Society for Electronics, Telecommunications, Computers,
Automatic Control and Nuclear Engineering
11000 Belgrade * Kneza Miloša 9/IV * Phone 011 32 33 957 * E-mail: etran@eunet.rs * http://etran.etf.rs

IcETRAN 2015

Proceedings of papers

International Conference on Electrical, Electronic and Computing Engineering

Silver Lake (Srebrno jezero), Serbia, June 8 – 11, 2015

ISBN 978-86-80509-71-6

ISBN 978-86-80509-71-6

Zbornik radova - 59. Konferencija za elektroniku, telekomunikacije, računarstvo, automatiku i nuklearnu tehniku, Srebrno jezero, 8. – 11. juna, 2015. godine
Proceedings of Papers - 2nd International Conference on Electrical, Electronic and Computing Engineering IcETRAN 2015, Silver Lake (Srebrno jezero), Serbia, June 8 – 11, 2015

Glavni urednik/Main Editor: Zoran Jakšić

Urednici / Editors: Zorica Nikolić, Veljko Potkonjak

Izdavač / Published by: **Društvo za ETRAN, Beograd/ETRAN Society, Belgrade**

Izrada / Production: Akademska misao, Beograd / Academic Mind, Belgrade

300 primeraka / 300 copies

ISBN 978-86-80509-71-6

CIP - Каталогизација у публикацији, Народна библиотека Србије, Београд

621.37/.38(082)(0.034.2)
534(082)
621.39(082)(0.034.2)
004(082)(0.034.2)
681.5(082)(0.034.2)
621.039(082)(0.034.2)
66.017(082)

ДРУШТВО за електронику, телекомуникације, рачунарство, аутоматику и нуклеарну технику. Конференција (59 ; 2015 ; Сребрно језеро)
ETRAN [i] IcETRAN [Elektronski izvor] : [zbornik radova : proceedings of papers / 59. Konferencija za elektroniku, telekomunikacije, računarstvo, automatiku i nuklearnu tehniku, Srebrno jezero, 8. - 11. juna, 2015. godine [i] 2nd International Conference on Electrical, Electronic and Computing Engineering IcETRAN 2015 Silver Lake (Srebrno jezero), Serbia, June 8 - 11, 2015 ; urednici, editors Zorica Nikolić, Veljko Potkonjak]. - Beograd : Društvo za ETRAN = Belgrade : ETRAN Society, 2015 (Beograd : Akademska misao). - 1 elektronski optički disk (CD-ROM) ; 12 cm
Sistemski zahtevi: Nisu navedeni. - Nasl. sa nasl. ekrana. - Tekst ćir. i lat. - Radovi na srp. i engl. jeziku. - Tiraž 300. - Bibliografija uz svaki rad. - Abstracts.

ISBN 978-86-80509-71-6

1. International Conference on Electrical, Electronic and Computing Engineering (2 ; 2015 ; Srebrno jezero)

a) Електроника - Зборници b) Акустика - Зборници c) Телекомуникације - Зборници d) Рачунарска технологија - Зборници e) Системи аутоматског управљања - Зборници f) Нуклеарна техника - Зборници g) Технички материјали - Зборници

COBISS.SR-ID 217144076

Filip Zivanovic, Vladimir Ciric, Natalija Stojanovic, Ivan Milentijevic, "Optimized One Iteration MapReduce Algorithm for Matrix Multiplication", Proceedings of 2nd International Conference on Electrical, Electronic and Computing Engineering (IcERTAN 2015), Srebrno jezero, Serbia, June 2015, RTI2.5. ISBN: 978-86-80509-71-6

Optimized One Iteration MapReduce Algorithm for Matrix Multiplication

Filip S. Živanović, Vladimir M. Ćirić, Natalija M. Stojanović, Ivan Z. Milentijević

***Abstract*—For the last several years Cloud Computing is becoming dominant technology for big data processing. Apache Hadoop, as one of the frameworks for data storage and computing on the cloud, in the form of PaaS using MapReduce paradigm, is widely adopted among both developers and researchers. Taking into the account great importance of matrix multiplication in computer science in general, the goal of this paper is development of new MapReduce algorithm for matrix multiplication, which is able to perform the multiplication in one iteration. The proposed one iteration MapReduce algorithm is optimized by prearranging the input data in such manner that matrix elements can be fetched in sequential order. The proposed algorithm is presented in detail. The implementation results are given. The results obtained on the cluster with 20 nodes are discussed.**

