Map Reduce on Red Green Blue Architecture

Lacine KABRE
Joseph Ki-Zerbo University,
Mathematical and computer analysis laboratory,
Ouagadougou, BURKINA FASO

Telesphore TIENDREBEOGO,
NAZI BONI University, Bobo Dioulasso,
Bobo Dioulasso,
BURKINA FASO

Abstract:- In massive data processing, platforms using MapReduce are designed for data centers, which are generally centralized.These platforms typically rely on a single node to maintain and coordinate MapReduce tasks, leading to a single point of failure. Our aim in this paper has been to propose a model for MapReduce computation on the Red Green Blue architecture, which is a decentralized, triple-node big data architecture. This architecture is based on the peer-to-peer networking protocol named Content Addressable Network. First, we implemented all the steps of the MapReduce computation approach, taking into account the properties of the Content Addressable Network protocol and the Red Green Bluearchitecture. We then carried out an experiment in a local network to evaluate performance in terms of processing speed and time. The experiment showed that latency decreased with the number of compute nodes. This study not only showed that the Red Green Blue architecture is viable as a massive data processing architecture, but also improved processing times as a function of network nodes. The robustness, scalability and lack of a single point of failure of the Red Green Bluearchitecture mean that MapReduce can be easily deployed in a wider variety of applications.

Keywords*:- P2P protocol, Map Reduce, RGB architecture, Big data Storage.*

I. INTRODUCTION

Over the past twenty years, the amount of data generated has only increased. Currently, we produce a very large mass of data every year, estimated at nearly 3 trillion (3,1018) bytes of data. It is estimated that in 2016, 90% of the world's data was created in the previous two years. Database Management System (DBMS) have been criticized for their monolithic architecture, which makes them "heavy" and costly to operate **[1]**.It is sometimes argued that they are inefficient for many data management tasks, despite their success in enterprise data processing. This has been dubbed the "big data problem". And today the term "big data" is used to designate this phenomenon of high-volume, diverse data with a velocity that is becoming increasingly difficult to control. Whereas early DBMS focused on modeling the operational characteristics of companies, "big data" systems are now geared towards modeling user behavior by analyzing vast quantities of interaction logs. In view of the sheer volume of data involved, several solutions have been put forward to restructure DBMS **[2]**, but the basic architecture has not changed dramatically. With the increase in data quantity and the availability of high-performance, relatively inexpensivehardware, database systems have been extended and parallelized to run on multiple hardware platforms **[3]**. Recently, a new distributed data processing framework called MapReduce has been proposed **[4]**, whose fundamental idea is to simplify parallel processing using a distributed computing platform that offers only two interfaces: map and reduce.

Hadoop**[5]** is the most popular framework implementing the map reduce computation model. Several other technologies developed around Hadoop make it efficient and easy to deploy in data centers **[6].** Typically, Hadoop is deployed on its HDFS (Hadoop Distributed File System) file system for greater efficiency. But Hadoop's architecture is centralized. Its main node, called the name node, is responsible for all the other nodes, called the data node. This central node constitutes a point of failure, as its unavailability brings the platform to a complete halt until it is restored **[7].**

In this research work, we propose the possibility of deploying MapReduce on a Big Data architecture. This is the Red Green Bleu(RGB) architecture, which uses the CAN distributed hash table and its network management methods [8]. Our contribution in this paper was first to define the components of MapReduce and how it integrates with the RGB architecture that constitutes the storage layer in our context, and then to build a prototype implementing MapReduce execution. We then deployed our system in a local environment and carried out an experiment by running a word counting process on a file stored on RGB architecture before concluding with an evaluation of the results, which show that our architecture is viable in a big data processing context.

II. MAP REDUCE PROGRAMMING MODEL

In 2004, Google published an article proposing a solution for processing large-scale analytical operations on a large cluster of servers: the MapReduce computing model. In this approach, users implement their own mapping and reduction functions, while the system is responsible for scheduling and synchronizing mapping and reduction tasks [4]. MapReduce is increasingly used in applications such as data mining, data analysis and scientific computing. Its widespread adoption and success are based on its distinctive features [1], which can be summarized as follows:

- **Flexibility**: it lets you write your own calculation and sorting methods on a variable amount of data **[1]**.
- **Scalability:** the MapReduce processing system can dynamically increase or decrease its performance as computing needs change **[1]**.
- **Efficiency**: MapReduce doesn't need to load data into a database, which usually entails high costs. It is therefore

highly efficient for applications that require data to be processed once (or only a few times)**[1]**.

- **Fault tolerance**: In MapReduce, each job is divided into several small tasks which are assigned to different machines**[1]**.

In the Map Reduce computation model, a computation takes a set of *key/value* pairs as input, and produces a set of *key/value* pairs as output **[4]**. MapReduce is based on two main methods: Map and Reduce. The Map method, written by the user, takes a data pair as input and produces a set of intermediate *key/value* pairs. MapReduce groups all intermediate values associated with the same key and passes them onto the Reduce function. The Reduce function, also written by the user, accepts an intermediate key and a set of values for this key. It merges these values to form a possibly smaller set of values. As a general rule, each invocation of the Reduce function produces only zero or one output value. Intermediate values are supplied to the user's reduction function via an iterator **[9]**. This allows us to manage lists of values that are too large and could saturate memory. Many different implementations of MapReduce exist, depending on the environment. Forexample, one can be implemented on a small shared-memory machine and another for a set of networked machines **[4]**. The following algorithms represent examples of pseudo code for Map and Reduce functions, and Fig.1 shows an illustration of MapReduce execution.

ALGORITHM 1: Map Function for UserVisits

```
input: String key, String value
1 String[] array = value.split("|");
2 EmitIntermediate(array[0],ParseFloat(array[2]));
```

ALGORITHM 2: Reduce Function for UserVisits

```
input: String key, Iterator values
1 float totalRevenue = 0;
2 while values.hasNext() do
3 |   totalRevenue += values.next();
4 end
5 Emit(key, totalRevenue);
```

In a MapReduce computing system, tasks are sent to worker nodes via the scheduling module. Each worker node is responsible for a mapping or reduction process **[1]**. The basic implementation of the MapReduce engine must include the following modules (marked by gray boxes in Figure 1).

