Performance evaluation of Apache Hadoop and Apache Spark for parallelization of compute-intensive tasks

Alexander Döschl
Max-Emanuel Keller
Peter Mandl
alexander.doeschl@hm.edu
max-emanuel.keller@hm.edu
peter.mandl@hm.edu
HM Hochschule München University of Applied Sciences
Munich, Germany

ABSTRACT

There have been numerous studies that have examined the performance of distribution frameworks. Most of these studies deal with the processing of large amounts of data. This work compares two of these frameworks for their ability to implement CPU-intensive distributed algorithms. As a case study for our experiments we used a simple but computationally intensive puzzle. To find all solutions using brute-force search, 15! permutations had to be calculated and tested against the solution rules. Our experimental application was implemented in the Java programming language using a simple algorithm and having two distributed solutions with the paradigms MapReduce (Apache Hadoop) and RDD (Apache Spark). The implementations were benchmarked in Amazon-EC2/EMR clusters for performance and scalability measurements, where the processing time of both solutions scaled approximately linearly. However, according to our experiments, the number of tasks, hardware utilization and other aspects should also be taken into consideration when assessing scalability. The comparison of the solutions with MapReduce (Apache Hadoop) and RDD (Apache Spark) under Amazon EMR showed that the processing time measured in CPU minutes with Spark was up to 30 % lower, while the performance of Spark especially benefits from an increasing number of tasks. Considering the efficiency of using the EC2 resources, the implementation via Apache Spark was even more powerful than a comparable multithreaded Java solution.

CCS CONCEPTS

• **Computing methodologies** → **MapReduce algorithms**; • **Applied computing** → **Enterprise architecture frameworks**.

KEYWORDS

Distributed Systems, Cluster Computing, Cloud Computing, Apache Hadoop, Apache Spark

iiWAS '20, November 30-December 2, 2020, Chiang Mai, Thailand

ACM ISBN 978-1-4503-8922-8/20/11…$15.00
https://doi.org/10.1145/3428757.3429121

ACM Reference Format:
Alexander Döschl, Max-Emanuel Keller, and Peter Mandl. 2020. Performance evaluation of Apache Hadoop and Apache Spark for parallelization of compute-intensive tasks. In *The 22nd International Conference on Information Integration and Web-based Applications & Services (iiWAS '20), November 30-December 2, 2020, Chiang Mai, Thailand.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3428757.3429121

1 INTRODUCTION

The minimization of processing times is one of the primary goals within computer science. However, since the possibilities for optimization within a computer are limited, it is reasonable to divide tasks that exceed a certain size among several computers for processing.

For parallelization of computationally intensive tasks GPUs are used in some cases, but the necessary hardware is not always available. In addition, GPU programming often requires major code adjustments and is not directly usable with many programming languages, which makes a rapid implementation process difficult. However, since many companies already have CPU focused infrastructures, it would be desirable to be able to run CPU-intensive tasks parallel on these systems.

Although frameworks from the big-data environment are primarily designed for the processing of large amounts of unstructured data, this does not exclude their use in other application scenarios [17, 20]. Since such technologies usually do not place high demands on the underlying hardware and usually offer interfaces for different programming languages, big-data frameworks can also provide a simple way to parallelize complex tasks. An example for this are machine learning libraries, which show that computationally intensive tasks can be solved very efficiently using big-data technologies, although the processed data is often small.

While previous studies almost only dealt with the suitability for the processing of large amounts of data, a comparison of different frameworks regarding their performance on CPU focused algorithms is missing. In this work the two frameworks Apache Hadoop and Apache Spark are being studied with regard to their suitability for fulfilling such tasks. As a case study the puzzle game Game 30 is used, that can be solved with a computationally intensive algorithm. To gain insights about the performance and scalability of both frameworks, solving algorithms were created with both frameworks, which were then used to perform measurements on the systems of Amazon Web Services (AWS) [4].

2 CASE EXAMPLE: GAME 30

The arithmetic game Game 30 [10] used in our case study has the objective of arranging 15 game pieces numbered from 1 to 15 in a square, so that the sum of the vertical, horizontal and diagonal lines (see Fig. 1) is 30 in each case.

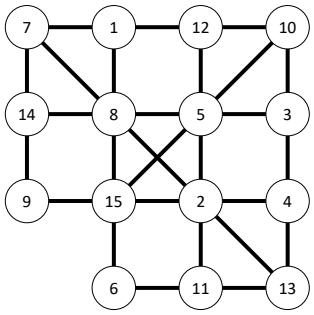


Figure 1: Game 30 with correct solution

For a valid solution, a total of ten sums (four column sums, four row sums and two diagonal sums) must match. If you want to find all of the 416 solutions with the brute-force approach, which means by trial and error, you are confronted with an enormous challenge, since every swapping of the game pieces requires a relatively complex examination. With 15 pieces there are at least 15! $\approx$ 1,3 quadrillion (exactly: 1,307,674,368,000) different possibilities to arrange the pieces. An arrangement of the game pieces as vector V with 15 elements can be formalized as in Eq. (1).

$$V = (e_1, e_2, \ldots, e_{15}); e_i \in W = 1, 2, 3, \ldots, 15; \\ i, j \text{ in } 1 \ldots 15; e_i \neq e_j \tag{1}$$

Each element e_i represents one piece. Each number from the set of values may only be assigned once during an assignment, repetitions are not allowed. Every vector V represents a concrete permutation without repetition from the value set W. As shown in Listing 1, the rules can be formulated as a system of equations with 10 equations and 15 variables, whereby the variables can only take values from W according to the convention explained before.