***Index Terms*—Distributed computing, Cloud computing, Hadoop MapReduce, Matrix multiplication.**

I. Introduction

The challenge that big companies are facing nowadays is big data storage and processing. Google was the first that designed a system for processing such data, which allows overcoming the problems that appears with big amount of data, while utilizing a fast local area network for data distribution [1]. The crucial novelty was new programming model in the form of MapReduce paradigm. MapReduce provides a simple heterogenous model for storing and analyzing data in heterogenous systems that can contain many nodes [1].

The MapReduce is a programming model for processing large data sets. Application developers specify a Map function that processes a $(key, value)$ pair to generate a set of intermediate $(key, value)$ pairs and a Reduce function that merges all intermediate values associated with the same intermediate key [1].

Doug Cutting led the development of an open source version of this MapReduce system called Hadoop, which in 2008 became independent Apache project. Soon after, Yahoo and others showed a big interest in this solution. Today, Hadoop is a core part of the computing infrastructure for a lot of big companies, such as Yahoo, Facebook, LinkedIn, Twitter, etc [2].

Filip S. Živanović is with the Faculty of Electronic Engineering, University of Niš, Aleksandra Medvedeva 14, P.O.Box 73, 18000 Niš, Serbia (e-mail: filip.zivanovic@elfak.ni.ac.rs).

Vladimir M. Ćirić is with the Faculty of Electronic Engineering, University of Niš, Aleksandra Medvedeva 14, P.O.Box 73, 18000 Niš, Serbia (e-mail: vladimir.ciric@elfak.ni.ac.rs).

Natalija M. Stojanović is with the Faculty of Electronic Engineering, University of Niš, Aleksandra Medvedeva 14, P.O.Box 73, 18000 Niš, Serbia (e-mail: natalija.stojanovic@elfak.ni.ac.rs).

Ivan Z. Milentijević is with the Faculty of Electronic Engineering, University of Niš, Aleksandra Medvedeva 14, P.O.Box 73, 18000 Niš, Serbia (e-mail: ivan.milentijevic@elfak.ni.ac.rs).

When working with MapReduce on Hadoop cluster, parallelism, fault tolerance, data distribution and load balancing are inherited from Hadoop system itself. It is also optimized to handle big amounts of structured and unstructured data, by using widely accessible and relatively chip computers for performance gain.

Matrices are widespread and applicable in variety of calculations. Large matrices can contain a big amount of data (1.000.000 elements, and more), which should be stored and processed. The amount of data within the matrix can be large, making the matrix operations data intensive. In some cases the most of the elements are equal to zero, so this feature can be further exploited. If maximum of 5-10% of matrix is filled up with non-zero elements, data can be assumed as a sparse matrix. Since zero elements don't affect multiplication, this could be taken into account during algorithm design.

There are plenty of different ways for acceleration of matrix multiplication. Some of them include parallel processing of data, such as multiplication on systolic arrays [3] and many-core architectures [4], and some include multiplication in distributed environments like Hadoop [5], [6]. Two major challenges that occur in all of them are: optimal matrix storage, and efficient matrix multiplication. Regarding the Hadoop storage, two main problems are data redundancy and inefficient sequential input of matrix data. Current MapReduce implementations usually have more than one MapReduce iteration [5].

The goal of this paper is the development of new MapReduce algorithm for matrix multiplication, which is able to perform the multiplication in one iteration. In order to achieve the goal of multiplication in one iteration, the data will be prearranged in manner that matrix elements can be fetched in sequential order. The proposed algorithm will be presented in detail. The implementation results will be given. The implementation results will be given. The results obtained on the cluster with 20 nodes will be discussed.

The paper is organized as follows: Section 2 gives a brief overview of MapReduce paradigm on Hadoop cluster. In Section 3 existing solutions for matrix multiplication on Hadoop are presented, while in Section 4 we propose a new MapReduce algorithm for matrix multiplication optimized for one iteration by data prearrangement. Section 5 is devoted to the implementation results, while in Section 6 the concluding remarks are given.

