PROCEEDINGS

ICIST 2016

6TH INTERNATIONAL CONFERENCE ON
INFORMATION SOCIETY AND TECHNOLOGY

6th International Conference on Information Society and Technology

ICIST 2016

Proceedings

Publisher

Society for Information Systems and Computer Networks

Editors

Zdravković, M., Trajanović, M., Konjović, Z.

ISBN: 978-86-85525-18-6

Issued in Belgrade, Serbia, 2016.

Dataflow of Matrix Multiplication Algorithm through Distributed Hadoop Environment

Vladimir M. Ćirić, Filip S. Živanović, Natalija M. Stojanović, Emina I. Milovanović, Ivan Z. Milentijević
Faculty of Electronic Engineering, University of Niš, Serbia
{vladimir.ciric, filip.zivanovic, natalija.stojanovic, emina.milovanovic, ivan.milentijevic}@elfak.ni.ac.rs

***Abstract* — Increasing of processors' frequencies and computational speed with components scaling is slowly reaching its saturation with current MOSFET technology. From today's perspective, the solution lies either in further scaling in nanotechnology, or in parallel and distributed processing. Parallel and distributed processing have always been used to speedup the execution further than the current technology had been enabling. However, in parallel and distributed processing, dependencies play a crucial role and should be analyzed carefully. The goal of this paper is the analysis of dataflow and parallelization capabilities of Hadoop, as one of the widely used distributed environment nowadays. The analysis is performed on the example of matrix multiplication algorithm. The dataflow is analyzed through evaluation of the execution timeline of Map and Reduce functions, while the parallelization capabilities are considered through the utilization of Hadoop's Map and Reduce tasks. The implementation results on 18-nodes cluster for various parameter sets are given.**

I. INTRODUCTION

The current projections by the International Technology Roadmap for Semiconductors (ITRS) say that the end of the road on MOSFET scaling will arrive sometime around 2018 with a 22nm process. From today's perspective, the solution for further scaling lies in nanotechnology [1]. However, parallel and distributed processing have always pushed the boundaries of computational speed, through history of computing, further than it had been enabled by the current chip fabrication technology. Two promising trends nowadays, which enable applications to deal with increasing computational and data loads, are cloud computing and MapReduce programming model [2].

Cloud computing provides transparent access to the large number of compute, storage and network resources, and provides high level of abstraction for data-intensive computing. There are several forms of cloud computing abstractions, regarding the service that is provided to users, including Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) [3].

MapReduce is currently popular PaaS programming model, which supports parallel computations on large infrastructures. Hadoop is MapReduce implementation, which has attracted a lot of attention from both industry and research. In a Hadoop job, Map and Reduce tasks coordinate to produce a solution to the input problem, exhibiting precedence constraints and synchronization delays that are characteristic of a pipeline communication between Maps (producers) and Reducers (consumers) [4]. In distributed processing in general, as well as in the MapReduce, the crucial problems that lie in front of designers, are data dependency and locality of the data. While data dependency influence is obvious, data locality has indirect influence on execution speed in distributed systems due to the communicational requirements. One of the roles of the Hadoop is to automatically or semi-automatically handle the data locality problem.

There are several models and simulators that can capture properties of MapReduce execution [2], [5]. The challenge to develop such models is that they must capture, with reasonable accuracy, the various sources of delays that a job experiences. In particular, besides the execution time, tasks belonging to a job may experience two types of delays: (1) queuing delays due to the contention at shared resources, and (2) synchronization delays due to the precedence constraints among tasks that cooperate in the same job [4].

The goal of this paper is the analysis of dataflow and parallelization capabilities of Hadoop. The analysis will be illustrated on the example of matrix multiplication algorithm in Hadoop, proposed in [6]. The dataflow will be analyzed through evaluation of the execution timeline of Map and Reduce functions, while the parallelization capabilities will be considered through the utilization of Hadoop's Map and Reduce tasks. The results of the implementation for various parameter sets in distributed Hadoop environment consisting of 18 computational nodes will be given.

The paper is organized as follows: Section 2 gives a brief overview of MapReduce programming model. In Section 3 dataflow of MapReduce phases for matrix multiplication algorithm is presented, and data dependencies are discussed. Section 4 is devoted to the analysis of the parallelization capabilities of the matrix multiplication algorithm, as well as to the implementation results, while in Section 5 the concluding remarks are given.