Listing 1: Equations of the rules of Game 30

```
V[00] + V[01] + V[02] + V[03] = 30; // Row 1
V[04] + V[05] + V[06] + V[07] = 30; // Row 2
V[08] + V[09] + V[10] + V[11] = 30; // Row 3
V[12] + V[13] + V[14]         = 30; // Row 4
V[00] + V[04] + V[07]         = 30; // Column 1
V[01] + V[05] + V[09] + V[12] = 30; // Column 2
V[02] + V[06] + V[10] + V[13] = 30; // Column 3
V[03] + V[07] + V[11] + V[14] = 30; // Column 4
V[00] + V[05] + V[10] + V[14] = 30; // Diagonal 1
V[03] + V[06] + V[09]         = 30; // Diagonal 2
```

A total of 416 permutations result in a valid solution in the system of equations. Since a complete solution of the game has a significant complexity, we decided to use this calculation game as an example for our performance and scalability evaluation of parallel algorithms.

3 DISTRIBUTED ALGORITHMS

A simple algorithm for the identification of valid solutions for Game 30 can be defined as shown in Listing 2 [18].

Listing 2: Single thread algorithm for Game 30

```
findSolutions() {
  // Initialize vector v with (1, 2, ..., 14, 15);
  while (another permutation exists) {
    Get next permutation perm(V);
    if (rulesSatisfied(perm))
      Print solution;
  }
}
```

The algorithm permutates the pieces of vector V and applies the rules for valid solutions to each permutation. If all conditions are fulfilled, a valid solution is found and can be returned. The execution of the permutation was implemented as described in [18], based on the lexicographic iteration [5].

In order to adapt the algorithm shown in Listing 2 for distributed processing, the vector that we want to permutate must be split. This means that a thread processes only a subset of the possible positions. A possible implementation is presented in Listing 3. We instantiate a new thread for any value the first piece of the game can have so that one program run will process the problem running 15 thread instances. Each individual thread of the algorithm permutates the remaining 14 game pieces and connects them with the respective first game piece to possible solutions, which are compared with the game rules.

Listing 3: Multi-thread algorithm for Game 30

```
for (i = 1; i < 16; i++) {
  Thread.run(findSolutions(i, pawnsToPerm);
}

findSolutions(staticLeadPawns, pawnsToPerm) {
  while (next permutation exists in pawnsToPerm) {
    Get next permutation perm(pawnsToPerm);
    if (rulesSatisfied(staticLeadPawns + perm)) {
      Print solution;
    }
  }
}
```

15 threads executed in parallel can reduce the runtime considerably, but a cluster often has even more cores that can be used. For this purpose, it is expedient to use a parallel computing framework that breaks down jobs into input splits and passes them as independent tasks for processing. Such a parallelization can be implemented in a simple way. If you vary the first two pieces, you get 210 (15×14) different combinations that can be mapped to input splits, whereas with three pieces there are already 2, 730 ($15 \times 14 \times 13$) input splits, and so on. Each input split can be assigned to a separate task for computation. Even though increasing the number of static pieces can have a positive effect on the distribution of the workload, this does not always have to result in a reduction of the runtime, as a new task must be initialized for each input split, which is associated with some overhead.

3.1 MapReduce algorithm with Apache Hadoop

Apache Hadoop is a framework programmed in Java that allows the development of distributed applications that can process large amounts of data. It enables horizontal scaling to thousands of computers (nodes) [6, 25]. The Hadoop framework consists of two components, the Hadoop Distributed File System (HDFS) and an implementation of the MapReduce paradigm. HDFS is used to share files by dividing them into smaller chunks and store them redundantly in order to provide them for many computing processes, which is of no great importance in our scenario. MapReduce is a programming model for parallelizing computing operations. This involves two phases, the map phase and the reduce phase, which are defined as Java methods. The map phase is supported by all nodes and solves a subproblem with the same Java method on each computer. The intermediate results are combined to form an overall solution in the reduce phase, which runs in a single reduce method in the standard configuration. If the merging of data is not necessary, you can omit the reduce phase and use a map-only job. Some even more complex problems, however, cannot be solved in a single MapReduce sequence, which makes it necessary to execute several MapReduce sequences in a row.

To distribute our calculation, we take the usual Hadoop mechanisms. To allow more than 15 map tasks to work in parallel, we need to parallelize more fine-grained. This can be done by specifying the first two game pieces for each map task. Since the first piece can take 15 different values, there are only 14 possibilities for the allocation of the second piece, so that altogether 210 possible combinations exist. All combinations of the two pieces are written to a common file, which is the input for the following Hadoop processing. For this each combination is stored in a separate line. Hadoop typically uses an `InputFormat`, which reads certain ranges from the input files and defines key-value pairs from them. These pairs are passed to the individual mapper instances, which can process them using the method `map()`. We use this function by first creating a file without Hadoop that holds all possible combinations of the first two pieces $((1,2),(1,3),\ldots,(15,14))$ of the game. With the specially configured `NLineInputFormat` our Hadoop application now reads the created file and assigns a new mapper instance to each line. The mapper instance then connects the passed static pieces with the remaining pieces to be permuted to possible solutions. This way, the solution search can be distributed over 210 (with two static pieces) or 2,730 (with three static pieces) concurrent map tasks. How the solution search works is described in [18] in greater detail. Afterwards the valid solutions found are passed to a single reducer instance, that combines the results and outputs them numbered in one file. In Listing 4 and 5 the mapper and the reducer class are illustrated.

