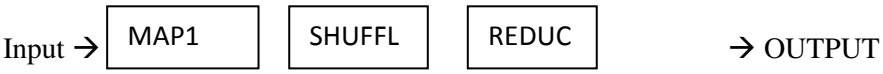


Figure 6: Diagrammatic Representation of Map Reduce

In this manner the map reduce works, one round of MapReduce computation consists of 3 steps

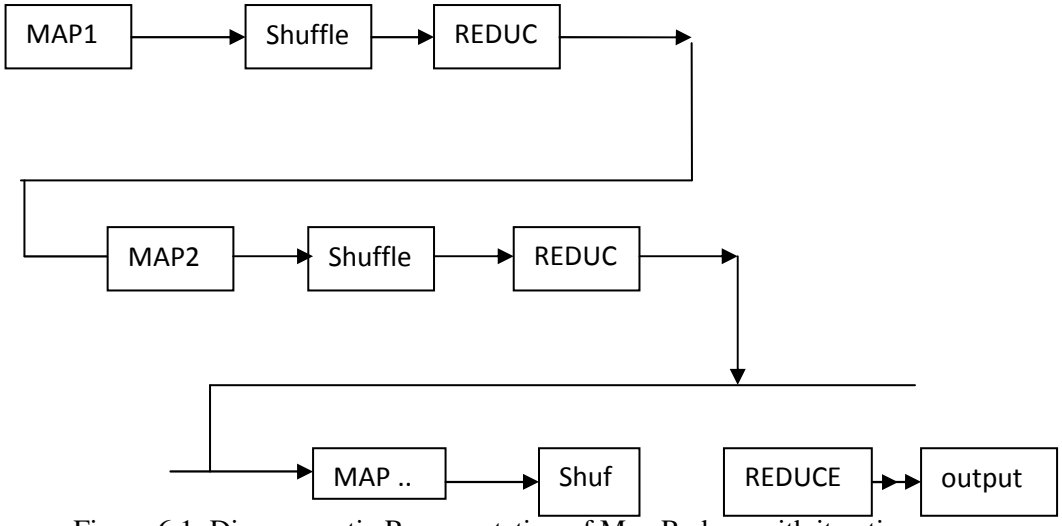


Figure 6.1: Diagrammatic Representation of Map Reduce with iterations

Theoretical Representation of Map Reduce (Fig 6.1)

1. Data are represented as a <key, value> pair
2. Map: <key, value> → multiset of <key, value> pairs user defined, easy to parallelize
3. Shuffle: Aggregate all <key, value> pairs with the same key. executed by underlying system
4. Reduce: <key, multiset(value)> → <key, multiset(value)> user defined, easy to parallelize
5. Can be repeated for multiple rounds[9]

Map Reduce works as a Job Tracker and Task Tracker.

- **Map/Reduce Master "Jobtracker"**
  - ❖ **Accepts** Map-Reduce jobs submitted by users
  - ❖ **Assigns** Map and Reduce tasks to Tasktrackers
  - ❖ **Monitors** task and tasktracker status, re-executes tasks upon failure
- **Map/Reduce Slaves "Tasktrackers"**
  - ❖ **Run** Map and Reduce tasks upon instruction from the Jobtracker
  - ❖ **Manage** storage and transmission of intermediate output.

Job tracker functions in the following manner:-

- Handles all jobs
- Makes all scheduling decisions
- Breaks jobs into tasks, queues up
- Schedules tasks on nodes close to data
  - ❖ Location information comes from InputSplit
- Monitors tasks
- Kills and restarts tasks if they fail/hang/disappear

Task tracker works in the following manner:-

- Asks for new tasks, executes, monitors and reports status

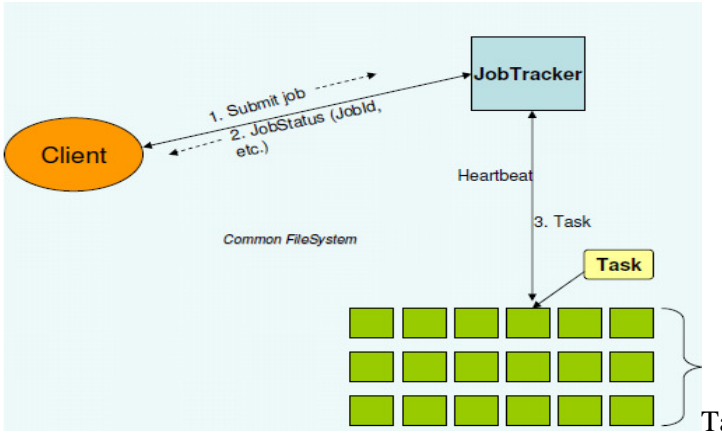


Figure 7 : Parallel MapReduce computations

The programmer can be mostly oblivious to parallelism and distribution; the programming model readily enables parallelism, and the MapReduce implementation takes care of the complex details of distribution such as load balancing, network performance and fault tolerance. The programmer has to provide parameters for controlling distribution and parallelism, such as the number of reduce tasks to be used which is described in the later part of this paper by referring the example. (Figure 6) Defaults for the control parameters may be inferable. In this section, I have made the clarification on the opportunities for parallelism in a distributed execution of MapReduce computations.