II. MapReduce paradigm on Hadoop cluster

The central part of the Hadoop architecture is a cluster. It represents a large collection of networked computers (hosts

- nodes), containing data for processing. Collection of 20-30 nodes, which are physically close and connected to one switch, is known as rack. Therefore, Hadoop cluster consists of collection of racks [7].

There are three types of nodes depending on their roles in cluster:

1) Client host - loads data into the cluster, forwards MapReduce job that describes the way of processing data, and collects the results of performed job at the end.
2) Master node - monitors two key components of Hadoop: storage of big data, and parallel executions of computations.
3) Slave node - performs actual data storage and computing.

As it is mentioned above, there are two main components of Hadoop system:

- Distributed File System - Hadoop DFS (HDFS), used for big data storage in cluster,
- MapReduce - framework of computing big data stored in HDFS.

The HDFS is a layer above existing file system of every node in cluster, and Hadoop uses its blocks to store input files or parts of them. Large files are split into a group of smaller parts called blocks (default block size is 64MB). Size of these blocks is fixed, so it is easy to index any block within the file, and because of that it is straightforward to use a file within HDFS that can be bigger than any individual disk in cluster [7].

Particular blocks of a file on HDFS can be replicated in order to achieve some level of data redundancy. Blocks replication on multiple nodes allow HDFS to be fault tolerant. Specifically, if for example node N1 stops working, there are more blocks that contain the same data as N1, and it allows them to continue the computations. Attribute of Hadoop that affects intensity of block replication is a replication factor, that can be set in configuration files. The example HDFS with replication factor 3 on 5 slave nodes is shown in Fig. 1.

Typical HDFS workflow has 4 parts: transferring input data from Client host to HDFS, processing data using MapReduce framework on the slave nodes, storing results by Master node on HDFS, and reading data by Client host from HDFS.

In essence, MapReduce technique consists of two transformations that can be applied many times on input files [7]: Map transformation, and Reduce transformation.

The Map transformation consists of M_t Mappers, or Map tasks, and the Reduce transformation consists of R_t Reducers, or Reduce tasks, where M_t and R_t are specified in system configuration of Hadoop . During the Map transformation, every Map task processes a small part of the input file and passes the results to the Reduce tasks. After that, during the Reduce transformation, Reduce tasks collect the intermediate results of Map tasks and combine them in order to get the output, i.e. the final result, as shown in Fig. 2.

During the execution of the Mapper, the Mapper calls a Map function, which performs required computations. Precisely, Map function transforms input dataset into the set of output values $(key, value)$. After that, intermediate data with the same key are grouped and passed to the same Reduce function. At the end, Reduce function summarizes all data with the same key in order to get the final result. Every output file is intended for specific Reduce task. (Fig. 2).

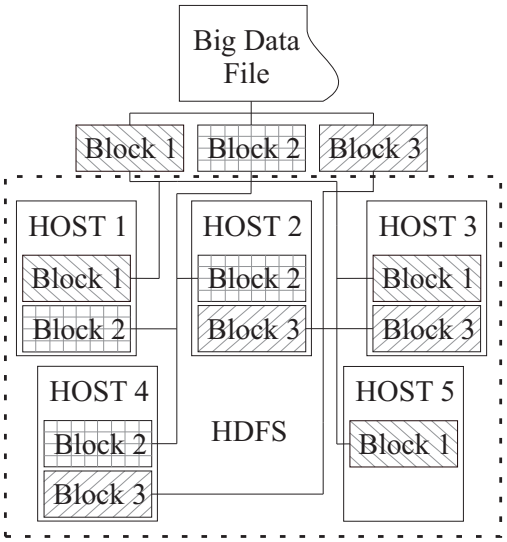

Fig. 1. HDFS

Transformations of Map and Reduce functions look as follows:

- Map : $(k_{in}, v_{in}) \rightarrow \text{list}(k_{intermediate}, v_{intermediate})$,
- Reduce : $(k_{intermediate}, \text{list}(v_{intermediate})) \rightarrow \text{list}(k_{out}, v_{out})$,

where k_{in}, $k_{intermediate}$ and k_{out} are keys, while v_{in}, $v_{intermediate}$ and v_{out} are values.

It should be mentioned that additional Shuffle and Sort functions can exist between Map and Reduce functions, that makes the job of the Reducer easier, by collecting all elements with the same key [7].