With regard to the communication between MapReduce tasks, it should be noted that it would not be necessary to declare global variables for status queries, since Hadoop already includes such a function. However, for better comparability with other solutions, we still use a global counter variable for the performed permutations. For this purpose, we decided to use the class `Counter`, whose objects are centrally managed in Hadoop. Even though the functionality is very limited, a periodic, asynchronous and efficient summation can be performed over several tasks. The total is returned at the end of processing.

Listing 4: Mapper class from Game 30 MapReduce job

```
public static class Game30Mapper extends
    Mapper<Object, Text, Text, NullWritable> {

  private Game30 game = new Game30();

  public void map(Object key, Text value,
                  Context context) {
    // Splits the read pawns on separator and
    // converts it into an array
    Integer[] staticLeadPawns =
        stringArrayToIntegerArray(
            value.toString().split("\\|"));

    // Deletes the static pawns from the set of
    // all pawns to get the pawns to permute
    Integer[] pawnsToPerm = ALL_PAWNS;
    for (Integer staticLeadPawn : staticLeadPawns) {
      pawnsToPerm = removeElement(pawnsToPerm,
          Arrays.asList(pawnsToPerm)
              .indexOf(staticLeadPawn));
    }

    game.findSolutions(staticLeadPawns,
                       pawnsToPerm, context);
  }
}
```

Listing 5: Reducer class from Game 30 MapReduce job

```
public static class Game30Reducer extends
    Reducer<Text, NullWritable, Text, NullWritable> {

  private int solutionNum;

  public void reduce(Text key,
                     Iterable<NullWritable> values,
                     Context context) {
    solutionNum++;
    context.write(new Text("Solution " + solutionNum
                           + ":\n" + key),
                  NullWritable.get());
  }
}
```

The architecture of the Hadoop implementation is illustrated in Fig. 2. Components of HDFS are not taken into account. The master node receives the MapReduce job from the client and hands it over to one or more worker nodes, which perform the actual computation. While the node managers are responsible for the container creation and resource monitoring on each node, the application master controls the sequentially task execution inside the containers [6].

3.2 Algorithm with Apache Spark

It has already been pointed out that some problems with Apache Hadoop can only be solved by running several MapReduce sequences in a row. This leads in intermediate results that have to be stored on the distributed file system after or before the individual steps and must be read in again. The operations for accessing the

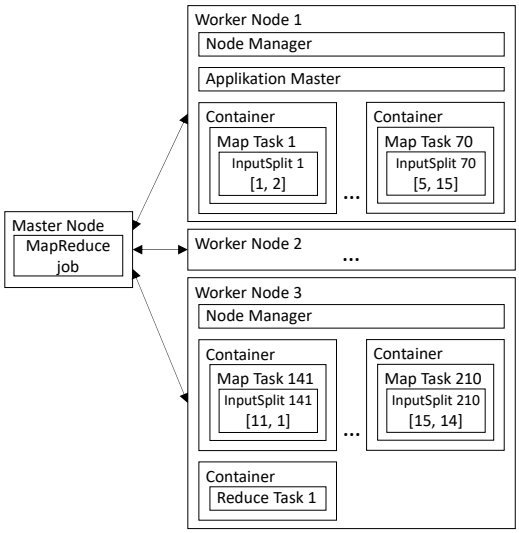


Figure 2: Architecture of the Hadoop solution

file system (I/O operations), the necessary serialization of objects, and the replication of data can dominate the execution time of a Hadoop application. In order to make such use cases more efficient, the framework Apache Spark was developed [26]. Spark is written in the functional language Scala, which also runs in the JVM and is fully compatible to Java. That is why Spark also includes a Java API based on the Scala classes.