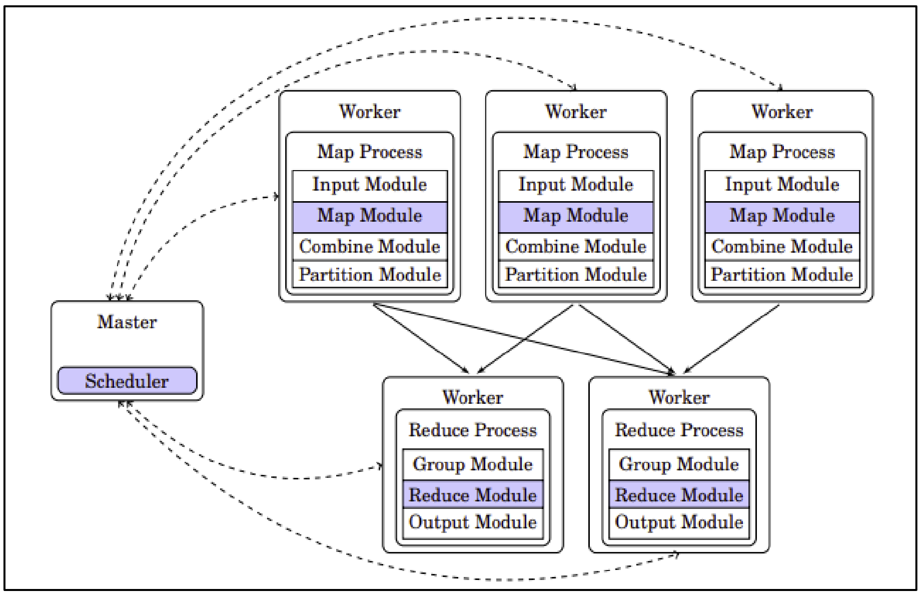


Fig. 1: Map Reduce Architecture [1]

- The *"Scheduler"* module is responsible for assigning Map and Reduce tasks to compute nodes (also known as worker nodes), based on data location, network status and other node statistics. It also controls fault tolerance by rescheduling a failed process to other worker nodes (if possible). "Scheduler" design significantly affects MapReduce system performance.
- The *"Map"* module analyzes a block of data and invokes the user-defined Map function to process the input data. After generating intermediate results (a set of *key/value* pairs), it groups the results based on partition keys, and notifies the master node of the position of the results.
- The *"Reduce"* module extracts data from the maps after receiving notification from the master node once all intermediate results have been obtained. The *"Reduce"* module merges the data by key and all values with the same key are grouped together. Finally, the user-defined function is applied to each *key/value* pair and the results are transmitted to the master node.

III. RED GREEN BLUE ARCHITECTURE

Today, the ever-increasing volumes of data from a variety of sources have given rise to a new range of technologies and architecture models. These so-called Big Data architectures are data pipelines capable of collecting, storing and processing data. Most often, data is stored and processed in semi-structured and unstructured formats. However, well-known architectures such as Lamda architecture, Zeta architecture and Iot architecture have a centralized data lake, while architectures such as microservice architecture and kappa have several databases forming their data lake. In our previous work, we proposed an architecture model that takes advantage of the aforementioned architecture models and overcomes the limitations of centralized data lake models. This architecture, called RGB architecture, is a model that uses the decentralized peer-to-peer network (CAN) protocol based on distributed hash tables**[8]**.In addition, we carried out a comparative study of the peer-to-peer network protocols Chord, CAN, Pastry and Kademlia, which showed that the CAN**[10]** protocol had a slightly higher routing time than the others, but ensured more efficient message transmission **[11]**. Storage operations with the CAN protocol therefore have a good success rate **[11].**

The RGB architecture is a triple-node architecture determined by the image properties Red Green Blue (RVB)**[13]** and based on the properties of the CAN protocol for information routing and node management. The figure (Fig.3) shows the conceptual model of the architecture. In the RGB architecture, we have a primary bootstrap node and three secondary nodes (R, G, B), called bootstrap secondary nodes, which are connected to other nodes for data storage. In the context of the map reduce computational model implementation, the data nodes play the role of worker nodes(see image Fig.10).

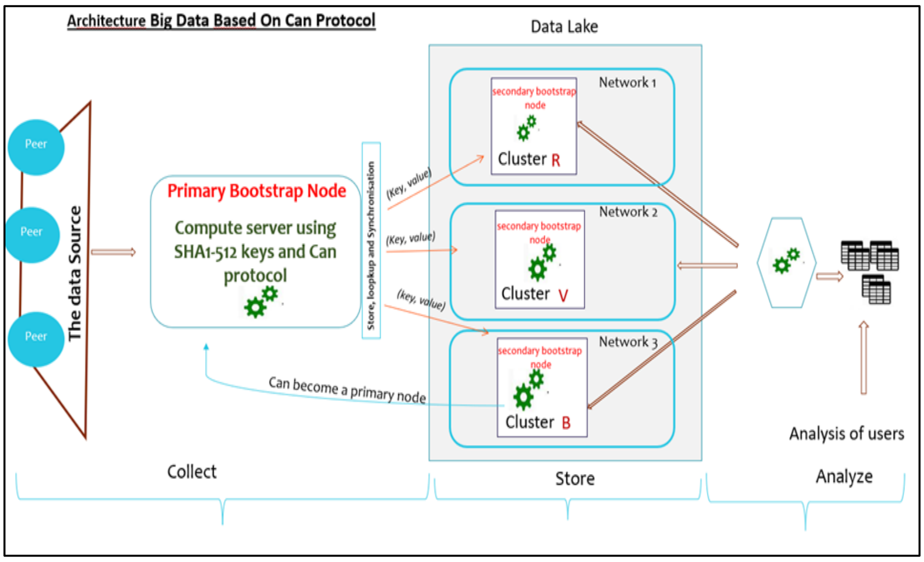


Fig. 2: Red Green Blue Architecture [8]

IV. THE OPERATING PRINCIPLE OF RGB ARCHITECTURE

A. The storage of Big Data objects

Object storage is an unstructured data storage technique. It therefore enables massive data to be stored with a unique identifier enabling the object to be located [3]. In the RGB architecture, an SHA-3 key is used to generate a unique identifier for each object. This identifier is called the Global Unique IDentifier (GUID). This 512-bit key is associated with the object to be stored, and the key/value pair is written to the CAN protocol hash table**[12]**.. The key is then divided into eight 64-bit subkeys. Each 64-bit sub-key is also associated with the object to be stored. Each key is then broken down into two parts of 24 bits and 40 bits. The 24 bits are used to determine the data peer cluster that will contain the data to be stored. The calculation process is detailed in [8].The algorithm 1 is a pseudo code of the calculation process.