Map task, or Mapper, begins when Client host sends parts of the file that consist of data for processing to Mapper, e.g. input splits. The size of the input split is fixed, which means that the Client host doesn't know an internal logical structure of the input file. It means that the Client host splits input file into byte sequences.

A single Mapper starts by reading an input split which had been sent to it. Input splits are denoted as Input Split 1 to Input Split 6 in Fig. 2. In order to generate $(key, value)$ pairs, every input split is parsed by one Map function. The key argument of the Map function input is usually the offset of the line in the input file, while the value argument is one line of input split. After that, specific Mapper performs certain operations on the data and generates output $(key, value)$ pairs (Fig. 2). The Mapper should generate $(key, value)$ pairs in such manner that several pairs with the same key can be grouped on the same Reducer.

As an addition to the Mapper, there is a Combine function (Combiner), which can be used to offload a part of the Reducer's job to the Mapper. In fact, if MapReduce uses Combiner, output pairs of Map function are not written on the

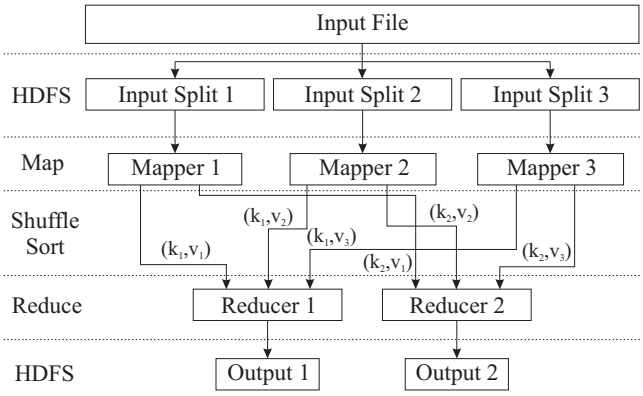

Fig. 2. MapReduce data and process flow

output immediately, but rather collected into the lists within the memory - one list for every key value, and further processed by Combiner [7].

There is a hidden phase between Mapper and Reducer, which contains Shuffle and Sort functions. It represents synchronization barrier, because it groups all data generated from Map function by its key values, sorts them by certain criteria, and passes them to appropriate Reducer in the form of $(key, list\ of\ values)$ pairs (Fig. 2) [7].

The Reduce task, or the Reducer, uses $(key, list\ of\ values)$ pairs as input arguments. Before Reduce function is started, its input files are spilled all over the cluster. As soon as all Map functions generate their outputs, they merge into one file per Reducer by Shuffle and Sort functions. All the data with the same key values are processed with Reduce functions in the same manner, and a single output per Reducer is formed from all its $(key, value)$ pairs. At the end, $(key, value)$ pairs that represent output data of MapReduce process are formed - one pair per Reduce function (Fig. 2) [7].

III. Matrix multiplication on Hadoop cluster

In order to develop an efficient algorithm for matrix multiplication on Hadoop cluster, we will briefly introduce and analyze the latest research in this area. In this section two existing algorithms for matrix multiplication on Hadoop cluster are presented.

Authors in [5], performed an extensive analysis using private cloud infrastructure in order to understand the performance implications of virtualized resources on distributed applications. For this analysis, several applications with different communication/computation characteristics were used, namely Matrix Multiplication, Kmeans Clustering, and Concurrent Wave Equation Solver. Instead of measuring individual characteristics such as bandwidth and latency using micro benchmarks, real applications are used to clarify the effect of virtualized resources.

The row/column decomposition used for matrix multiplication on Hadoop cluster that authors in [5] proposed is shown in Fig. 3. If result matrix C is obtained by multiplying matrices A and B, matrix B is split into a set of column blocks and matrix A is split to a set of row blocks (Fig. 3). The number of MapReduce iterations of proposed algorithm is equal to the number of rows in matrix A. The number of Map functions per iteration is equal to the number of columns of matrix B (Fig. 3). The number of Reducers per iteration is one. In each iteration, each Map function consumes two inputs: one column of matrix B, and one row of matrix A. If decomposition of matrices is considered for one iteration, then all Mappers perform computation with the same row of matrix A and with different column of matrix B. Considering different iterations through time, each Mapper performs a computation with different row of matrix A, and the same column of matrix B (Fig. 3) [5].