II. BACKGROUND

The challenge that big companies are facing lately is overcoming the problems that appears with big amount of data. Google was the first that designed a new system for processing such data, in the form of a simple model for storing and analyzing data in heterogeneous systems that can contain many nodes. Open source implementation of

this system, called Hadoop, became an independent Apache project in 2008. Today, Hadoop is a core part of a lot of big companies, such as Yahoo, Facebook, LinkedIn, Twitter, etc [7].

The Hadoop cluster consists of collection of racks, each with 20-30 nodes, which are physically close and connected. The cluster consists of three types of nodes depending on their roles: (1) Client host - responsible for loading data into the cluster, forwarding MapReduce job that describes the way of processing data, and collecting the results of performed job at the end; (2) Master node - in charge of monitoring two key components of Hadoop: storage of big data, and parallel executions of computations; (3) Slave node - used for performing actual data storage and computing.

There are two main components of Hadoop system: (1) Distributed File System - Hadoop DFS (HDFS), used for big data storage in cluster; (2) MapReduce - framework used for computing big data stored in HDFS.

The HDFS lies as a layer above existing file system of every node in the cluster, and its blocks are used for storing input data in the form of the input splits (Figure 1). Large files can be split into a group of small parts called blocks, which have default size of 64MB. The size of these blocks is fixed, due to the simplification of indexing [9]. Usually, HDFS workflow consists of 4 parts: (1) transferring input data from Client host to HDFS, (2) processing data using MapReduce framework on the slave nodes, (3) storing results by Master node on HDFS, and (4) reading data by Client host from HDFS.

There are two transformations in MapReduce technique that can be applied many times on input files: Map transformation, which consists of M_T Mappers or Map tasks, and the Reduce transformation, which consists of R_T Reducers or Reduce tasks. The parameters M_T and R_T are specified in system configuration of Hadoop, R_T explicitly, and M_T implicitly through specification of the blocksize. In the Map transformation, each Map task processes one small part of the input file and forwards the results to the Reduce tasks. After that, in the Reduce transformation, Reduce tasks gather the intermediate results of Map tasks and combine them to get the output, i.e. the final result, as shown in Figure 1.

The Mappers, during theirs execution, executes M_F Map functions to perform required computations. One Map function transforms input data, according to input (key_{in}, $value_{in}$) pairs, into the set of intermediate (key_{im}, $value_{im}$) pairs (Figure 1). Let us note that the number of executed Map functions M_F is equal to the number of different keys key_{in}, and that this number doesn't need to be equal to the configured number of Map tasks M_T.

In the phase between Map and Reduce, called Shuffle and Sort, all intermediate data with the same key key_{im} are grouped and passed to the same Reduce function (Figure 1). The number of executed Reduce functions R_F is equal to the number of different keys key_{im}. It doesn't need to be equal to the configured number of Reduce tasks R_T. In the end, all data from the Reduce tasks are written into separate output.

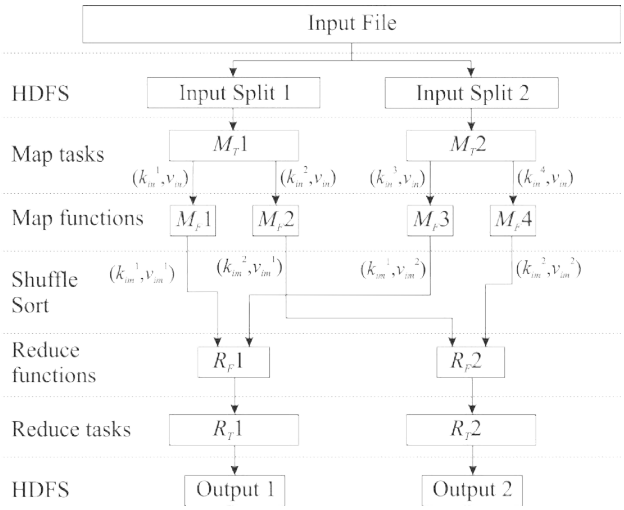

Figure 1. MapReduce data and process flow

MapReduce inherits parallelism, fault tolerance, data distribution and load balancing from Hadoop system itself [8]. As mentioned before, it consists of two main phases, namely, Map and Reduce, each one implemented by multiple tasks (M_T+R_T) running on multiple nodes (N) [4].