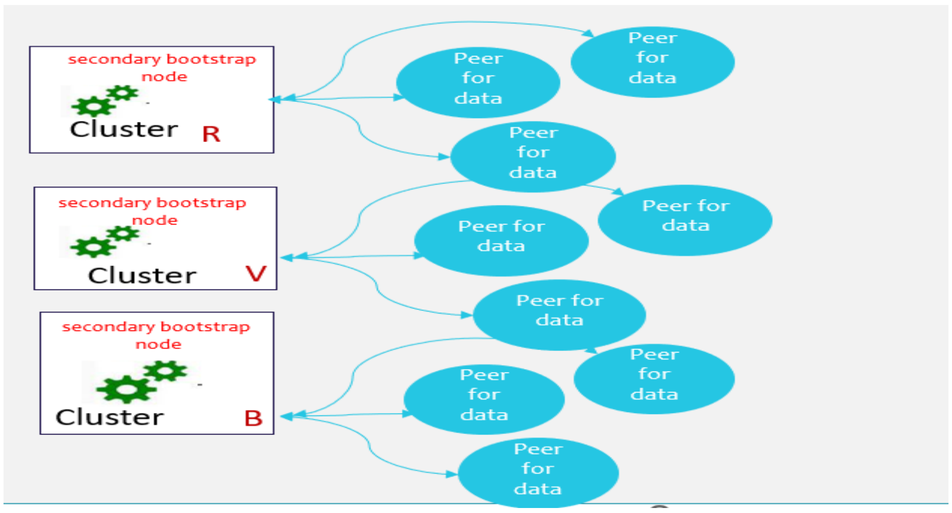
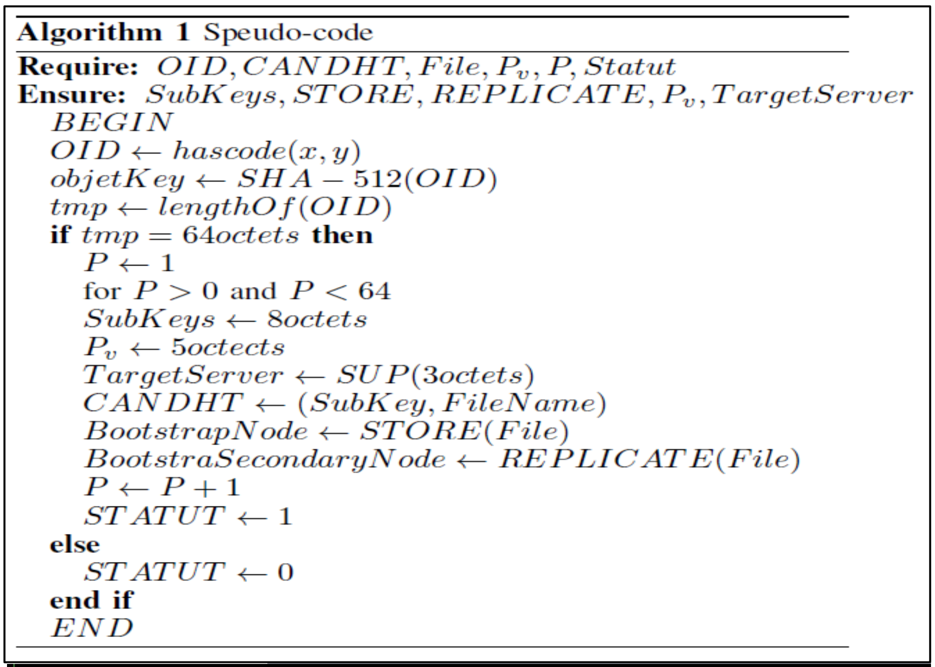
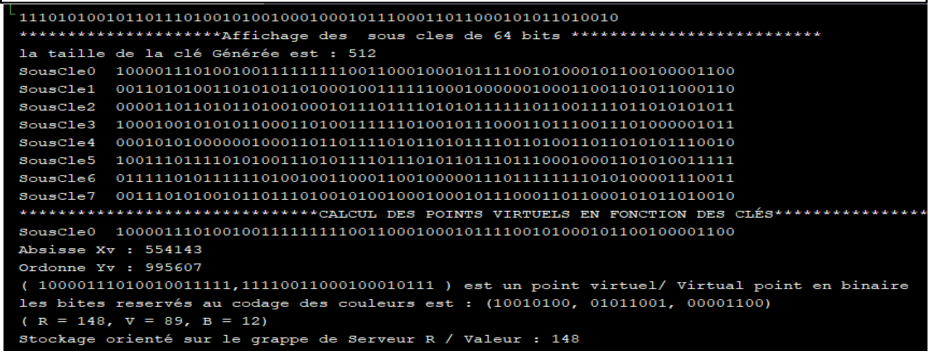


Fig. 3: Architecture of secondary bootstrap node [8]

Figure (Fig.5) is a screenshot showing the calculation of keys during a storage request in the RGB architecture. This is the result of implementing algorithm 1.



```
Algorithm 1 Speudo-code
Require: $OID, CANDHT, File, P_v, P, Statut$
Ensure: $SubKeys, STORE, REPLICATE, P_v, TargetServer$
  $BEGIN$
  $OID \leftarrow hascode(x, y)$
  $objetKey \leftarrow SHA-512(OID)$
  $tmp \leftarrow lengthOf(OID)$
  if $tmp = 64octets$ then
    $P \leftarrow 1$
    for $P > 0$ and $P < 64$
    $SubKeys \leftarrow 8octets$
    $P_v \leftarrow 5octects$
    $TargetServer \leftarrow SUP(3octets)$
    $CANDHT \leftarrow (SubKey, FileName)$
    $BootstrapNode \leftarrow STORE(File)$
    $BootstraSecondaryNode \leftarrow REPLICATE(File)$
    $P \leftarrow P + 1$
    $STATUT \leftarrow 1$
  else
    $STATUT \leftarrow 0$
  end if
  $END$
```



```
11101010010110111010010100100010001011100011011000101011010010
*********************Affichage des  sous cles de 64 bits **************************
la taille de la clé Générée est : 512
SousCle0  1000011101001001111111110011000100010111100101000101100100001100
SousCle1  0011010100110101011010001001111110001000000100011001101011000110
SousCle2  0000110110101101001000101110111101010111111011001111011010101011
SousCle3  1000100101010110001101001111110100101110001101110011101000001011
SousCle4  0001010100000001000110110111101011010111101101001101101010101110010
SousCle5  1001110111101010011101011110111010110111011100010001101010011111
SousCle6  0111110101111110100100110001100100000111011111111010100001110011
SousCle7  0011101010010110111010010100100010001011100011011000101011010010
******************************CALCUL DES POINTS VIRTUELS EN FONCTION DES CLÉS****************
SousCle0  1000011101001001111111110011000100010111100101000101100100001100
Absisse Xv : 554143
Ordonne Yv : 995607
( 10000111010010011111,11110011000100010111 ) est un point virtuel/ Virtual point en binaire
les bites reservés au codage des couleurs est : (10010100, 01011001, 00001100)
( R = 148, V = 89, B = 12)
Stockage orienté sur le grappe de Serveur R / Valeur : 148
```

Fig. 4: Calculating storage keys

B. Dynamics of nodes

One of the design objectives of the RBG architecture is to have an almost totally autonomous, self-managed system with high fault tolerance. In its operation, the secondary bootstrap nodes are synchronized with the primary bootstrap node **[8]**. The first secondary node to detect the unavailability of the primary node will replace it while waiting for it to recover. As a result, the system will continue to operate and storage will be carried out on the other two remaining nodes. Adding and removing data pairs is managed by the CAN protocol **[8]**.