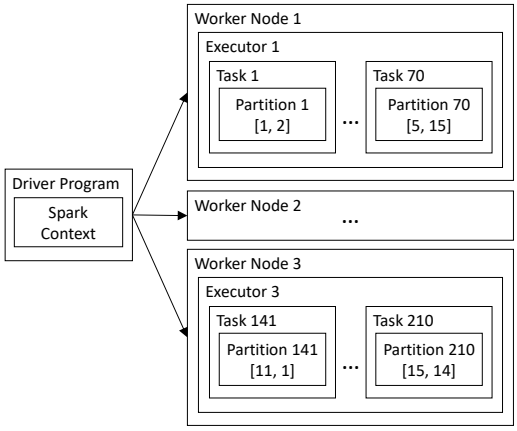


Figure 3: Architecture of the Spark solution

Like Apache Hadoop, Apache Spark uses the distributed file system HDFS. However, instead of the paradigm MapReduce it uses the abstraction RDD (Resilient Distributed Dataset). RDDs are read-only collections of objects that are stored as partitions distributed over the cluster in the memory of the participating nodes. An RDD can be created by operations from data stored on a distributed file system or an existing RDD. If changes are made to the data, new RDDs are derived from the old RDDs, which are also stored in the fast RAM. As a result, the time-consuming intermediate storing and reading of data to and from the distributed file system is no longer necessary. The operations of the API can be divided basically into two groups: transformations and actions. The group of transformations is used to transform current RDDs into new RDDs and includes functions like map, join or filter. Each individual element of an RDD is processed using this function and the result is transferred to a new RDD. Instead of changes to the data, only a lineage of the operations used is recorded. Transformations are only executed with a delay, when they are actually used. Operations from the group of actions represent such usages as they return results in a certain form or write them to a distributed file system. If parts of an RDD or individual partitions are lost due to an error, they can be re-computed based on the lineage information to achieve error tolerance and scalability as in MapReduce. However, Spark has the advantage over Hadoop that these additional costs only occur in case of an error.

Besides the advantages of Apache Spark in handling and maintaining data, we also assume performance advantages for applications that almost exclusively involve computing operations. Our case study represents such a task and was transferred to a Spark application by using the common Spark operations.

For the Hadoop solution, we first had to create an input file in HDFS, but for processing with Spark it is sufficient to create an object of the Java class List that holds all possible combinations of the static pieces. From this list a RDD data can be generated, where every possible combination of the static pieces represents an element of the RDD (see Listing 6).

Here it should be noted that the elements of an RDD are divided into partitions and each partition can be processed by a single task. For the greatest possible parallelization, each element must therefore have its own partition. This ensures that the search for solutions can be calculated with 210 (for two static pieces) or 2,730 (for three static pieces) concurrent tasks. The distribution of elements and partitions among the cluster's worker nodes is illustrated by the architecture depicted in Fig. 3.

The architecture consists of a driver program, which controls the execution of the program, and one or more worker nodes, which perform the actual calculation. For this purpose, each worker has an executor that sequentially executes a series of tasks that search for solutions. The figure shows 210 tasks that are executed evenly distributed on the nodes to transform the static pieces into an RDD. The result of each task is a partition of the RDD `data` that contains a pair of the static pieces and is stored with the task.

In order to transform the static pieces in the RDD `data` into valid solutions, they must be permutated with the remaining pieces and validated against the rules (see Listing 6, line 19). Here it is important to note that for every single combination of the static pieces there can be zero or more solutions. In order to handle this, the operation `flatMap()` is called, which makes it possible to transform each element of an RDD into none, one or even more elements of a new RDD. The transformation of the elements into new elements is done by a method, which is passed to the operation `flatMap()` and called for each element one after another. In our case it is the method `calculateSolutions()`, which converts each of the combinations of static pieces into zero or more valid solutions, as shown in the Spark program in Listing 7. The method `calculateSolutions()` is based on the well-known method `findSolutions()`, shown in Listing 2, which connects the static pieces with the remaining

Listing 6: Main program of the Spark implementation

```
private static void playGame30Spark(
                      String outputDir) {
  try {
    // Permutations of the static lead pawns
    List<Integer[]> initList =
                     new ArrayList<Integer[]>();
    ...
    // Transform static pawns to RDD
    JavaRDD<Integer[]> data =
                  javaSparkContext.parallelize(
                    initList, initList.size());

    // Accumulator for count of solutions
    LongAccumulator numberOfPermutations =
                  sc.longAccumulator(
                    "number_of_permutations");

    // Find solutions
    JavaRDD<String> solutionText = data.flatMap(
               Game30Spark.calculateSolutions(
                    numberOfPermutations));

    // Repartition solutions
    JavaRDD<String> solutionSinglePartition =
                     solutionText.repartition(1);

    // Zip solutions with numeric index
    JavaPairRDD<String, Long> solutionsIndexed =
          solutionSinglePartition.zipWithIndex();

    // Enumerate solutions with index
    JavaRDD<String> solutionsEnumerated =
             solutionsIndexed.map(
               Game30Spark.enumerateSolutions());

    // Persist solutions to HDFS
    solutionsEnumerated
       .saveAsTextFile(outputDir);

    System.out.println(
       "Calculated permutations: "
          + numberOfPermutations.value());
    ...
}
```

Listing 7: Functions for calculating and enumerating the solutions

```
public static class Game30Spark {
  public static FlatMapFunction<Integer[], String>
      calculateSolutions(
        LongAccumulator numPermutations) {
    return e -> {
      Game30 game30 = new Game30();
      ArrayList<String> solutionsFound;

      Integer[] pawnsToPerm = ALL_PAWNS;

      // Deletes the static pawns from the set of
      // all pawns to get the pawns to permutate
      for (Integer staticLeadPawn : e) {
        pawnsToPerm = removeElement(pawnsToPerm,
            Arrays.asList(pawnsToPerm)
                .indexOf(staticLeadPawn));
      }

      // Calculate the solutions
      solutionsFound = game30.findSolutions(e,
                                pawnsToPerm,
                                numPermutations);

      return solutionsFound.iterator();
    };
  }

  public static Function<Tuple2<String,Long>,String>
      enumerateSolutions() {
    return e -> {
      StringBuilder solutionNumber =
                                new StringBuilder();
      solutionNumber.append("Solution ");
      solutionNumber.append(Long.valueOf(e._2 + 1));
      solutionNumber.append(":\n");
      solutionNumber.append(e._1);
      return solutionNumber.toString();
    };
  }
}
```