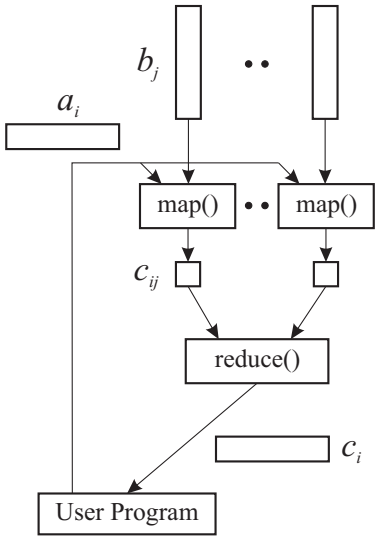

Fig. 3. Row/Column decomposition

It can be concluded that algorithm for matrix multiplication proposed in [5] can be improved and optimized, using the fact that the Map and Reduce functions are called many times during the computation, which can increase network load.

In [6], three different approaches in mapping matrix elements to Hadoop processes are considered. Only one of them shows an improvement compared to algorithm for matrix multiplication proposed in [5]. In that approach, complete process of matrix multiplication is done in one MapReduce iteration. Each Map function performs computation with different columns of matrix A, i.e. in general, with different vertical blocks of matrix A, and with different horizontal blocks of matrix B. However, reading of vertical matrix blocks is not efficient, due to the fact that reading of one vertical block requires to read many lines from the input file [6]. The matrix decomposition into blocks can be further discussed in order to reduce cache misses, and additionally improve the performances [8].

In this paper we propose acceleration of Map function by prearrangement of the input data which enables a sequential fetch of matrix elements without a need for reading all lines from the input file that consist a required block.

IV. Design of MapReduce Algorithm for Matrix Multiplication with Data Prearrangement

As shown in the previous section, performances of matrix multiplication using MapReduce can be improved by execut-

ing the complete MapReduce job during one iteration, while the elements of the matrices are fetched sequentially.

In order to enable a MapReduce matrix multiplication job to be performed in one iteration, all result elements should be generated during the first Reducer call. Therefore, every Map function needs to generate one part of result matrix, which can be either one row or one column. In both cases, every Map function should receive one row or column from the first matrix, and the whole second matrix. It would be a big load for the network and it would cause a big data redundancy. The lowest redundancy is for the case where every node receives different data, but in such manner that each element exists only on one node, not taking into the account an automatic replication performed by Hadoop. This could be done using the feature of matrix multiplication as shown in Fig. 4: only the elements from the first matrix with the column index associated to the row index of the elements from the second matrix will be multiplied. According to this, all elements of the first column from the first matrix will only multiply the elements of the first row from the second matrix, and vice versa.

$$
\begin{aligned}
c_{00} &= a_{00} * b_{00} + a_{01} * b_{10} + a_{02} * b_{20} \\
c_{01} &= a_{00} * b_{01} + a_{01} * b_{11} + a_{02} * b_{21} \\
c_{02} &= a_{00} * b_{02} + a_{01} * b_{12} + a_{02} * b_{22} \\
c_{10} &= a_{10} * b_{00} + a_{11} * b_{10} + a_{12} * b_{20} \\
c_{11} &= a_{10} * b_{01} + a_{11} * b_{11} + a_{12} * b_{21} \\
c_{12} &= a_{10} * b_{02} + a_{11} * b_{12} + a_{12} * b_{22} \\
c_{20} &= a_{20} * b_{00} + a_{21} * b_{10} + a_{22} * b_{20} \\
c_{21} &= a_{20} * b_{01} + a_{21} * b_{11} + a_{22} * b_{21} \\
c_{22} &= a_{20} * b_{02} + a_{21} * b_{12} + a_{22} * b_{22}
\end{aligned}
$$

$$
\begin{aligned}
(a_{00}, a_{10}, a_{20}) * (b_{00}, b_{01}, b_{02}) &= (c^0_{00}, c^0_{01}, c^0_{02}, c^0_{10}, c^0_{11}, c^0_{12}, c^0_{20}, c^0_{21}, c^0_{22}) \\
(a_{01}, a_{11}, a_{21}) * (b_{10}, b_{11}, b_{12}) &= (c^1_{00}, c^1_{01}, c^1_{02}, c^1_{10}, c^1_{11}, c^1_{12}, c^1_{20}, c^1_{21}, c^1_{22}) \\
(a_{02}, a_{12}, a_{22}) * (b_{20}, b_{21}, b_{22}) &= (c^2_{00}, c^2_{01}, c^2_{02}, c^2_{10}, c^2_{11}, c^2_{12}, c^2_{20}, c^2_{21}, c^2_{22})
\end{aligned}
$$