V. DATA SECURITY LAYER IN RGB ARCHITECTURE

RGB architecture introduces data-centric security. Search operations require verification of all subkeys associated with the object being searched. During storage, a coordinate point (x,y) is chosen in the two-dimensional virtual space of CAN. Applying the SHA-3 encryption function to this point provides the object's GUID, which in turn is split into 8 subkeys. The sub-key space is called the virtual image space. Figure Fig.6 illustrates the association between the CAN image space and the virtual image space.

Let *Pu*, the CAN unique point, SHA-3 the encryption function, the object identifier is verified by the following formula

$$SHA3(Pu) = \sum_{i=1}^{8} (P'_i) \tag{1}$$

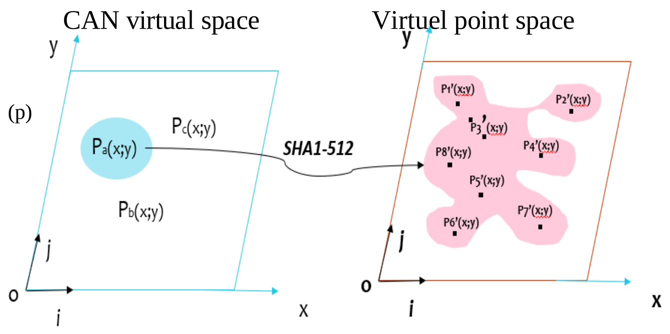


Fig. 5: The image shows the association of an SHA3 key with a virtual point space

To reinforce the security of data in transit, FPE**[17]** technology is used in addition to the verification of keys identifying objects. This technology preserves data integrity from capture to storage in the data lake. The application of these techniques results in a data lake with reliable data**[17].**

VI. RESULT ON LOOKUP OPERATIONS

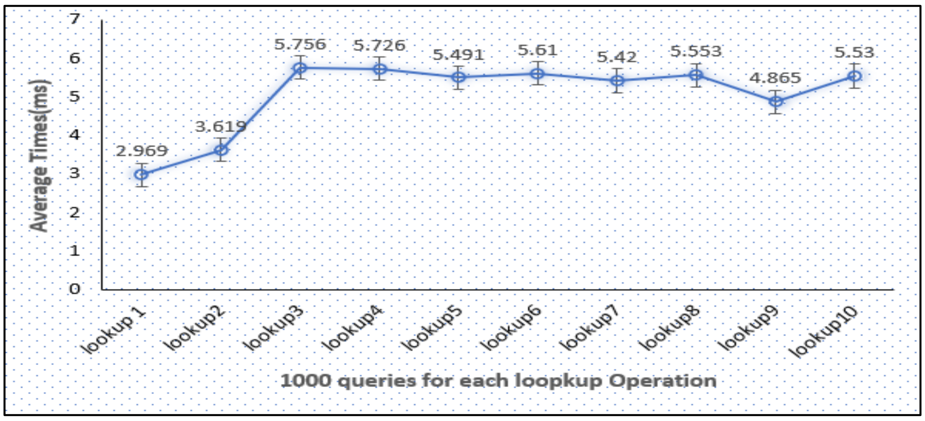


Fig. 6: Average Time for LOOKUP Operations

The performance of search and storage operations has been evaluated[8]. The figure shows the evolution of latency times for search requests. For 1000 store requests (STORE request), we generate around 1000*8 keys in the hash table. For search requests (LOOKUP request), the operation was carried out several times and the average search times were calculated using the following formula: the calculation of the average time taken for the searches is done with the following formula:

$$\boldsymbol{TM_L} = \frac{\sum_{i=1}^{n}(\mathbf{T}_i)}{\boldsymbol{N_L}}; \; ; \; n = 1000 \tag{2}$$

N_L is the total number of LOOPUK requests and Tiis the time for lookup.At the start of the experiment, latency was 2.9 ms (milliseconds). A peak of 5.7 ms was also observed. But as we repeated the experiment, the latency time dropped slightly to 4.9 ms.

VII. MAP REDUCE AND RGB ARCHITECTURE

Today. several technologies are designed for massive data processing and adapt to cloud architectures. Their architecture relies on several nodes with specific roles to coordinate task execution. These nodes perform scheduling and distribution tasks and contribute to network fault tolerance. However, the failure of individual nodes has a major impact on the overall system.Our MapReduce implementation on the RGB Architecture provides a dynamic framework for MapReduce, and is ableto be running on any arbitrary distributed configuration. Our framework exploits the characteristics of CAN**[12]** distributed hash tables coupled with our color-codedcomputing approach to manage distributed file storage, fault tolerance and data retrieval**[8]**.

Our approach to implementing MapReduce has been to develop modules as extensions to the CAN protocol, taking advantage of existing functionality. By treating each task as a data object, we can distribute them in the same way as files, relying on the protocol to route them and ensure their robustness.

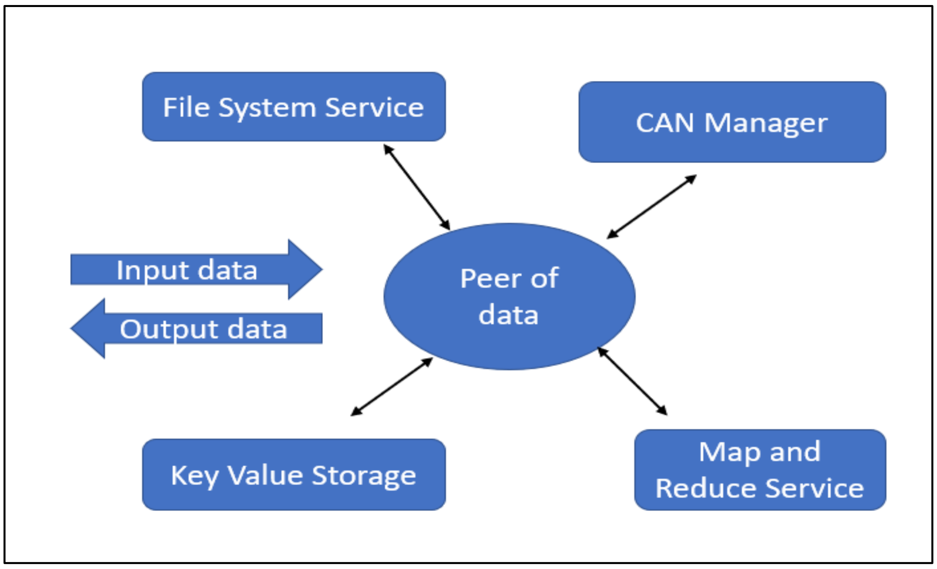


Fig. 7: Basic Architecture of node on RGB Architecture

VIII. MAP REDUCE AND DATAFLOW

In the implementation of MapReduce in the RGB architecture, each data peer ("worker") will perform the tasks and send its result to the "secondary bootstrap node". Note that these peers of data are also nodes in a P2P network using the CAN protocol. They therefore retain their master slave status.To launch the MapReduce process, the user contacts the primary bootstrap node. The primary bootstrap node first identifies the file using the RGB system search algorithms on one of the clusters (R or G or B) before delegating full management of all MapReduce steps to it.