pieces to be permutated and compares them with the rules of the game. The found solutions are then returned in a new RDD called `solutionText`, which holds the solutions in the form of a series of strings. To make it easier to compare the output of the found solutions with the solution representation of the Hadoop implementation, a similar formatting must be applied. However, in contrast to the implementation with MapReduce, the sequential numbering of the solutions is more complicated with Spark's paradigms. In a preliminary step, the solutions must first be numbered using key-value pairs so that the numbering can then be applied to the solutions to be output. For this purpose, the transformation `zipWithIndex()` is used, which transforms values into the format `(K, V)`. The string of the solution is transferred into the key `K` and the index of the element in the RDD forms the new value `V`. With the following call to `map()`, the key-value pairs are transformed into simple strings. For this purpose, the method `enumerateSolutions()` must be called, which increases the value of the index by one and uses it as the solution number. The detailed procedure is shown in Listing 7.

Finally, the solutions found are stored as text files on the distributed file system. For this purpose, the action `saveAsTextFile()` is called on the newly created RDD `solutionsEnumerated`, which saves the file under the path passed as parameter. Since this method is the first called action, this is the point at which the actual computation starts.

In order to count the permutations calculated during the execution of the program, an accumulator is used. Accumulators are distributed variables similar to counter variables in Apache Hadoop. They have the specific feature that values can be added in a distributed fashion, while the result can only be read out and output by the driver program.

4 RELATED WORK

The solving of games with the help of computer systems has a long tradition. In case of Apache Hadoop, the standard library already contains quizzes as examples, which shows the suitability of the framework for such use cases. Similar to Game 30, the game

Pentomino uses a CPU-intensive solving algorithm, which after an adaptation of [22] also corresponds to a distributed combinatorial search.

But there are even more CPU focused tasks that can be done with Spark. The processing of matrices, which is explained in [8], is one of them. Another example is MLlib, an Apache Spark based framework for machine learning algorithms [19]. Last but not least the computation of neural networks can be distributed with Spark, which can be used for example for the real-time detection of DDoS attacks [13].

GPU programming offers another possibility to process program code in parallel [9]. In this case, the calculations are executed on graphics cards that are specialized in parallel processing of computing operations. However, the programming of such applications is not trivial using Java. With JCuda [15] and JOCL [16] there are two frameworks that support GPU code development, but the program logic to be executed by the GPU must be programmed in C. The Java program is only used to call the C program and to process the returned results. The resulting complexity leads to the fact that GPU programming with Java has been used less often in practice than its high degree of innovation would suggest.

Even though GPUs are very well suited for parallel processing of arithmetic operations, this does not make big-data frameworks and their programming patterns obsolete. One reason for this is that the program code is usually only executed on a single graphics card. If several GPUs are to be connected across computers, proven distribution paradigms such as MapReduce come back into focus [11]. Furthermore, a big-data framework can serve as a useful abstraction layer. Apache Hadoop and Spark, for example, make it possible to distribute work dynamically between CPUs and GPUs [14].

It's certainly not enough to use processing time with Game 30 as the only criterion for choosing a big-data framework. In [7, 24] further criteria of an appropriate selection process are explained. At this point, we also refer to comparisons and measurements that deal with the primary field of application of big-data frameworks: the processing of large amounts of data [23].

For the puzzle Game 30 examined in this paper, alternative solutions are also feasible. The optimization of algorithms for the calculation of permutations alone is a continuously investigated topic [21].

5 MEASUREMENTS AND EVALUATION

In a previous comparison [18] of two Game 30 implementations with Multi-Threading and Apache Hadoop, the systems from Amazon Web Services (AWS) [1] were used so that the performance of the solutions can be easily reproduced. This allows us to run the Spark application on the same hardware configuration, what makes a comparison between the implementations possible.

5.1 Test environment

The systems used are called EC2 instances and are available in different hardware and software versions [1]. An EC2 instance of the type c3.4xlarge with 16 CPU threads of the Intel Xeon E5-2680 v2 processor and 30 GB RAM was used for the performance measurement of the multi-threading solution. The Hadoop or Spark configuration of the EC2 instances is called Amazon Elastic MapReduce (short: EMR) [2] and is available in different types of fixed memory. Since the application developed interacts very little with the data layer, we decided against using HDFS. Instead, we used the directly available object storage S3 [3], which is compatible to HDFS, to store all input and output files of the Hadoop application. The following EC2 instances were used for the Hadoop tests:

- m3.xlarge: 4 CPU threads of the processor Intel Xeon E5-2670 v2 and 15 GB RAM
- c3.2xlarge: 8 CPU threads of the processor Intel Xeon E5-2680 v2 and 15 GB RAM
- c3.4xlarge: 16 CPU threads of the processor Intel Xeon E5-2680 v2 and 30 GB RAM

The same configuration was now used for the experiment with Apache Spark.