Fig. 4. Matrix multiplication feature

As shown in the Fig. 4, element $a_{0,0}$, is needed for multiplication of elements only from the first row of matrix B. The same case is for the elements $a_{1,0}$ and $a_{2,0}$. The multiplication of the element $a_{i,0}$ and $b_{0,j}$ will produce an intermediate result $c^0_{i,j}$ (Fig. 4), which is the first intermediate result of the resulting element $c_{i,j}$ ($c_{i,j} = c^0_{i,j} + c^1_{i,j} + c^2_{i,j}$). Elements $a_{0,1}$, $a_{1,1}$, and $a_{2,1}$ multiply elements only from the second row of matrix B, and so on.

MapReduce reads one column from the first, and one row from the second matrix, and sends them consecutively to the nodes. In particular, those are elements from the first column of matrix A denoted as $(a_{i,0}), i = 0, 1, 2$ in Fig. 4, and from the first row of matrix B denoted as $(b_{0,i}), i = 0, 1, 2$. However, because of the MapReduce feature to read lines of the input file sequentially, we propose transposition of the first matrix so that appropriate columns can be sequentially fetched. After that, in order to additionally optimize reading of the input matrices, only one input file can be created for both matrices, where each line consists of one column of the matrix A, i.e. one row of transposed matrix A^T, and corresponding row of matrix B.

The further optimization can be done in the case of sparse matrices, which by definition have no more than 5-10% of non-zero elements. In that case the elements with zero value will not be included in the input file, and processed by MapReduce Fig. 5.

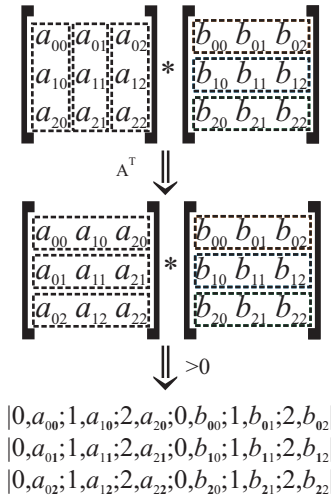

Fig. 5. Input file format

The input file is created as it is shown in Fig. 5. It enables MapReduce function to send line by line of the input file to the cluster nodes, without redundant data. The index that stands by the value of each element from the first matrix in the lower part of Fig. 5 indicates row index of the intermediate data, while the index that stands by the value of each element from the second matrix indicates a column index of the intermediate data.

After the preparation of the file in the format shown in Fig. 5, Mapper performs the multiplication as it is shown in the Fig. 6. Each Map function multiplies every single element from the left half of the received line (elements of the appropriate column from the first matrix), by every single element from the right half of the received line (elements of the appropriate row from the second matrix). In the example shown in Fig. 6 nine intermediate results of the resulting matrix are produced: $c^h_{0,0}, c^h_{0,1}, \ldots, c^h_{2,2}$, where h is the index of the function. The index i of the row and the index j of the column of the intermediate result $c^h_{i,j}$ together will represent the key for the reduce function, while the value will be the intermediate result $c^h_{i,j}$. The Reducers will perform final summarization of intermediate results, one Reducer per resulting element $c_{i,j}$. The pseudo-code of the Map function is given as follows:

```
Map (key, value)
  line = value.split(;)
  for ( i = 0 .. number_of_matrix_A_elements −1)
    elementA = line[i].split(,)
    for ( j = number_of_matrix_A_elements ..
              .. line.length −1)
      elementB = line[j].split(,)
```

i = $elementA[0]$
j = $elementB[0]$
$k = i\ ,\ j$
$v = elementA[1]\ *\ elementB[1]$
$emit\ (k, v)$.