The "secondary bootstrap node" that receives a computation request will first select a set of nodes to perform the tasks, also known as "jobs". This stage corresponds to an initialization process, in which the secondary bootstrap node confirms, on the basis of its routing table, the availability of the nodes it has chosen to perform the calculations.

The secondary bootstrap node's job is to divide the data file into smaller units called "data atoms" and then send each data unit to its elected peers. In this way, each compute node performs the job on the data unit assigned to it. The results are stored as a new data atom, which is then sent to the secondary bootstrap node. This operation takes $(d/4)(n^{1/d})$ hops according to the routing algorithms of the CAN**(12)** protocol, with ***d*** the dimension of the space and ***n*** the number of nodes. The data atom can be a block of text, the result of a summation or a subset of the elements to be processed. MapReduce functions are applied to these data atoms.

For a particular intermediate value, or a subset of elements to be sorted, the details of the "job" distribution are defined by the user.

For a user who wants to run a MapReduce job on data stored in the RGB architecture, the primary bootstrap node locates the data file using its key, as described in **[8],** and sends the command to the secondary bootstrap node, which becomes fully responsible for executing the MapReduce job. The secondary node have the role of JobTracker. It coordinates the execution of the jobs and returns the result to the primary node.

In our design, elected peers must finish executing a task before they want to leave the network. To avoid losing an atom of data, a timeout is assigned to each task. If a task is not completed when the timeout expires, the neighboringnode takes over the task and it is deleted from the node that was in charge of execution. In effect, each node also maintains the state of its neighbor.

When the delay expires, the peer who voluntarily leaves the network overloads his neighbor with the data atom for which he was responsible. The neighbor determination mechanism is based on the CAN protocol **[12]**. One of the advantages of our system is its ease of development. The user doesn't get involvedthe uniform distribution of tasks, nor about the failure of a network node. If a node fails during an operation, its task is reassigned to another. This makes the system extremely robust during runtime. For this architecture, all a developer has to do is write the Map and Reduce functions, which define how to divide the work to be done into portions and the task to be performed on each portion to obtain results.

IX. EXPERIMENTATION AND DEPLOYMENTS

A. *Experimentation*

The aim of this paper is to show that our architecture is viable for processing massive data using the MapReduce model. To this end, we will carry out an experiment in a local network and seek to evaluate the following aspects:

- speed of execution of Jobs submitted by compute nodes;
- The evolution of calculation times as a function of calculation nodes;
- Execution speed as a function of the number of jobs submitted.

The acceleration or speed of computation can be demonstrated by showing that a "Job" distributed to several nodes runs faster than when assigned to a single node. In other words, show that $\exists n$ $Tn < T1$, where Tn is the time, it takes n nodes to complete a "Job". To establish scalability, we need to show that the cost (in terms of time) of distributing work grows logarithmically. Also, we need to show that the larger the "Jobs" to be completed, the higher the number of nodes if we want to achieve a low execution time. To estimate the execution time, we use the formula **[15]**:

$$T_n = \frac{T_1}{n} + k * \log_2(n) \qquad (3)$$

T_1/n is the time the job would take if it was distributed in an ideal universe and *k*log2(n)* is the network time, *k* being an unknown constant depending on network latency. For the purposes of this article, our experiment is based on counting the number of words in a file. Fig.9 shows the steps in the calculation process.

B. *Deployments*

To evaluate the performance of our MapReduce implementation, we chose to deploy it on a local network. This implementation was entirely realized in Java using the java.net, File, Stream API and regular expressions. Our implementation implements all the routing and maintenance procedures defined by the CAN(12) protocol, which is used to implement the RGB(8) architecture. The machines used are configured on the Windows file system. Our implementation is therefore able to easily manipulate (create, read and write) files. To start the experiment, MapReduce commands and job descriptions are sent to the Primary Bootstrap Node, which performs a file search operation before transferring the commands to one of the secondary nodes. We tested our computing system by running a word frequency count. The tasks were tested in several configurations; we varied the initial network size and the size of the Jobs. Each Map job is defined by the number of nodes that must execute it, and produces a result that constitutes an input for the "Shuffle" process. Reducing these results involves adding up the respective fields. Our experiment counts the occurrence of each word in a file stored on the RGB**[8]** architecture.

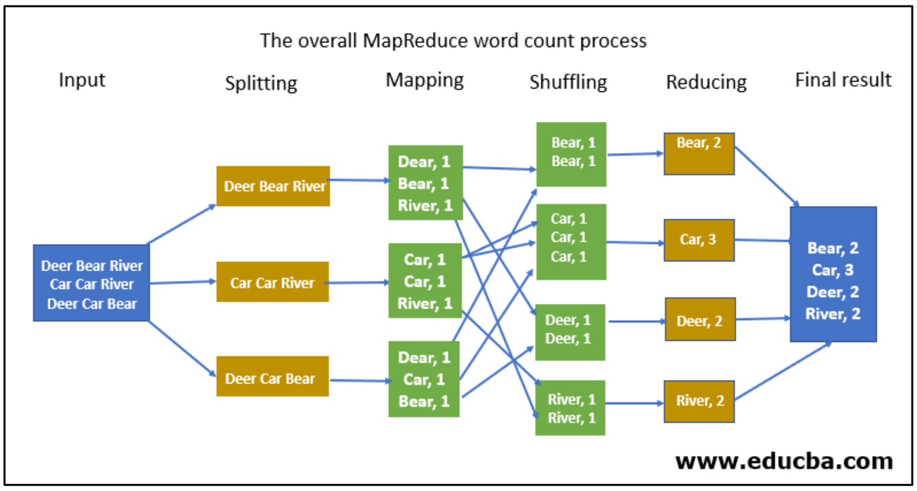
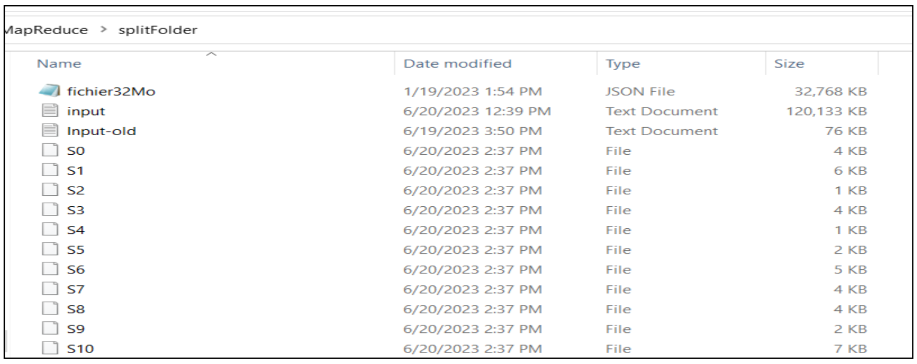