5.2 Results

To evaluate the performance and scalability of the Hadoop solution, several test cases with different cluster configurations were ran. These tests were now reproduced for comparison with our Spark implementation. The test results are shown in Table 1. It should be noted that EMR divides the worker nodes into the groups core and task nodes in addition to managing nodes, the so-called master nodes. The core nodes represent a part of the distributed data management HDFS and execute MapReduce tasks, while task nodes only perform the latter. Since we do not use HDFS in our application, it would have made sense to use only task nodes. However, this is not possible due to the fact that EMR requires a core node.

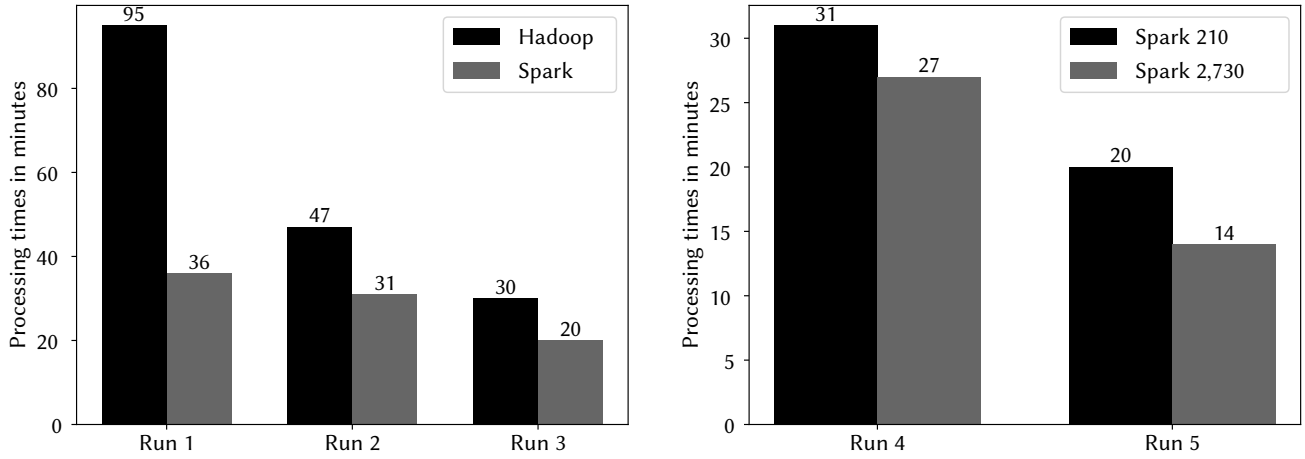
Studying Table 1, this raises the question why the number of map slots or worker cores between the test runs was not increased by exactly 100 %. With regard to the Spark implementation, this behavior can be explained by the execution location of the driver program that holds the `SparkContext`. Other than expected, the driver code and other administrative processes are not executed in the master node, but in one of the worker nodes. As a result, a Spark-Executor including its cores is no longer available during the execution of parallel processing. Although the unavailable resources can be minimized by changing the application configuration, our tests showed that such changes have only a minor positive effect on processing times and were therefore not pursued further. The unexpected scaling behavior of the Hadoop map slots is not due to configuration specifications, so the existence of an additional, EMR-internal configuration can be considered the cause. Without further internal information from Amazon it is not possible to find out how exactly the number of map or reduce slots is calculated. However, our tests showed that only nodes of the group core nodes exhibit the incomprehensible scaling behavior. In contrast, EC2 instances that belong to the task nodes group behave as specified in the configuration files.

A comparison of the execution times of the implementations with Apache Hadoop and Apache Spark in Table 1 and Fig. 4a shows that the Spark variant had a runtime that was about one third faster in all test cases. This indicates a lower overhead of Spark compared to Hadoop.

In terms of scalability, however, both implementations share the same difficulties, which can be explained by the ratio of the number

Table 1: Processing times with 210 and 2,730 input splits

	Run 1		Run 2		Run 3		Run 4		Run 5	
Cluster	1 x m3.xlarge (Master) 1 x c3.4xlarge (Core) 3 x c3.2xlarge (Task)		1 x m3.xlarge (Master) 2 x c3.4xlarge (Core) 6 x c3.2xlarge (Task)		1 x m3.xlarge (Master) 4 x c3.4xlarge (Core) 12 x c3.2xlarge (Task)		1 x m3.xlarge (Master) 2 x c3.4xlarge (Core) 6 x c3.2xlarge (Task)		1 x m3.xlarge (Master) 4 x c3.4xlarge (Core) 12 x c3.2xlarge (Task)	
System	Hadoop	Spark	Hadoop	Spark	Hadoop	Spark	Spark 210	Spark 2,730	Spark 210	Spark 2,730
Map slots	38	36	78	76	158	156	76		156	
Runtime	95 min	36 min	47 min	31 min	30 min	20 min	31 min	27 min	20 min	14 min
Incr. Map slots	-	-	105.26 %	111.11 %	102.56 %	105.26 %	-		105.26 %	
Decr. Runtime	-	-	-50.53 %	-44.46 %	-36.17 %	-36.48 %	-	-	-35.48 %	-48.15 %



(a) Hadoop and Spark

(b) Spark 210 and Spark 2,730

Figure 4: Processing times of the scenarios Run 1 to Run 5

of input splits to the number of map slots or worker cores [18]. However, since Apache Spark has a smaller overhead, the number of input splits or tasks to be processed can be increased to 2,730. This allows the workload to be distributed more evenly among the working nodes, resulting in a better scalability with Spark.