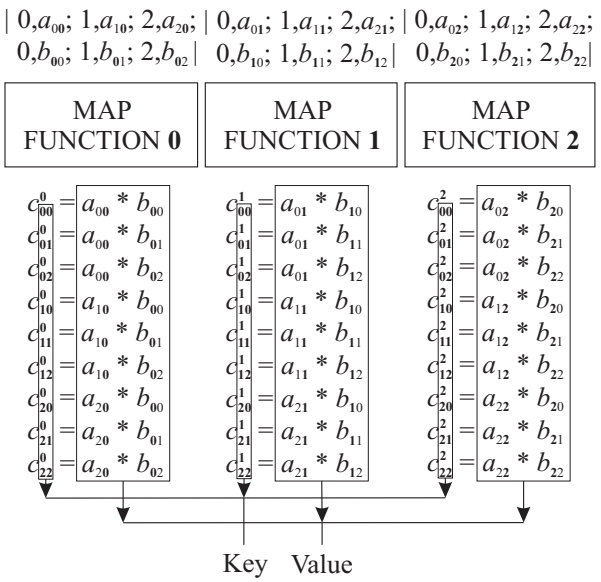

Fig. 6. Map function

After the Map function, as it is shown in Fig. 7, every Reduce function needs only to summarize all intermediate data, e.g. to summarize all intermediate values associated with the same key. Reduce functions are denoted as R1 to R9 in Fig. 7. The pseudo-code of the Reduce function is given below:

$Reduce\ (key, valueList)$
$sum = 0$
$for\ each\ value\ in\ valueList$
$sum\ +=\ value$
$value = sum$
$emit\ (key, value)$.

V. Implementation results

In order to illustrate the performances of the proposed MapReduce algorithm for matrix multiplication with data prearrangement, the algorithm is implemented and executed on the Hadoop cluster that consists of 21 nodes. The characteristics of the nodes are the following: 3 out of 21 nodes are "IBM" (Intel(R) Pentium(R) 4, CPU@3.00GHz, RAM: 2.4GB), while the other 18 are "Lenovo" (Intel(R)Core(TM)2 Duo, CPU E4600@2.40GHz, RAM: 1GB). It should be mentioned that one of the "IBM" machines is the master node, and the remaining 20 are slaves.

Measurements were performed with the following parameters:(1) both matrices are sparse, with fixed density of 10%; (2) the dimensions of the matrices vary within the range 500 to 5000, with the step of 500; (3) number of slave nodes is 20; (4) number of mappers and reducers is 20.

If the block size remains at its default value of 64MB, it is likely, due to the small size of matrices which are tested, that

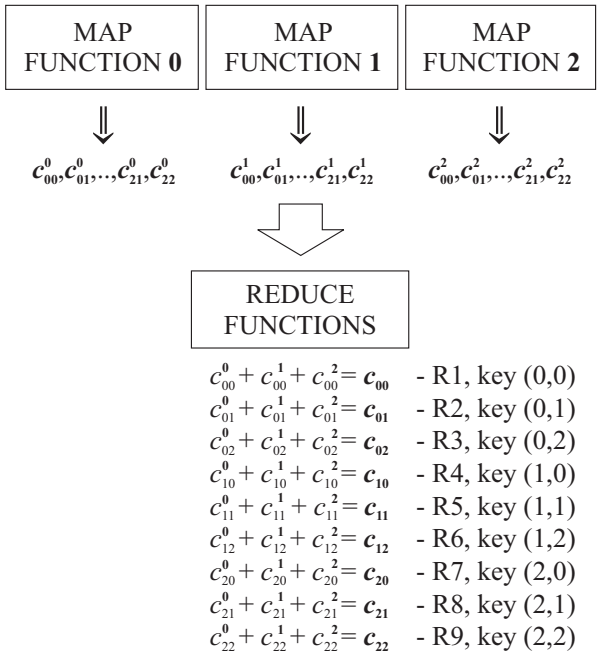

Fig. 7. Reduce function

prepared input file will be stored in one block only. In that case only one mapper will be utilized, leaving the remaining 19 configured mappers unused. Thus, block size is configured in such manner that every mapper gets one block ($block\_\ size\ =\ input\_\ file\_\ size\ /\ number\_\ of\_\ mappers$).

Both files that represent the first and the second matrix are implemented as textual files, which gives the MapReduce textual input.

