BigData and MapReduce with Hadoop

Ivan Tomašić[1], Roman Trobec[1], Aleksandra Rashkovska[1], Matjaž Depolli[1], Peter Mežnar[2], Andrej Lipej[2]
[1]Jožef Stefan Institute, Jamova 39, 1000 Ljubljana
[2]TURBOINŠTITUT d.d., Rovšnikova 7, 1210 Ljubljana, Slovenia
ivan.tomasic@ijs.si, roman.trobec@ijs.si, aleksandra.rashkovska@ijs.si, matjaz.depolli@ijs.si, peter.meznar@turboinstitut.si, andrej.lipej@turboinstitut.si

The paper describes the application of the MapReduce paradigm for processing and analyzing large amounts of data, coming from a computer simulation for a specific scientific problem. The Apache Hadoop open source distribution was installed on a cluster built of six computing nodes, each with four cores. The implemented MapReduce job pipeline is described and the essential Java code segments are presented. The experimental measurements of the employed MapReduce tasks execution times result in a speedup of 20, which indicates that a high level of parallelism is achieved.

I. INTRODUCTION

BigData, by its definition, exceeds the abilities of personal commodity hardware environments. This data is so large that it must be distributed across multiple, possibly thousands of machines, for it to be processed in a reasonable time [1]. Some examples of big data include web logs, social networks data, Internet documents, Internet search indexing, data from scientific research and measurements, medical records, etc.

Besides new approaches and software needed for efficient and secure BigData storage, new massively parallel software platforms have emerged that handle processing and analyzing of such huge amounts of data. Well known and established example is Apache Hadoop Big Data platform with its MapReduce implementation [2].

MapReduce was developed within Google [3] as a mechanism for parallel processing of big data. It is an abstraction that allows performing simple computations while hiding the details of parallelization, data distribution, load balancing and fault tolerance. Typical MapReduce computation processes many terabytes of data on thousands of machines.

We will illustrate the MapReduce application on data from a computer simulation of heat transfer in a biomedical application, in particular, cooling of a human knee after surgery [4]. The data is considered to be big because we simulate wide range of cases in order to cover as much as possible variations in the simulation input parameters, i.e., simulate scenarios with different initial states of the temperature filed before cooling, different knee dimensions, and different temperatures of the cooling liquid.

II. MAPREDUCE PARADIGM

MapReduce is a programming model and an associated implementation for processing and generating large data sets [3]. Some problems that can be simply solved by MapReduce are: distributed grep, count of URL access frequency, various representations of the graph structure of web documents, term-vector per host, inverted index, etc.

The MapReduce user specifies two functions called *Map* and *Reduce*. *Map* takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the *Reduce* function, which accepts the intermediate sets of key/value pairs. It aggregates or merges together these values in some other way to form a new, possibly smaller, set of values. The *Map* and *Reduce* functions, in the Hadoop MapReduce implementation, execute in separate Java Virtual Machines (JVMs) [5]. Their executions are referred to as *Map* and *Reduce* tasks.

The main limitation of the MapReduce paradigm is that each *Map* and *Reduce* task has not to be dependent on any data generated in some other *Map* or *Reduce* task of the current job, as user cannot control the order in which the mappings or reductions execute.

The MapReduce, as a paradigm, may have different implementations. For the purpose of solving our problem we have used MapReduce implemented in Apache Hadoop [2] and distributed in Cloudera Hadoop distribution [6].

A. Apache Hadoop MapReduce implementation

Apache Hadoop is an open-source software framework aimed for developing data-intensive distributed applications that can run on large clusters of commodity hardware. It is licensed under the Apache v2 license. Hadoop was originally developed by Yahoo! as a clone of Google's MapReduce and Google File System (GFS).

The Hadoop project is comprised of four modules: Hadoop Common, Hadoop Distributed File System (HDFS), Hadoop YARN and Hadoop MapReduce [2].

At the beginning of a MapReduce job, Hadoop divides the input data to be processed into fixed-size pieces called splits, and creates a separate *Map* task for each split. Each *Map* task runs the *Map* function for each record in the split. The splitting is introduced because it is generally shorter to process each split, compared to the time needed to process the entire input data as an entirety.