Figure 2 shows a simple example of a timeline representing the execution of a Hadoop job composed of M_T=2 Mappers and R_T=1 Reducer, running on N=3 nodes. The number of Map functions in algorithm shown in Figure 2 is M_F=4, and the number of Reduce functions is R_F=4. There is one additional Reducer R_M that collects outputs from all Reduce functions. The notation used for particular Map functions within Map tasks is M_F^i, where i represents the number of the function. The order at which the Reduce functions R_F^i, i=1,2,3,4, begin their execution is defined by the order at which the Map functions M_F^i, i=1,2,3,4, finish theirs. Precisely, Reduce function R_F^i should start as soon as Map function M_F^i finishes and the node that executes Reduce task is idle. At the end, the merge task (R_M) can start only after all Reduce tasks finish.

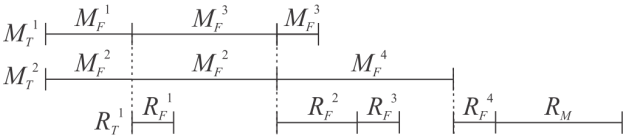

Figure 2. Execution timeline of Hadoop job

In Figure 2, Map tasks are denoted as M_T^i, where i denotes the number of particular Mapper. As shown in Figure 2, two Map tasks, M_T^1 and M_T^2, start execution immediately at the beginning of the job execution, on separate nodes, while the Reduce task (R_T^1) is blocked and, therefore, waits. As soon as the first Map function (M_F^1) finishes, the first Reduce function (R_F^1) can begin its execution. Also, another Map function M_F^3 is assigned to the task M_T^1 that was executing M_F^1. This point in time is shown in Figure 2 with a dotted vertical line. It also represents a synchronization point when the set of functions executing in parallel changes. From this point in time, M_F^3, M_F^2 and R_F^1 are executing. Since M_F^3 starts

executing only after M_F^1 finishes, there is a serial precedence between them.

The execution of a Hadoop job represents a set of synchronization points, and each one of them delimits the parallel execution of different sets of functions. In order to maximize performance due to the synchronization characteristic of Hadoop system, and to utilize parallelization capabilities of Hadoop, the number of Map tasks, Map functions, Reduce tasks and Reduce functions should be carefully planned in accordance to the available number of computational nodes.

III. DATAFLOW OF MATRIX MULTIPLICATION IN HADOOP ENVIRONMENT

We will illustrate parallelization capabilities of Hadoop on the example of matrix multiplication algorithm proposed in [6]. Let us briefly discuss the dataflow timeline of the algorithm from [6], and allocation of computations onto Map and Reduce functions M_F and R_F. Let A and B be matrices of order IxK and KxJ, respectively, and let C be their product as

$$c_{i,j}=\sum_{k=1}^{K} a_{i,k} \cdot b_{k,j} . \quad (1)$$

According to the matrix multiplication algorithm proposed in [6], the value of the key key_{in} that distinguishes the Map functions is common index k from (1). In this case, the total number of Map functions M_F, that are executed by Map tasks, is equal to M_F=K, i.e. to the number of columns in matrix A and the number of rows in matrix B. Map function M_k obtains all partial products $c_{i,j}^{k}=a_{i,k}\cdot b_{k,j}$, where i=1,2,...,I, and j=1,2,...,J.
The example of the multiplication of matrices A and B of order 2x3 and 3x4, respectively, is shown in Figure 3 and Figure 4.