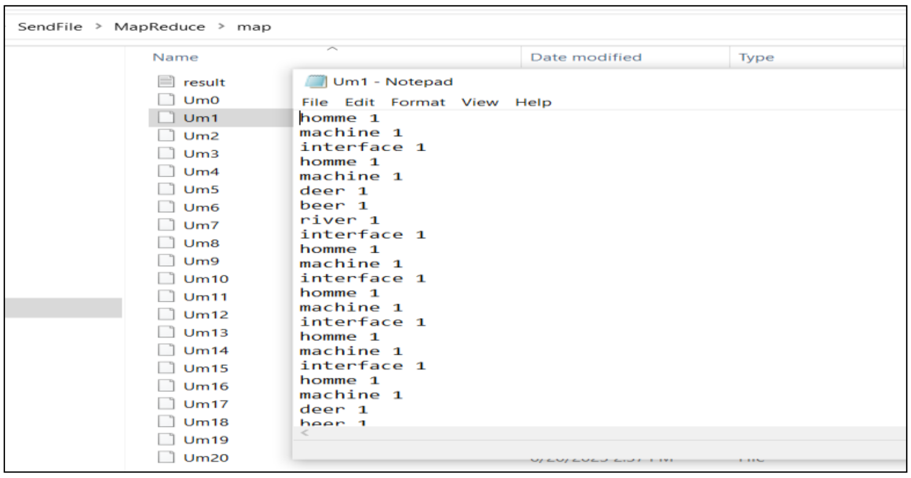
Fig. 8: Map Reduce processes for WordCount[16]



apReduce > splitFolder

Name	Date modified	Type	Size
fichier32Mo	1/19/2023 1:54 PM	JSON File	32,768 KB
input	6/20/2023 12:39 PM	Text Document	120,133 KB
Input-old	6/19/2023 3:50 PM	Text Document	76 KB
s0	6/20/2023 2:37 PM	File	4 KB
s1	6/20/2023 2:37 PM	File	6 KB
s2	6/20/2023 2:37 PM	File	1 KB
s3	6/20/2023 2:37 PM	File	4 KB
s4	6/20/2023 2:37 PM	File	1 KB
s5	6/20/2023 2:37 PM	File	2 KB
s6	6/20/2023 2:37 PM	File	5 KB
s7	6/20/2023 2:37 PM	File	4 KB
s8	6/20/2023 2:37 PM	File	4 KB
s9	6/20/2023 2:37 PM	File	2 KB
s10	6/20/2023 2:37 PM	File	7 KB

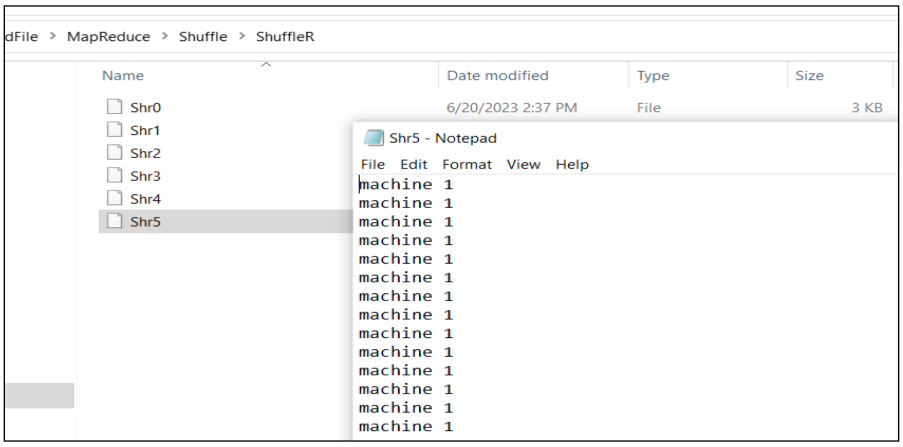
Fig. 9: Example of a file block created after a Splitting operation



result, Um0, Um1, Um2, Um3, Um4, Um5, Um6, Um7, Um8, Um9, Um10, Um11, Um12, Um13, Um14, Um15, Um16, Um17, Um18, Um19, Um20

```
homme 1
machine 1
interface 1
homme 1
machine 1
deer 1
beer 1
river 1
interface 1
homme 1
machine 1
interface 1
homme 1
machine 1
interface 1
homme 1
machine 1
interface 1
homme 1
machine 1
deer 1
beer 1
```

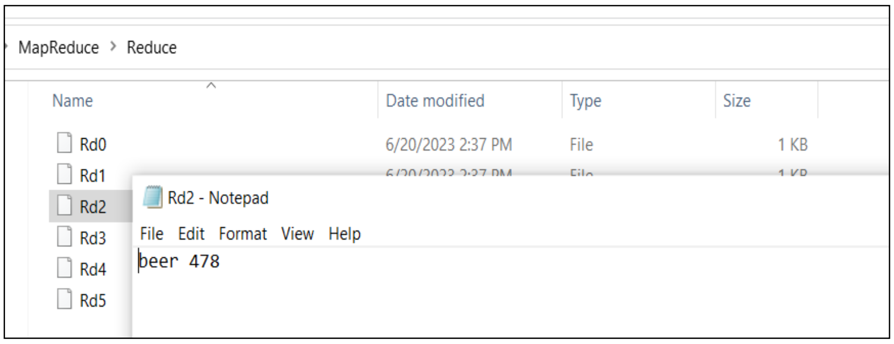
Fig. 10: Example of a file created after running Map


dFile > MapReduce > Shuffle > ShuffleR
Shr5 - Notepad
File Edit Format View Help

Name	Date modified	Type	Size
Shr0	6/20/2023 2:37 PM	File	3 KB
Shr1			
Shr2			
Shr3			
Shr4			
Shr5			

```
machine 1
machine 1
machine 1
machine 1
machine 1
machine 1
machine 1
machine 1
machine 1
machine 1
machine 1
machine 1
machine 1
machine 1
```

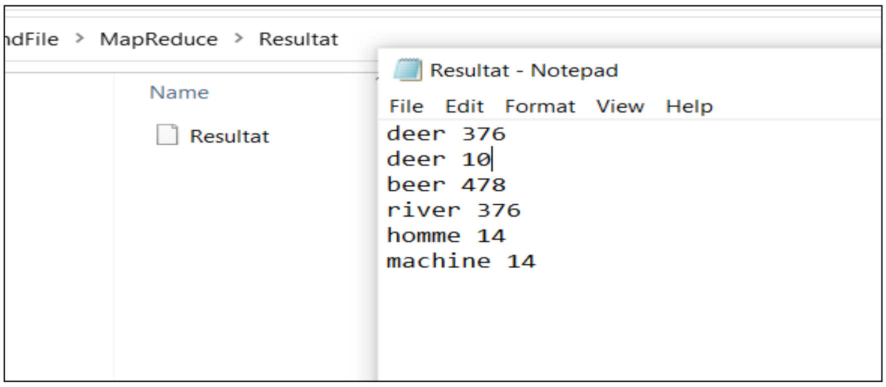
Fig. 11: Example of files created after Shuffle