The parallel processing can be better load-balanced if the splits are small. However, if the splits are too small, then the time needed to manage the splits and the time for *Map* task creation may begin to dominate the total job execution time, resulting in an inefficient run. The default split size is the same as the default size of an HDFS block, which is 64 MB. Besides splitting, Hadoop does also the data locality optimization by trying to run the *Map* task on a node where the input data resides in the HDFS.

Map tasks write their outputs to their local disks, not to the HDFS. The *Map* outputs are therefore not replicated. If an error happens on a node running a *Map* task before its output has been consumed by a *Reduce* task, then Hadoop resolves the error by re-running the corrupted *Map* task on another node to recreate the required output.

The *Map* tasks partition their outputs, each creating one partition for each *Reduce* task. Each partition may contain many different keys and associated values. Each *Map* task offers all the records for a given key only to a single *Reduce* task. That is accomplished by packing all the records for a specific key in a single partition. The records are distributed to *Reduce* tasks and merged in a way that all records sharing the same key, will be processed by the same *Reduce* task.

In Hadoop implementation, all *Reduce* tasks wait until all *Map* tasks complete or fail [7]. *Reduce* tasks, unlike *Map* tasks, cannot convey on data locality because the input to a single *Reduce* task is generally formed from outputs of multiple *Map tasks*. Each *Map* task's output is firstly sorted and then transferred across the network to the node where its corresponding *Reduce* task is running. The sorted map outputs are merged on this node, before being passed to the *Reduce* task to be executed on this node. The number of *Reduce* tasks is specified independently for a given job. Each *Reduce* task outputs a single file, which is usually stored in the HDFS.

Figure 1 shows schematically the MapReduce data flow. We see that each *Reduce* task is fed by multiple *Map* tasks; therefore the data flow between *Map* and *Reduce* tasks is colloquially known as "shuffle".

Additionally to the *Map* and *Reduce* functions, Hadoop allows the user to specify a so called *Combiner* function, which is run on each node that has run *Map* tasks. It receives all the data emitted by the *Map* tasks on a given node as input, and forms the output that is an input to a *Reduce* function. The *Combiner* function is used to achieve data reduction before sending it over the network to a *Reduce* task, in order to minimize data transfer between *Map* and *Reduce* tasks. Such an approach reduces the influence of available bandwidth on the performances of a MapReduce job.

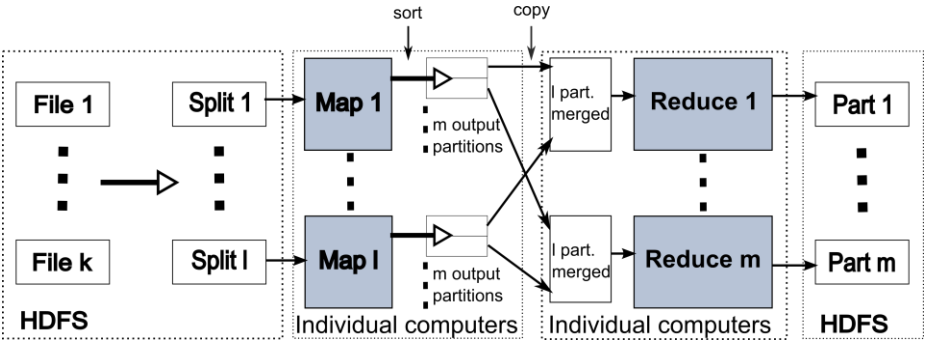

Figure 1. Schematic representation of MapReduce data flow

III. ANALYZING SIMULATION DATA WITH MAPREDUCE

A. Source of input data

The computer simulation of two hours cooling of a human knee after surgery is performed for 10 different knee sizes, 10 different initial temperature states before cooling, and 10 different temperatures of the cooling pad. This results in 1000 simulation cases. The results are gathered in 100 files, each for one knee size and one initial state, and for all cooling temperatures. Each file contains 71 970 rows or approximately 44 MB of data. Each data row is composed of the following parameters: RT, D, IS, CT, T1, T2, … , T85, where are: RT - relative time in a simulation case, D - knee size, IS –initial state, CT – cooling temperature, T1-T85 – inner and outer knee temperatures, i.e. temperatures at a particular location in the knee center, 8 locations on the knee skin and 8 respective locations under the cooling pad, all taken in the