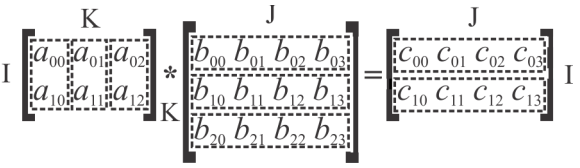

Figure 4. The example of Matrix multiplication $C_{2,4}=A_{2,3}\cdot B_{3,4}$

According to (1), all elements of the first column of the matrix A, i.e. a_{00} and a_{10} in Figure 4, are needed for the multiplication with all elements of the first row of the matrix B, b_{00}, b_{01}, b_{02} and b_{03}. The same holds for other columns of the matrix A, and the rows of the matrix B, as it is shown with dashed lines in Figure 4.
From the above, every Map function M_k will get k-th column of matrix A and k-th row of matrix B, as shown in Figure 3. Within each Map function M_k, every element $a_{i,k}$, i=1,2,...,I, of matrix A will be multiplied with every element $b_{k,j}$, j=1,2,...,J, of matrix B, producing partial products $c_{i,j}^{k}=a_{i,k}\cdot b_{k,j}$. For example, within Map function M_1, the element a_{00} will be multiplied by b_{00}, producing partial result c_{00}^{0}, as denoted with gray circles and arrows in Figure 3. The same stands for all other elements from M_1. As a result, Mapper M_1 will produce first intermediate results for all elements in the resulting matrix C. On the other hand, while the Mappers are responsible for multiplying, Reducers are responsible for summarizing intermediate results $c_{i,j}^{k}$ for every element $c_{i,j}$ in the resulting matrix C. In the example given in Figure 3, c_{00}^{0}, c_{00}^{1} and c_{00}^{2} are summarized into c_{00}.
According to the computations allocation of the particular matrix multiplication algorithm, there is no data dependency between Map functions, and all Map functions can be executed in parallel. The same holds for the Reduce functions.
On the other hand, each Reduce function can start its execution only when all Map functions finish their computations. Therefore, in this algorithm, there is no overlapping between Map and Reduce phase (Figure 3).

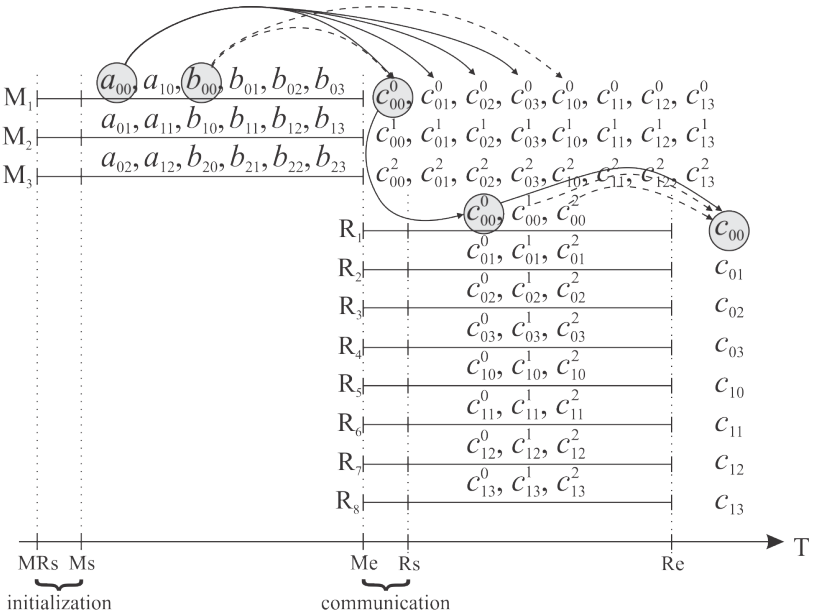

Figure 3. The dataflow timeline of the matrix multiplication algorithm in the MapReduce distributed environment

IV. IMPLEMENTATION RESULTS

In the previous section it was shown how the partial computations are allocated to Map and Reduce functions. As mentioned before, the numbers of Map and Reduce functions are parameters of the algorithm, while Map and Reduce tasks are configured according to the capabilities of the cluster.

For this particular matrix multiplication algorithm, all Map functions can start in parallel at the point denoted with Ms on the T axis in Figure 3. Ideally, the number of nodes N, and the number of the Map tasks M_T should be equal to the number of required Map functions M_F. However, as the number of Map functions M_F is equal to the dimension K of matrices A and B, this number will always in practice overcome the number of available nodes N in the cluster. Therefore, one Map task will execute many Map functions. The same holds for the Reduce functions. All Reduce functions can start in parallel at the point of time denoted as Rs in Figure 3 and last until Re. The number of available Reduce tasks R_T will limit the parallelization in this case, as well.

The algorithm is implemented and executed on the Hadoop cluster consisting of N=18 nodes. The characteristics of nodes are the following: Intel(R) Core(TM)2Duo, CPU E4600@2.40GHz, RAM: 1GB.