Opportunities for parallelism

Parallel map over input: Input data is processed such that key/value pairs are processed one by one. It is well known that this pattern of a list map is amenable to total data parallelism. That is, in principle, the list map may be executed in parallel at the granularity level of single elements. Clearly, MAP must be a pure function so that the order of processing key/value pairs does not affect the result of the map phase and communication between the different threads can be avoided.

Parallel grouping of intermediate data: The grouping of intermediate data by key, as needed for the reduce phase, is essentially a sorting problem. Various parallel sorting models exist. If we assume a distributed map phase, then it is reasonable to anticipate grouping to be aligned with distributed mapping. That is, grouping could be performed for any fraction of intermediate data and distributed grouping results could be merged centrally, just as in the case of a parallel-merge-all strategy. Parallel map over groups: Reduction is performed for each group (which is a key with a list of values) separately. Again, the pattern of a list map applies here; total data parallelism is admitted for the reduce phase— just as much as for the map phase.

Parallel reduction per group: Let us assume that REDUCE defines a proper reduction.(Figure 7) That is, REDUCE reveals itself as an operation that collapses a list into a single value by means of an associative operation and its unit. Then, each application of REDUCE can be massively parallelized by computing sub-reductions in a tree-like structure while applying the associative operation at the nodes. If the binary operation is also commutative, then the order of combining results from sub-reductions can be arbitrary. Given that we already parallelize reduction at the granularity of groups, it is non-obvious that parallel reduction of the values per key could be attractive.

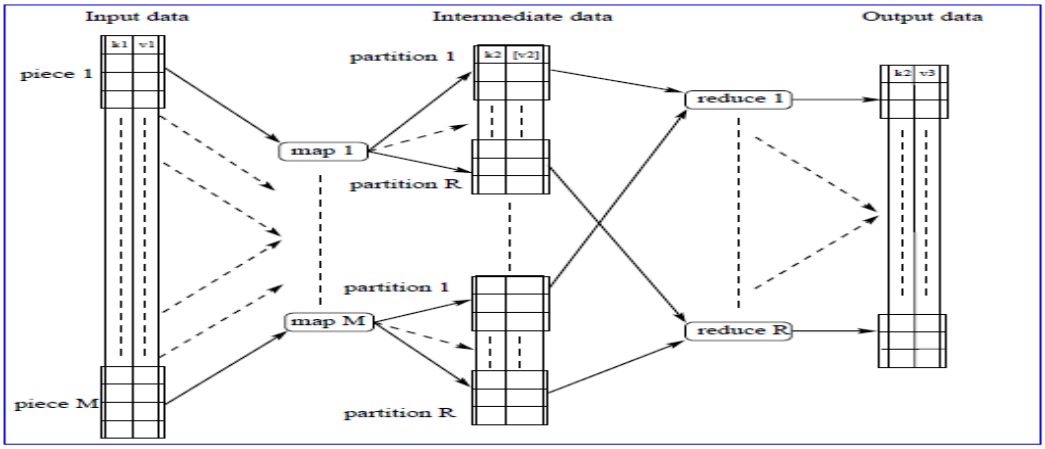


Figure 8: Map split input data and reduce partitioned intermediate data

Master/ Worker Relationship

- The MASTER:
  - ❖ initializes the array and splits it up according to the number of available WORKERS
  - ❖ sends each WORKER its sub-array