current and in previous time steps. The parameters are comma separated. In our test case, the task addressed to MapReduce was to find the number of occurrences of a certain set of parameters with the same values, i.e., knee temperatures T from different locations at the current or at several time steps. We defined and examined 8 cases with different sets of T parameters as a key in question. The cases are given in Table 1. The ones with odd numbering take only the current values for the temperatures at: Case 1 – the knee center; Case 3 – the knee center and 4 locations on the knee skin; Case 5 – the knee center, 4 locations on the knee skin, and 4 respective locations under the cooling pad; Case 7 – all current temperatures. Their respective cases with temperatures taken also in 4 previous time steps are the ones with even numbering.

Table 1. Test cases

CASE	***Parameters***
1	T1
2	T1-T5
3	T1,T6,T11,T16,T21
4	T1-T21
5	T1,T6,T11,T16,T21,T46,T51,T56,T61
6	T1-T21,T46-T61
7	T1,T6,T11,T16,T21,T26,T31,T36,T41,T46,T51,T56,T61,T66,T71,T76,T81
8	T1-T85

B. MapReduce jobs

The MapReduce jobs pipeline, used for solving our test cases, is illustrated in Figure 2. The input data consists of 100 files each approximately being 44 MB in size. The sizes of files are smaller than the HDFS block size, which is in our case 64 MB, hence the number of input *Map* tasks in Job 1 is equal to the number of input files [8], i.e., each input file is processed by a different *Map* task. Because the number of *Reduce* tasks is not explicitly set for Job 1, it becomes, by default, equal to the number of task tracker nodes, in our case 6, multiplied by the value of the `mapred.tasktracker. .reduce.tasks.maximum` configuration property [8], which is in our case 2. The output of Job 1 is therefore consisted of 12 files. Each file contains unique combinations of temperatures and the number of their occurrences. The details of the jobs implementations are given in Figure 3 and in the text below.

Then, Job 2 combines *Reduce* tasks' outputs form Job 1 into a single file (in Job 2, the number of *Reduce* tasks is explicitly set to 1). It also substitutes the input columns and sorts them in the output file by the number of occurrences of each combination of temperatures. The number of *Map* tasks in Job 2 depends on the test case (The computer simulation of two hours cooling of a human knee after surgery is performed for 10 different knee sizes, 10 different initial temperature states before cooling, and 10 different temperatures of the cooling pad. This results in 1000 simulation cases. The results are gathered in 100 files, each for one knee size and one initial state, and for all cooling temperatures. Each file contains 71 970 rows or approximately 44 MB of data. Each data row is composed of the following parameters: RT, D, IS, CT, T1, T2, … , T85, where are: RT - relative time in a simulation case, D - knee size, IS –initial state, CT – cooling temperature, T1-T85 – inner and outer knee temperatures, i.e. temperatures at a particular location in the knee center, 8 locations on the knee skin and 8 respective locations under the cooling pad, all taken in the current and in previous time steps. The parameters are comma separated. In our test case, the task addressed to MapReduce was to find the number of occurrences of a certain set of parameters with the same values, i.e., knee temperatures T from different locations at the current or at several time steps. We defined and examined 8 cases with different sets of T parameters as a key in question. The cases are given in Table 1. The ones with odd numbering take only the current values for the temperatures at: Case 1 – the knee center; Case 3 – the knee center and 4 locations on the knee skin; Case 5 – the knee center, 4 locations on the knee skin, and 4 respective locations under the cooling pad; Case 7 – all current temperatures. Their respective cases with temperatures taken also in 4 previous time steps are the ones with even numbering.

Table 1) and varies between 12 and 36 because the sizes of Job 1 output files depend on the test case.