Name	Date modified	Type	Size
Rd0	6/20/2023 2:37 PM	File	1 KB
Rd1	6/20/2023 2:37 PM	File	1 KB
Rd2			
Rd3			
Rd4			
Rd5			

```
beer 478
```

Fig. 12: Example of files created after running Reduce



```
deer 376
deer 10
beer 478
river 376
homme 14
machine 14
```

Fig. 13: Example of a file containing the calculation result

X. RESULTS

In a test context, we evaluated the latency of MapReduce requests. We chose a file with a fixed size of 120 MB. This file contains a set of words. The Map and Reduce tasks consist in counting certain keywords that we specified as arguments at the start of the program launch. First, we carried out an initial test to ensure that all the steps would run successfully. To do this, we configured the RGB architecture and the nodes on a single machine with 32 GB ram capacity and an SSD disk. The addresses of the computing peers and secondary nodes are managed using text files. We ran the same Job several times, varying the number of nodes from 1 to 10.

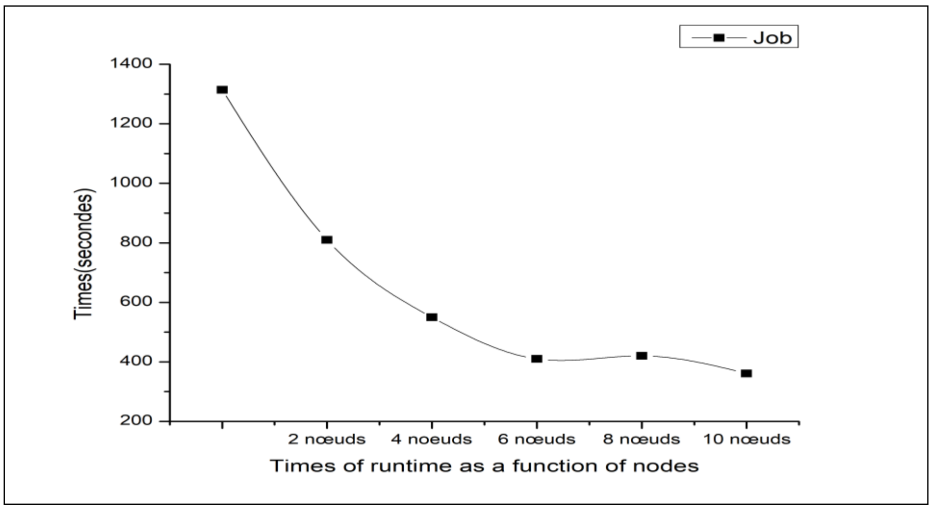


Fig. 14: Job execution time as a function of nodes

Fig.15 shows the evolution of calculation times for the same. Job. We obtained an average value of 1214 milliseconds, approximately (1.3 seconds) for one node, and an average value of 361 milliseconds for 10 nodes. Thegreater the number of nodes, the longer the execution time. This implies that the processes of dividing files into blocks, distributing these blocks, counting and sorting are successfully completed.

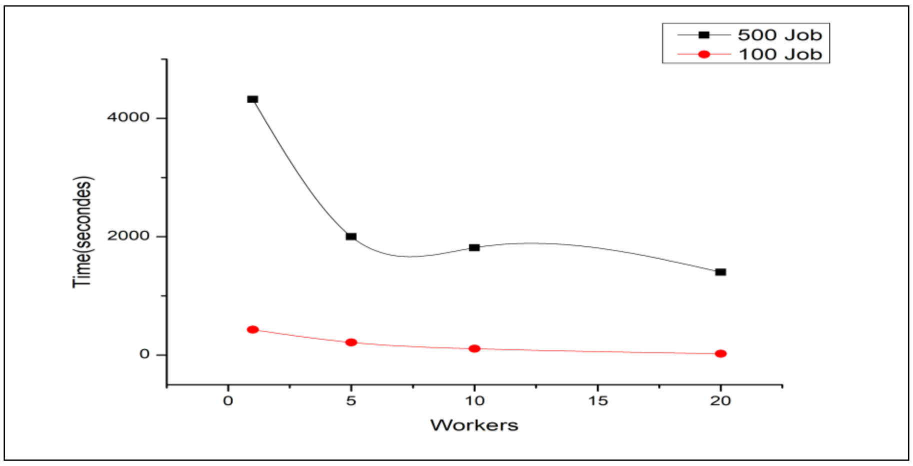


Fig. 15: Processing time as a function of compute nodes

Fig.16 shows the result of experimenting with MapReduce on the RGB architecture in the deployment environment described above. We decided to run 100 jobs and then 500 jobs simultaneously, varying the number of nodes (Workers) from 1 node to 20 nodes. For the 100 jobs, we have 429.4 seconds for 1 node versus 22.10 seconds for 20 nodes. For 500 jobs, we have 4322 seconds for 1 node versus 1400 seconds for 20 nodes. For this experiment, we observe a progressive decrease in processing time, as shown in Fig.16. We can therefore deduce an acceleration factor by calculating *(T_1/T_n)*. This gives 19.41 for 100 jobs and 3.08 for 500 jobs respectively. Note that the higher the number of jobs, the longer the computation time, but the shorter it is if several nodes are assigned to the jobs.

The graphs in Fig.17 show the evolution of computation times for jobs between 1 and 20 compute nodes. This estimate is based on a proportional calculation and the data collected in the previous analyses. The values (in seconds) given in the table indicate calculation times.

Table 1: Estimated time spent depending on nodes

Workers	**100 Jobs**	**200 Jobs**	**300 Jobs**	**400 jobs**	**500 jobs**
	Time(s)				
1	429.4	858	1287	1716	4321.9
5	214.1	729.8	1158.8	1587.8	2002.7
10	107.5	601.5	1030.5	1459.5	1815.0
20	22.1	472.3	901.73	1330.3	1399.3

Table 2 : Estimated time spent depending on nodes

Workers	**600 jobs**	**700 jobs**	**800 jobs**	**900 jobs**	**1000 jobs**
	Time(s)				
1	518.2	6049.17	6913.14	7777.13	8641.11
5	4217.2	5081.17	5945.14	6810.13	7971.11
10	3249.2	4113.17	4977.14	5843.13	7301.11
20	2281.2	3145.17	4009.14	4876.13	6631.11

Graphs on Fig.17 have the same shape, showing an improvement in calculation time despite the large numberof jobs submitted.