- ❖ receives the results from each WORKER

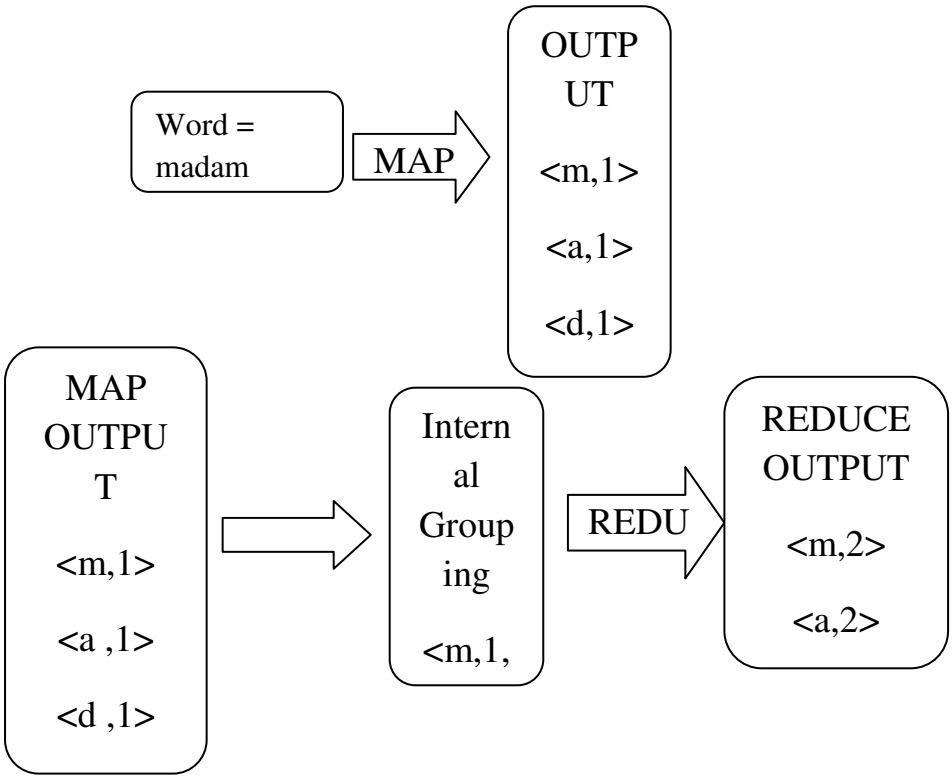
- The WORKER:
  - ❖ receives the subarray from the MASTER
  - ❖ performs processing on the subarray
  - ❖ returns results to MASTER

The Map Reduce programming is done in the following manner.

- Restricted parallel programming model meant for large clusters
  - ❖ User implements Map() and Reduce()
- Parallel computing framework (HDFS lib)
  - ❖ Libraries take care of EVERYTHING else (abstraction)
    - o Parallelization
    - o Fault Tolerance
    - o Data Distribution
    - o Load Balancing
- Useful model for many practical tasks[10]

Finding the occurrence of a letter in a string by using Map Reduce

- o ***map*** **(apply function to all items in a collection) and**
- o ***reduce*** **(apply function to set of items with a common key)**
- o A user-defined function to be applied to all data, map: (key,value) → (key, value)
- o Another user-specified operation reduce: (key, {set of values}) → result
- o A set of *n* nodes, each with data
  - ▪ This data is collected by key, then *shuffled*, and finally *reduced*[11]



The above diagram shows the depiction of map-reduce example. The below given small snippet of code is written for the above depiction.

```
map( letter  value, String value)  {

//key : string, letter

//value : contents of string

for each letter in value :

        Emit(letter , "1")
```

```
reduce (letter key, iterator values)  {

// key: a letter

//values: a list of counts of a letter
in string

int result = 0;

For each v in values :

        result  += ParseInt(v);
```

In this manner we come to know about the working of map-reduce example through the diagram above given and the code. Here the counting of letter 'm' or 'a' or 'd' , this task is executed on parallel machines. By using all similar characters are mapped and then reduce operation is performed to count how many times each character is occurring.

In this paper, the author has explained the working of parallel processing. The author has illustrated the working of Map Reduce framework in Hadoop Distributed File System. The Map Reduce framework simplifies the complexity of running distributed data processing functions across multiple nodes in a cluster in a parallel manner. Author has clarified the role of the Map Reduce with the help of an example for finding the occurrences of a character when a string is entered. Map Reduce allows a programmer with no specific knowledge of distributed parallel programming to create the Map Reduce functions running in parallel across multiple nodes in the cluster.
The fault tolerance feature is implemented by the Map Reduce by using Replication. Hadoop achieves fault tolerance by means of data replication. More importantly, the Map Reduce platform can offer fault tolerance that is entirely transparent to programmers. The author wants to study in the future on the performance related issues when this Map Reduce technique is used on multiple nodes

References

[1] http://developer.yahoo.com/hadoop/tutorial/module3.html
[2] Google's MapReduce Programming Model—Revisited_Ralf L¨ammel
[3]HDFS Architecture Guide
[4]Hadoop in Action – Chuck Lam
[5] Introduction to Hadoop - Dr. G Sudha Sadhasivam,PSG College of Technology Coimbatore
[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
[7] Review of Distributed File Systems: Concepts and Case Studies ECE 677 Distributed Computing Systems
[8] Levy E. and Silberschatz A., "Distributed FileSystems: Concepts and Examples"
[9] Yahoo Research-http://dimacs.rutgers.edu/Workshops/Parallel/slides/suri.pdf
[10] Source from AIAA 2011: Survey of Parallel Data Processing in Context with MapReduce by Madhavi Vaidya
[11]http://ieeexplore.ieee.org: Improving MapReduce performance through data placement in heterogeneous Hadoop clusters

List of Abbreviations

HDFS- Hadoop Distributed File Systems

GFS- Google File Systems

DFS – Distributed File Systems