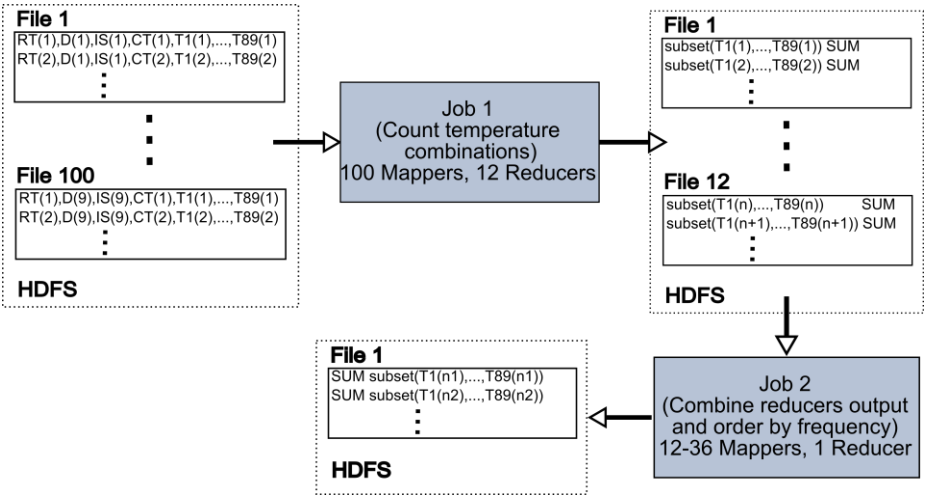

Figure 2. MapReduce jobs pipeline

In Figure 3. , the Java code segments for *Map* and *Reduce* functions are shown.

```
//Job 1

public void map(LongWritable key,Text value,OutputCollector<Text,IntWritable> output,
Reporter reporter) throws IOException{
        String line = value.toString();
        String[] lineElements  = line.split(",");
        String SearchString = null
        //depending on a case (Table 1) concatenate different lineElements in
        //SearchString
        …
        word.set(SearchString);
        output.collect(word, new IntWritable(1));
}

public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException{
        int sum = 0;
        while (values.hasNext()){
                sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));

}
```

```
//Job 2
public void map(LongWritable key, Text value, OutputCollector<IntWritable,Text> output,
Reporter reporter) throws IOException{
        String line = value.toString();
        //\t is the default delimiter used by a reducer
        String[] lineElements  = line.split("\t");
        output.collect(new IntWritable(Integer.parseInt(lineElements[1])),
                                                        new Text(lineElements[0]));
}
public void reduce(IntWritable key, Iterator<Text> values, OutputCollector<IntWritable,
Text> output, Reporter reporter) throws IOException{
        //there is only one value
        output.collect(key, values.next());
}
```

Figure 3. Java code of *Map* and *Reduce* tasks for Job 1 and Job 2.

In the *Map* function of Job 1, from each input comma separated line, only the relevant columns (see The computer simulation of two hours cooling of a human knee after surgery is performed for 10 different knee sizes, 10 different initial temperature states before cooling, and 10 different temperatures of the cooling pad. This results in 1000 simulation cases. The results are gathered in 100 files, each for one knee size and one initial state, and for all cooling temperatures. Each file contains 71 970 rows or approximately 44 MB of data. Each data row is composed of the following parameters: RT, D, IS, CT, T1, T2, … , T85, where are: RT - relative time in a simulation case, D - knee size, IS –initial state, CT – cooling temperature, T1-T85 – inner and outer knee temperatures, i.e. temperatures at a particular location in the knee center, 8 locations on the knee skin and 8 respective locations under the cooling pad, all taken in the current and in previous time steps. The parameters are comma separated. In our test case, the task addressed to MapReduce was to find the number of occurrences of a certain set of parameters with the same values, i.e., knee temperatures T from different locations at the current or at several time steps. We defined and examined 8 cases with different sets of T parameters as a key in question. The cases are given in Table 1. The ones with odd numbering take only the current values for the temperatures at: Case 1 – the knee center; Case 3 – the knee center and 4 locations on the knee skin; Case 5 – the knee center, 4 locations on the knee skin, and 4 respective locations under the cooling pad; Case 7 – all current temperatures. Their respective cases with temperatures taken also in 4 previous time steps are the ones with even numbering.