We executed the algorithm for two scenarios: (1) fixed number of Reduce tasks, equal to the number of nodes (R_T=N=18), and various number of Map tasks ($1 \leq M_T \leq 2 \cdot N$=36), and (2) fixed number of Map tasks, equal to the number of nodes (M_T=N=18), and various number of Reduce tasks ($1 \leq R_T \leq 2 \cdot N$=36). Let us note that in both cases square matrices of order 1.500x1.500 were considered. Thus, the number of Map functions is M_F=1.500, and the number of Reduce functions is R_F=2.250.000.

The obtained results for the MapReduce algorithm for described scenarios are graphically presented in Figure 5. Let us note that for each result shown in Figure 5 there are M_T+R_T tasks configured. Thus, the minimum number of tasks for the first scenario is 1+18=19, and the maximum is 36+18=54, which are executed on 2·18=36 cores. From the results given in Figure 5 it can be seen that the parallelism is underutilized if the total number of tasks is less then 36 (value M/R=18 in Figure 5), due to the fact that there are unused cores. If the number of tasks is greater then the number of cores (Figure 5), there is additional overhead for synchronization that slows down the execution. Due to the characteristic of the matrix multiplication algorithm, the optimal cluster utilization is when the total number of tasks is equal to the number of cores (Figure 5).

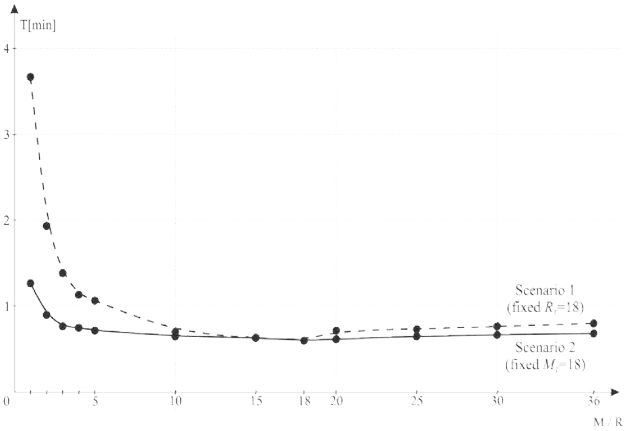

Figure 5. Execution time of MapReduce algorithm for matrix multiplication

V. CONCLUSION

In this paper the analysis of dataflow and parallelization capabilities of Hadoop is illustrated on the example of matrix multiplication algorithm. The dataflow is analyzed through evaluation of the execution timeline of Map and Reduce functions, while the parallelization capabilities are considered through the utilization of Hadoop's Map and Reduce tasks. The results of the implementation for various parameter sets in distributed Hadoop environment consisting of 18 computational nodes are given. It is shown that the optimal cluster utilization is when the total number of tasks is equal to the number of cores.

VI. ACKNOWLEDGMENT

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

VII. REFERENCES

[1] Ciric, Vladimir, et al. "Tropical algebra based framework for error propagation analysis in systolic arrays." Applied Mathematics and Computation 225 (2013): 512-525.

[2] Wang, Guanying, et al. "Using realistic simulation for performance analysis of MapReduce setups." Proceedings of the 1st ACM workshop on Large-Scale system and application performance. ACM, 2009.

[3] Rakumar Buyya, James Broberg, Andrzej Goscinski, "Cloud Computing: Principles and Paradigms", Willey, 2011.

[4] Vianna, Emanuel, et al. "Analytical performance models for MapReduce workloads." International Journal of Parallel Programming 41.4 (2013): 495-525.

[5] Ganapathi, Archana. "Predicting and optimizing system utilization and performance via statistical machine learning." (2009).

[6] Živanović S. Filip, Ćirić M. Vladimir, Stojanović M. Natalija, Milentijević Z. Ivan. "Optimized One Iteration MapReduce Algorithm for Matrix Multiplication". IcETRAN, 2015.

[7] Lam, Chuck. Hadoop in action. Manning Publications Co., 2010.

[8] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: a flexible data processing tool." Communications of the ACM 53.1 (2010): 72-77.

[9] White, Tom. "Hadoop: The definitive guide." O'Reilly Media, Inc., 2012.