The obtained results for the proposed MapReduce algorithm for matrix multiplication are given in the Fig. I. The first column stands for the matrix dimension, while in the second column the computation time is given. The third column stands for direct application of conventional 3-loop matrix multiplication on single host.

matrix dimension	Hadoop time (min)	CPU time (min)
500	0,3	0,47
750	0,35	1,03
1000	0,4	2,72
1250	0,57	5,37
1500	0,77	14,85
1750	1,11	41,07
2000	1,55	-
2500	2,4	-
3000	3,82	-
3500	5,12	-
4000	8,12	-
4500	11,38	-
5000	15,18	-

TABLE I
OBTAINED RESULTS FOR MAPREDUCE ALGORITHM FOR MATRIX MULTIPLICATION

The results from Fig. I are graphically represented in Fig. 8.

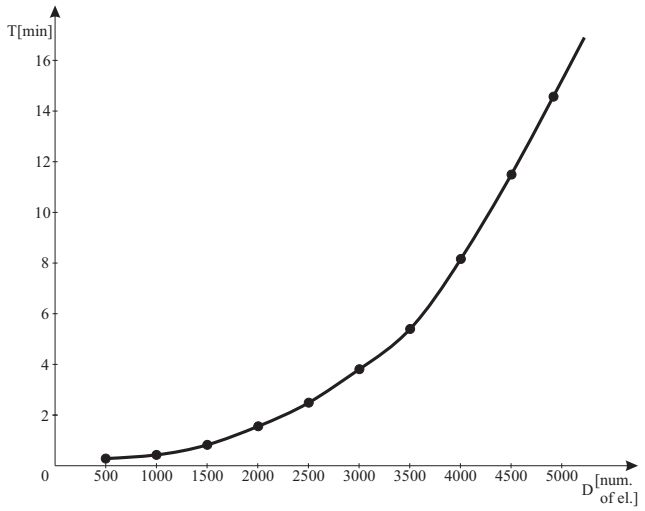

Fig. 8. Executions time dependency of matrix dimension for MapReduce algorithm for matrix multiplication

It should be mentioned that the proposed algorithm can be additionally accelerated by using binary files instead of textual files as an input. It means that the Map function will avoid the text parsing overhead by reading a binary file instead of a textual file, mostly by eliminating time for string splitting, which can influence the obtained results.

VI. Conclusion

In this paper the development of new MapReduce algorithm for matrix multiplication, which is able to perform the multiplication in one iteration, is presented. In order to achieve the goal of multiplication in one iteration, the data was prearranged in such manner that matrix elements could be fetched in sequential order. The proposed algorithm was presented in detail. The data was prearranged in such manner which introduced acceleration of Map function by enabling sequential fetch of matrix elements without a need for reading all lines from the input file that consist a required block. The implementation results were given. The results were obtained on the cluster with 20 nodes.

Acknowledgment

The research was supported in part by the Serbian Ministry of Education, Science and Technological Development (Project TR32012).

References

[1] J. Dean, *"MapReduce: a flexible data processing tool"*, Communications of the ACM, Vol. 53 Issue 1, January 2010, pp. 72-77.

[2] C. Lam, *"Hadoop in Action"*, Stamford, Connecticut, USA : Manning Publications, 2010

[3] I. Ž. Milovanovic, M. P. Bekakosb, I. N. Tselepis, E. I. Milovanovic, *"Forty-three ways of systolic matrix multiplication"*, International Journal of Computer Mathematics, Vol. 87, Issue 6, 2010

[4] N. Bell, M. Garland, *"Efficient Sparse Matrix-Vector Multiplication on CUDA"*, NVIDIA Technical Report NVR-2008-004, 2008

[5] J. Ekanayake, G. Fox, *"High Performance Parallel Computing with Clouds and Cloud Technologies"*, Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, Vol. 34, 2010, pp 20-38

[6] Y. Shen, L. Li, *"Matrix Layout Problem in MapReduce"*, Electrical and Computer Engineering, Duke University, 2012

[7] T. White *"Hadoop: The Definitive Guide"*, O'Relly Media, 2012

[8] M. Gusev, S. Ristov, *"A Superlinear Speedup Region for Matrix Multiplication"*, Journal Concurrency and Computation, Practice and Experience, Willey InterScience Publisher, Vol. 26, Issue 11, 10 August 2014, pp. 1847-1868.