Table 1) are extracted. For example, in the test case 2 only the columns belonging to T1 to T5 will be extracted in the SearchString variable. The *Reduce* function sums, i.e., counts the number of occurrences of each combination of temperatures (the key) and outputs it as the new value of the current key. Because all the values for one key are processed by a single *Reduce* task, it is evident that the output from Job 1 consists of unique combinations of temperatures and the number of their occurrences.

In Job 2, the *Map* function inverts its key/value pairs, making temperature occurrences the keys, and emits them to the *Reduce* function, which outputs the received pairs. The sorting by the occurrences is done by the framework as explained in section II.

C. Results

The ten highest numbers of temperature occurrences, for each test case from Table 1, are given in Table 2.

Table 2. Top 10 temperature combinations occurrences for each case

Case 8	*Case 7*	*Case 6*	*Case 5*	*Case 4*	*Case 3*	*Case 2*	*Case 1*
294	298	294	298	387	391	8933	11159
224	228	224	228	319	323	8860	11097
211	215	217	227	294	298	8778	10945
181	199	216	221	267	271	8351	10924
168	194	211	220	232	264	7807	10729
165	185	187	215	231	256	7695	10720
161	185	181	215	224	253	7626	10706
159	183	175	199	224	248	7551	10645
158	172	168	199	216	247	7504	10602
154	172	165	195	216	245	7456	10591

We see that the lowest numbers appear in test case 8, which could be expected because in case 8 the largest number of parameters (T) was analyzed.

Table 3 shows the execution times, for Job 1 and Job 2, respectively, depending on the test cases. It also shows the total CPU time, spent for each case, and associated total duration of the analysis.

The total time spent for *Maps* and *Reduces* in Job 1 and Job 2 for all test cases and on all executing nodes is: $t_s = 9903 + 1264 + 941 + 139 = 12247$ s, while

the total duration of the complete MapReduce analysis is: t_m = 377 + 248 = 625 s. The ratio t_s/t_m is 19.6, which can assess the level of parallelism achieved. The analysis is almost 20 times faster if implemented by MapReduce paradigm.

Table 3. MapReduce tasks execution times

Job1									
Case:	1	2	3	4	5	6	7	8	Total
Total time spent by all maps in (s)	1,122	1,080	1,119	1,187	1,121	1,287	1,162	1,826	**9,903**
Total time spent by all reduces (s)	100	80	91	148	108	207	118	413	**1,264**
Map tasks avg. time (s)	11	10	11	11	11	12	11	18	
The last *Map* task finished at (s)*	33	31	32	35	33	35	32	54	
Shuffle avg. time (s)	5	3	3	7	5	7	5	14	
The last *Shuffle* task finished at (s)*	36	33	33	36	35	39	35	57	
Reduce tasks avg. time (s)	2	2	3	5	3	9	4	20	
The last *Reduce* task finished at(s)*	39	36	37	42	39	49	39	79	
CPU time spent (s)	588	618	667	790	686	933	719	1,494	6,494
Total duration (s)	40	37	38	43	49	51	40	79	**377**

Job2									
Case:	1	2	3	4	5	6	7	8	Total
Total time spent by all maps in (s)	32	31	51	78	59	184	64	443	**941**
Total time spent by all reduces (s)	4	4	10	16	12	31	12	50	**139**
Map tasks avg. time (s)	2	2	3	6	4	7	5	12	
The last *Map* task finished at (s)*	7	7	12	14	15	13	12	20	
Shuffle avg. time (s)	1	1	6	5	6	7	4	8	
The last *Shuffle* task finished at (s)*	8	10	15	16	19	22	13	30	
Reduce tasks avg. time (s)	1	1	3	10	5	23	8	40	
The last *Reduce* task finished at(s)*	10	12	19	26	24	45	21	71	

CPU time spent (s)	7	9	55	95	70	185	73	330	823
Total duration (s)	13	14	22	28	26	48	24	73	**248**