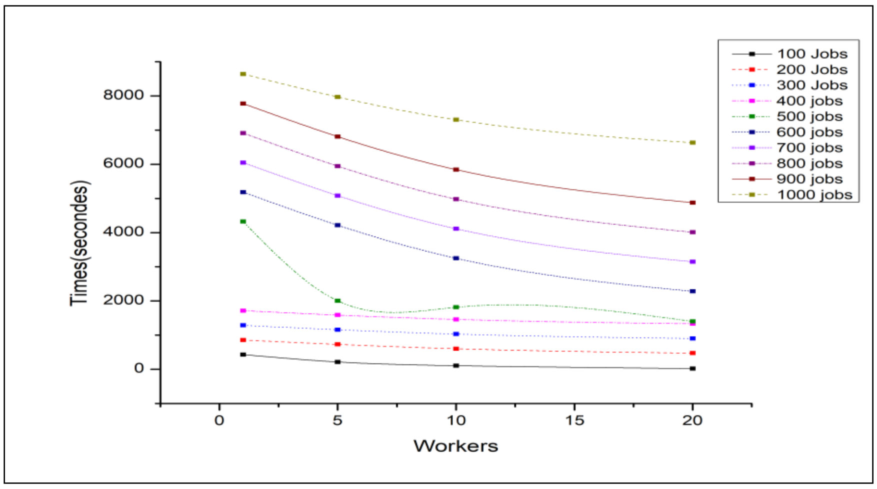


Fig. 16: Estimating execution time as a function of nodes

Furthermore, the graphs in Fig.16 and Fig.15follow a logarithmic function. Taking into account the speed of execution, we make a projection based on the equation 3. This produces the graphs shown in Fig.17.

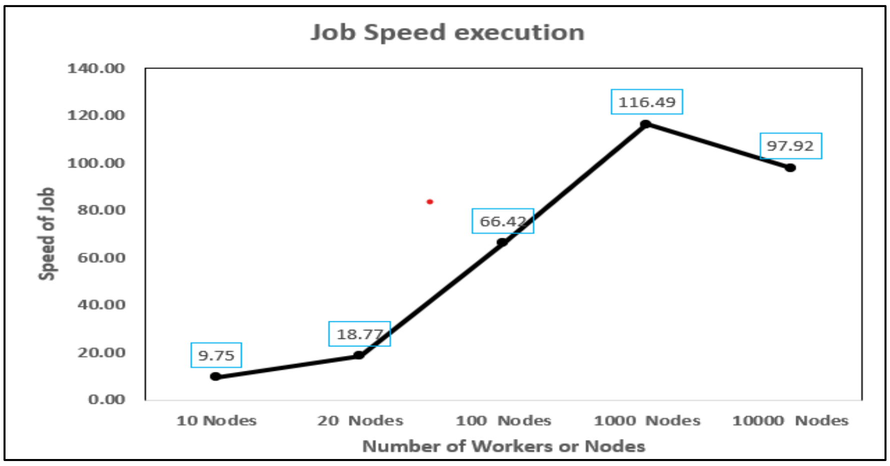


Fig. 17: Job execution speed by node

Fig.18 shows a theoretical estimate of execution speed as a function of the number of nodes. Based on 500 jobs submitted, for 100 nodes we have an execution speed 66.42 times the execution speed of a node loaded with the same number of jobs. At 10,000 nodes, the speed can reach 97.92 times the speed of a loaded node.

XI. CONCLUSION

In this article, we present MapReduce on the RGB architecture, a massive data processing architecture based on the peer-to-peer networking protocol. This architecture has self-managed, dynamic compute nodes thanks to the distributed hash table property used in the CAN protocol. We therefore experimented with MapReduce operation in a decentralized environment and showed that MapReduce is scalable, load-balanced and fault-tolerant thanks to the dynamism of the nodes in the RGB architecture from a network point of view.We implemented a fully functional version of MapReduce on the RGB architecture and carried out detailed experiments to test its performance. These experiments confirmed that the architecture is robust and efficient. P2P network protocols are traditionally known for file sharing. We have demonstrated that it can also be used to build a data pipeline and perform distributed computations on large volumes of data.

In the near future, we intend to further optimize the performance of MapReduce and the RGB architecture by studying an efficient load-balancing system.

REFERENCES

[1]. Li F, Ooi BC, Özsu MT, Wu S. Distributed data management using MapReduce. ACM Comput Surv. 2014 Jan 1;46(3):31:1-31:42.

[2]. Chaudhuri S, Weikum G. Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System. Proceedings of the 26th International Conference on Very Large Data Bases, VLDB'00. 2000 Dec 20;

[3]. Özsu MT, Valduriez P. Principles of Distributed Database Systems, Third Edition [Internet]. New York, NY: Springer New York; 2011 [cited 2023 Jun 6]. Available from: https://link.springer.com/10.1007/978-1-4419-8834-8

[4]. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM. 2008 Jan;51(1):107–13.

[5]. Apache Hadoop [Internet]. [cited 2023 Jun 7]. Available from: https://hadoop.apache.org/

[6]. PoweredBy - HADOOP2 - Apache Software Foundation [Internet]. [cited 2023 Jun 7]. Available from: https://cwiki.apache.org/confluence/display/HADOOP2/PoweredBy

[7]. Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop Distributed File System. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). 2010. p. 1–10.

[8]. Big Data Storage based on CAN Protocol [Internet]. 2023 [cited 2023 Jun 8]. Available from: https://www.researchsquare.com

[9]. Lämmel R. Google's MapReduce programming model — Revisited. Science of Computer Programming. 2008 Jan;70(1):1–30.

[10]. Dromard J. Etat de l&#39;art: Réseaux pair à pair, supervision, sécurité et approches collaboratives. 2010 Jan 1 [cited 2023 Jun16];Available from: https://www.academia.edu/55363840/Etat_de_lart_R%C3%A9seaux_pair_%C3%A0_pair_supervision_s%C3%A9curit%C3%A9_et_approches_collaboratives

[11]. Kabre L. Comparative Study of can, Pastry, Kademlia and Chord DHTS. 2021 Jan 1 [cited 2023 Jun 16]; Available from: https://www.academia.edu/51132180/COMPARATIVE_STUDY_OF_CAN_PASTRY_KADEMLIA_AND_CHORD_DHTS

[12]. Ratnasamy S, Francis P, Handley M, Shenker S, Karp R. A Scalable Content-Addressable Network.

[13]. Ferrah I. Un nouveau classificateur de codage RVBRNA et analyse fractale pour l'étude et le diagnostic d'un isolateur pollué sous tension alternative 50 Hz [Internet] [Thesis]. 2021 [cited 2023 Jun 16]. Available from: http://repository.enp.edu.dz/jspui/handle/123456789/9817

[14]. Kofi D, Mouaddib EM, Salvi J. Décodage d'un motif structurant codé par la couleur.

[15]. Rosen A. Towards a Framework for DHT Distributed Computing [Internet]. Georgia State University; [cited 2023 Jun 29]. Available from: https://scholarworks.gsu.edu/cs_diss/107

[16]. MapReduce Word Count | Guide to MapReduce Word Count | Examples [Internet]. EDUCBA. 2020 [cited 2023 Jun 16]. Available from: https://www.educba.com/mapreduce-word-count/

[17]. Conference: Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers, DOI:10.1007/978-3-642-05445-7_19.