Table 1 and Fig. 4b show the results for Spark with 210 (run 4) and 2,730 (run 5) input splits. If we compare the execution times, we can see that the Spark implementation benefits from the increase of input splits. This effect can be seen in a reduction of the processing time in test run 4 from 31 min to 27 min, which corresponds to a reduction in runtime of about 13 %. In the case of test run 5, the time saving is even about 30 %. Even more significant than these values, however, is the insight how the processing time changes in relation to the number of worker cores. If we compare the execution times of the test runs 4 and 5 with 210 input splits, it can be seen that by doubling the hardware, the computing time could only be reduced by 35.48 %. If the tasks were divided into 2,730 input splits, the reduction was 48.15 %, which confirms a more efficient and hardware-independent scalability.

Unlike the implementation with Spark, the implementation with Hadoop could not take advantage of the finer load balancing because the task creation was too time-consuming in terms of runtime. For example, when executing test run 2 with 2,730 input splits, 52 minutes instead of 47 minutes were required. Although there are cases where the Hadoop variant can also benefit from the increase in input splits, but only if there is a very unfavorable ratio between the number of input splits and the number of available map slots.

When comparing the processing times of big-data solutions to the execution speed of a multi-threaded implementation (see Table 2), it is noticeable that Spark requires fewer CPU thread minutes than the multi-threaded implementation, although the multi-threaded implementation generates less overhead. The reason for this behavior could not be determined yet, but a plausible explanation could be the better load balancing with Spark. Specifically, this means that the CPU threads offered by AWS compete for hardware resources. The Amazon service Elastic MapReduce seems to handle this situation more effectively than the multi-threaded implementation.

Table 2: CPU thread minutes of the fastest multi-threaded, Hadoop and Spark runs

	Multi-threaded	Hadoop	Spark
CPU threads	15	78	76
Processing time	179 min	47 min	27 min
CPU thread minutes	2,685	3,666	2,052

6 CONCLUSION AND FUTURE WORK

The goal of this paper was to evaluate the parallelization of computationally intensive tasks using two big-data frameworks. For this purpose, a calculation game was selected as an illustrative case study. Furthermore, MapReduce and Spark algorithms were developed to compute all solutions of this computational game using the brute-force method. To perform the computationally intensive tests, virtual computers and virtual Hadoop or Spark clusters where rented from the cloud provider AWS.

The comparison of the two implementations with Hadoop and Spark showed that Spark was able to reduce the processing time by at least one third in all test cases with the same hardware, which indicates a lower overhead at Spark, since the same solving algorithm were used. Both implementations were found to be almost linearly scalable in terms of processing time under ideal conditions, although they are subject to the same difficulties if there is an unfavorable ratio of the number of input splits to the number of map slots or worker cores [18]. However, since Apache Spark has a smaller overhead, the number of input splits or tasks to be processed could be increased. This allowed the workload to be distributed more evenly among the work nodes, which in turn helped improve scalability. Even though Apache Spark is designed for faster in-memory processing of large amounts of data, we were able to show with this work that also pure computation algorithms can benefit from a migration from Hadoop to Spark. Such a migration would offer advantages without major drawbacks. At the same time, however, the gain in performance is in a range that encourages a change, but does not make it strictly necessary.

Compared to a simple multi-threaded implementation in Java, the implementation via Apache Spark was even the more powerful solution from the perspective of the utilization of the resources of the EC2 instances. With Apache Hadoop this behavior could not be achieved.

The transformation of the multi-threaded implementation to an Apache Hadoop or Spark application shows that the existing program logic does not necessarily have to be touched in such cases, since it is possible to call the existing methods from the tasks. The distribution is more difficult if the input parameters of the program logic cannot be broken down into several pieces. With regard to processing times, it is recommended in all cases to avoid communication between tasks and, in the case of Hadoop, the writing of extensive intermediate results. Even if these recommendations can be followed, the complexity of the calculation to be solved is still a criterion for the usefulness of distributed CPU-intensive processing, since the overhead of the frameworks can dominate the processing time for too trivial tasks.

The presented work offers many possibilities in terms of ongoing research and further experiments. For example, the handling of error cases could be further optimized in such a way that our application stores intermediate results to minimize the effort of recalculations in case of failures. Moreover, Amazon's cloud environment, with the support of Apache Flink, offers another big-data framework that could be studied by solving computational games with regard to its applicability for computationally intensive tasks.

However, even after a series of measurements we were not able to fully understand the scaling behavior of Amazons Hadoop implementation. For example, when the hardware of the cluster is doubled, the available Hadoop map slots are more than doubled, which can not be explained by entries in the Hadoop configuration files. Looking at the Spark implementation, the additional question arises why the capacity of the master node is not being used, since both the driver code and the managing processes are running on a working node. Another question is how to design a multi-threading algorithm in Java to make the most efficient use of (virtual) CPU-intensive threads so that its performance exceeds the performance of a similar algorithm running with Spark.

The results of our experiments are interesting both from the point of view of academic research as well as from the perspective of practical application. Therefore, the Java source code of the implementations presented in this work is provided to the public [12].