* relative to the Job launch time

Although the interpretation of the temperature values and their occurrences in a specified combination are behind the scope of this paper, each execution case (The computer simulation of two hours cooling of a human knee after surgery is performed for 10 different knee sizes, 10 different initial temperature states before cooling, and 10 different temperatures of the cooling pad. This results in 1000 simulation cases. The results are gathered in 100 files, each for one knee size and one initial state, and for all cooling temperatures. Each file contains 71 970 rows or approximately 44 MB of data. Each data row is composed of the following parameters: RT, D, IS, CT, T1, T2, … , T85, where are: RT - relative time in a simulation case, D - knee size, IS –initial state, CT – cooling temperature, T1-T85 – inner and outer knee temperatures, i.e. temperatures at a particular location in the knee center, 8 locations on the knee skin and 8 respective locations under the cooling pad, all taken in the current and in previous time steps. The parameters are comma separated. In our test case, the task addressed to MapReduce was to find the number of occurrences of a certain set of parameters with the same values, i.e., knee temperatures T from different locations at the current or at several time steps. We defined and examined 8 cases with different sets of T parameters as a key in question. The cases are given in Table 1. The ones with odd numbering take only the current values for the temperatures at: Case 1 – the knee center; Case 3 – the knee center and 4 locations on the knee skin; Case 5 – the knee center, 4 locations on the knee skin, and 4 respective locations under the cooling pad; Case 7 – all current temperatures. Their respective cases with temperatures taken also in 4 previous time steps are the ones with even numbering.

Table 1) draws different amounts of data to the *Map* and *Reduce* functions in Job 1 and Job 2, which influences their execution times, as evident from Table 3.

IV. CONCLUSION

We successfully implemented the analysis of a large amount of simulation results with two MapReduce jobs. The pipelining between jobs can be further refined if a need occurs. For example, an additional job may be inserted in the pipeline if some kind of filtering of temperature combinations is required. The described application is quite general and can be applied in a similar way on

other data sets, e.g. the datasets coming from computer simulations of hydro turbines for the purpose of maximization of their efficiency, which is performed routinely at Turboinštitut Ljubljana, Slovenia.

The measured execution times show that the speed-up of approximately 20 is achieved by MapReduce in the solution of our test cases, interestingly, on a computing cluster with only 6 TaskTracker nodes, however, each with 4 cores.

Although MapReduce concept has proved to be very efficient for solving our analysis of test cases, further test are needed to assess its efficiency on more nodes and on larger files. There are indications that parallel relational database management systems are also up for the challenges that BigData imposes [9]. We plan to investigate the abilities of some parallel relational databases and compare their efficiency with the MapReduce paradigm.

ACKNOWLEDGEMENTS

This research was funded in part by the European Union, European Social Fund, Operational Programme for Human Resources, Development for the Period 2007-2013.

REFERENCES

[1] B. Franks, "What is big data and why does it matter?," *Taming the big data tidal wave: finding opportunities in huge data streams with advanced analytics*, pp. 3-29, Hoboken, New Jersey: John Wiley & Sons, Inc., 2010.

[2] "Welcome to Apache™ Hadoop®!," Oct., 2012; http://hadoop.apache.org/.

[3] J. Dean, and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the Acm,* vol. 51, no. 1, pp. 107-113, Jan, 2008.

[4] R. Trobec, M. Šterk, S. Almawed *et al.*, "Computer simulation of topical knee cooling," *Comput. biol. med,* vol. 38, pp. 1076-1083, 2008.

[5] T. White, "How MapReduce Works," *Hadoop: The Definitive Guide*, pp. 167-188, Gravenstein Highway North, Sebastopol: O'Reilly Media, Inc., 2010.

[6] "CDH Proven, enterprise-ready Hadoop distribution – 100% open source," Oct, 2012; http://www.cloudera.com/hadoop/.

[7] T. White, "MapReduce," *Hadoop: The Definitive Guide*, pp. 15-40, Gravenstein Highway North, Sebastopol: O'Reilly Media, Inc., 2010.

[8] T. White, "MapReduce Types and Formats," *Hadoop: The Definitive Guide*, pp. 189-224, Gravenstein Highway North, Sebastopol: O'Reilly Media, Inc., 2010.

[9] A. Pavlo, E. Paulson, A. Rasin *et al.*, "A comparison of approaches to large-scale data analysis," in SIGMOD-PODS'09 2009, pp. 165-178.