REFERENCES

[1] Amazon. 2020. Amazon EC2. https://aws.amazon.com/de/ec2/
[2] Amazon. 2020. Amazon EMR. https://aws.amazon.com/de/emr/
[3] Amazon. 2020. Amazon S3. https://aws.amazon.com/de/s3/
[4] Amazon. 2020. Amazon Web Services. https://aws.amazon.com/de/
[5] Anonym. 2009. Next permutation: When C++ gets it right. http://wordaligned.org/articles/next-permutation
[6] Apache. 2020. Apache Hadoop. http://hadoop.apache.org/
[7] Sandeep Bhargava, Drdinesh Goyal, and Bright Keswani. 2019. Performance Comparison of Big Data Analytics Platforms. *International Journal of Engineering, Applied and Management Sciences Paradigms (IJEAM)* Volume 54 (2019), 342–348. Issue Issue 2.
[8] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan Sparks, Aaron Staple, and Matei Zaharia. 2016. Matrix Computations and Optimization in Apache Spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016-08-13). ACM, New York, NY, USA, 31–38. https://doi.org/10.1145/2939672.2939675
[9] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. 2013. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *J. Parallel and Distrib. Comput.* 73, 1 (2013), 4–13. https://doi.org/10.1016/j.jpdc.2012.04.003
[10] Jao Cerreia. 2020. Jao Cerreia. https://www.joaocorreia.de
[11] Yi Chen, Zhi Qiao, Hai Jiang, Kuan-Ching Li, and Won Woo Ro. 2013. MGMR: Multi-GPU Based MapReduce. In *Grid and Pervasive Computing*, James J. Park, Hamid R. Arabnia, Cheonshik Kim, Weisong Shi, and Joon-Min Gil (Eds.). Vol. 7861. Springer Berlin Heidelberg, Berlin, Heidelberg, 433–442. https://doi.org/10.1007/978-3-642-38027-3_46 Series Title: Lecture Notes in Computer Science.
[12] Alexander Döschl, Max-Emanuel Keller, and Peter Mandl. 2020. Permutation-games. https://github.com/CCWI/permutation-games
[13] Chang-Jung Hsieh and Ting-Yuan Chan. 2016. Detection DDoS attacks based on neural-network using Apache Spark. In *2016 International Conference on Applied System Innovation (ICASI)* (2016-05). IEEE, New Your City, NY, 1–4. https://doi.org/10.1109/ICASI.2016.7539833
[14] K. R. Jayaram, Anshul Gandhi, Hongyi Xin, and Shu Tao. 2019. Adaptively Accelerating Map-Reduce/Spark with GPUs: A Case Study. In *2019 IEEE International Conference on Autonomic Computing (ICAC)* (2019-06). IEEE, New Your City, NY, 105–114. https://doi.org/10.1109/ICAC.2019.00022
[15] JCuda. 2020. JCuda. http://www.jcuda.org/
[16] JOCL. 2020. JOCL. http://www.jocl.org/

[17] Max-Emanuel Keller, Peter Mandl, Alexander Döschl, Daniel Kailer, and Markus Grimm. 2017. Verarbeitung komplexer XML-basierter Massendaten in BigData-Anwendungen. *AKWI* 2017, 6 (2017), 20–27. https://ojs-hslu.ch/ojs302/index.php/AKWI/article/view/93

[18] Peter Mandl and Alexander Döschl. 2017. Klassisches Multi-threading versus MapReduce zur Parallelisierung rechenintensiver Tasks in der Amazon Cloud. *HMD Praxis der Wirtschaftsinformatik* 55, 2 (2017), 445–461. https://doi.org/10.1365/s40702-017-0360-z

[19] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (2016), 1235–1241. Publisher: JMLR.org.

[20] Hamid Mushtaq, Nauman Ahmed, and Zaid Al-Ars. 2017. Streaming Distributed DNA Sequence Alignment Using Apache Spark. In *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)* (2017-10). IEEE, New York City, NY, 188–193. https://doi.org/10.1109/BIBE.2017.00-57

[21] Eric Ouellet and Omar Saad. 2018. Permutations: Fast implementations and a new indexing algorithm allowing multithreading. https://www.codeproject.com/Articles/1250925/Permutations-Fast-implementations-and-a-new-indexi

[22] Weiming Shi and Bo Hong. 2013. Clotho: an elastic MapReduce workload/runtime co-design. In *Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware - ARM '13*. ACM Press, New York, NY, USA, 1–6. https://doi.org/10.1145/2541583.2541588

[23] J. Veiga, R. R. Expósito, X. C. Pardo, G. L. Taboada, and J. Tourifio. 2016. Performance evaluation of big data frameworks for large-scale data analytics. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, New York City, NY, 424–431.

[24] Jorge Veiga, Roberto R. Expósito, and Juan Touriño. 2018. Performance Evaluation of Big Data Analysis. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Zomaya (Eds.). Springer International Publishing, Berlin, Heidelberg, 1–6. https://doi.org/10.1007/978-3-319-63962-8_143-1

[25] Ramon Wartala. 2012. *Hadoop: zuverlässige, verteilte und skalierbare Big-Data-Anwendungen*. Open Source Press, Munich, Germany.

[26] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia