An Efficient Implementation of the Algorithm by Lukáš et al. on Hadoop

Giuseppe Cattaneo[1], Umberto Ferraro Petrillo[2], Michele Nappi[1], Fabio Narducci[1], and Gianluca Roscigno[1(✉)]

[1] Dipartimento di Informatica, Università degli Studi di Salerno, 84084 Fisciano, SA, Italy
{cattaneo,mnappi,fnarducci,giroscigno}@unisa.it
[2] Dipartimento di Scienze Statistiche, Università di Roma "La Sapienza", 00185 Roma, Italy
umberto.ferraro@uniroma1.it

Abstract. Apache Hadoop offers the possibility of coding full-fledged distributed applications with very low programming efforts. However, the resulting implementations may suffer from some performance bottlenecks that nullify the potential of a distributed system. An engineering methodology based on the implementation of smart optimizations driven by a careful profiling activity may lead to a much better experimental performance as shown in this paper.

In particular, we take as a case study the algorithm by Lukáš *et al.* used to solve the Source Camera Identification problem (i.e., recognizing the camera used for acquiring a given digital image). A first implementation has been obtained, with little effort, using the default facilities available with Hadoop. A deep profiling allowed us to pinpoint some serious performance issues affecting the initial steps of the algorithm and related to a bad usage of the cluster resources. Optimizations were then developed and their effects were measured by accurate experimentation. The improved implementation is able to optimize the usage of the underlying cluster resources as well as of the Hadoop framework, thus resulting in a much better performance than the original naive implementation.

Keywords: Distributed computing · Hadoop · Source Camera Identification

1 Introduction

Current technologies provide decision-makers with the ability to collect a huge amount of data, i.e., Big Data, that requires the development of tools and methodologies with a high scalability degree. *Digital Image Forensics* area is one of the application fields where the problem of analyzing Big Data is arising [4,6,13,16]. Efficient solutions are usually available in the scientific literature. However, with the growth of digital photography, there is a need to assess how these solutions scale and/or how their performance can be optimized on distributed systems. Nowadays, paradigms and technologies, such as *MapReduce* [9] and Apache Hadoop [20], allow us to develop in a relatively simple way and without dealing with some of the most intricate aspects of distributed programming, such as inter-process data communication.

In this paper, which is a prosecution of the work has been presented in [7], we have engineered a Hadoop-based implementation of the Lukáš *et al.* algorithm [16] to solve the problem of *Source Camera Identification* (SCI). This consists in recognizing the camera used for acquiring a given digital image. The first implementation has been developed in a straightforward way with the help of the standard facilities available with Hadoop. The resulting distributed code exhibited shorter execution times than the original one when executed on a cluster of computers, although its performance was quite below expectations. A closer investigation revealed the existence of several performance issues due to the inability of this implementation to take full advantage of the underlying cluster resources. Therefore, we developed an engineering methodology aiming at pinpointing the causes behind the performance issues we observed and at solving them through the introduction of some theoretical and practical optimizations. The resulting implementations succeed in delivering a performance much better than the original distributed implementation.

The rest of the paper is organized as follows. Section 2 describes the *MapReduce* paradigm, with an emphasis on Apache Hadoop. Section 3 presents the case study, that is, the algorithm by Lukáš *et al.*. Section 4 presents the first attempt of porting the algorithm by Lukáš *et al.* on a Hadoop cluster and Sect. 5 shows the preliminary experimental results. In Sect. 6 we focus on some serious performance issues exhibited by our distributed implementation, and we propose and analyze further optimizations. Finally, in Sect. 7 we draw some conclusions and future directions of our work.

2 Apache Hadoop

In the recent years, several different architectural and technological solutions have been proposed for processing big amounts of data. An increasingly popular computing paradigm is *MapReduce* [9]. Within this paradigm, the computation takes a set of input $<key, value>$ pairs, and produces a set of output $<key, value>$ pairs. The user expresses the computation as two functions: *map* and *reduce*. The map function takes an input pair and produces a set of intermediate $<key, value>$ pairs. The *MapReduce* framework groups together all the intermediate *values* associated with a same intermediate *key* and passes them to the reduce function. The reduce function accepts an intermediate *key* and the set of values for that *key*. Then it merges together these values to form a possibly smaller set of values. Typically zero or one output pair is produced per reduce function. Map and reduce functions are executed, as tasks, on the different computers of a distributed system.

Differently from traditional paradigms, such as explicit parallel constructs based on message-passing, the *MapReduce* allows for *implicit parallelism*.

Namely, all the operations related to the exchange of data between the nodes involved in a computation are modelled according to a file-based approach and are transparently accomplished by the underlying middleware. Activities like data distribution, data replication, synchronization, scheduling, fault tolerance, redundant execution, data local computation, load balancing and efficient use of the network and disks, are in charge of the *MapReduce* framework. In this way, the programmer is focused only on defining the behavior of the map and reduce functions, and on deciding how data will feed the corresponding map and reduce steps. In general, no specific skills in parallel and distributed systems are required. The *MapReduce* paradigm has been first successfully adopted by Google for creating scalable, fault tolerance and massively-parallel programs that process large amounts of data using large commodity clusters. However, *MapReduce* has now gained a wider audience and it is used in several fields like satellite data processing, bioinformatics and machine learning (see, e.g., [2,8,10,15,17]).

Apache Hadoop [20] is currently the most popular framework supporting the *MapReduce* paradigm. It is a Java based open source grid computing environment useful for reliable, scalable and distributed computing. From an architectural viewpoint, Hadoop is mainly composed of a data processing framework plus the *Hadoop Distributed File System* (HDFS) [19]. The data processing framework organizes a computation as a sequence of user-defined *MapReduce* operations on datasets of $<key, value>$ pairs. These operations are executed as tasks on the nodes of a cluster. The HDFS is a distributed file system optimized to run on commodity hardware and able to provide fault tolerance through replication of data. Some Hadoop features used for source camera identification are available in our previous contribution [7].

3 The Case Study: Lukáš *et al.* Algorithm

The Source Camera Identification (SCI) problem concerns the identification of the digital camera used for capturing a given input digital image. A common identification strategy consists in analyzing the *noise* in a digital image to find clues about the digital sensor that originated it. *Pixel Non-Uniformity* noise (PNU) is a deterministic noise resulting from the different sensitivity of the pixel detectors to light. This difference is due to the inhomogeneity of the wafers of silicon and the imperfections derived from the manufacturing process of the sensor. Thanks to its deterministic and systematic nature, the PNU noise is the ideal candidate for providing a sort of fingerprint of digital cameras. Lukáš *et al.* in [16] were pioneers in demonstrating the feasibility of using the PNU noise for solving the SCI problem. The authors observed that PNU was successful in identifying the source camera used to take the considered picture, even distinguishing between cameras of the same brand and model.

The problem of performing Source Camera Identification on large datasets has not received much attention in the scientific literature. One of the few contributions in this area is presented in [13]. Here the authors tested over one million images spanning 6,896 individual cameras covering 150 models. Another relevant contribution is presented in [14] where the authors describe a fast searching algorithm based on the usage of a collection of *fingerprint digests*. They performed their experimentation on a database of $2,000$ iPhones, proving the feasibility of the approach proposed.

In our work we exploited the original version of the SCI algorithm by Lukáš *et al.* [16] applied to color images in the RGB space. Let I be the RGB image under scrutiny and $CamSet = \{C_1, C_2, \ldots, C_n\}$ the set of candidate origin cameras for I. The algorithm operates in four steps:

– **Step I: Calculating Reference Patterns.** It computes the *Reference Pattern* RP_C, that is, the sensor fingerprint, for each camera C belonging to $CamSet$. The approach proposed by Lukáš *et al.* consists in estimating RP_C by extracting the *Residual Noises* (RNs) from a set of pictures taken by using C and, then, combining these noises together, as an approximation of the PNU noise. The residual noise of an image I can be defined as $RN_I = I - F(I)$, where $F(I)$ is a filter function that returns the noise-free variant of I. The operation described above is applied pixel-by-pixel (for each color channel) and is iterated over a group of images with the same spatial resolution, here named *enrollment* images, taken by using C. This returns a group of residual noises, including both a random noise component and the PNU noise estimation of C. The sum of the residual noises is then averaged to obtain a tight approximation of the camera C fingerprint, i.e., RP_C.
– **Step II: Calculating Correlation Indices.** A set of calibration and testing images using each of the cameras belonging to $CamSet$ is introduced. The *Pearson's correlation* between the fingerprint of each camera C and the residual noise of each image T taken from the calibration/testing set is computed (the higher the values, the higher the probability that an image T has been taken by using a camera C). Cropping or resizing operations are performed in case the resolution of T does not match the resolution of the images used for determining RP_C.
– **Step III: Identification System Calibration.** The identification is based on the definition of a set of three acceptance thresholds (one for each color channel) to be associated to each of the cameras under scrutiny. If the correlation between the residual noise of I and the Reference Pattern of a camera C, on each color channel, exceeds the corresponding acceptance threshold, then C is assumed to be the camera that originated I. The thresholds are chosen so to minimize the *False Rejection Rate* (FRR) for calibration images taken by using C, given an upper bound on the *False Acceptance Rate* (FAR) for calibration images taken by using a camera different than C (Neyman-Pearson approach). The correlations of the testing images are then used to validate the identification system by comparing them to the acceptance thresholds.
– **Step IV: Performing Source Camera Identification.** It concerns the identification of the camera that captured I. Here the algorithm first extracts the residual noise from I, RN_I, then correlates it with the Reference Patterns of all the input cameras using the system calibrated in the third step. If the correlation exceeds the decision threshold of a certain camera, on each of the color channels, a match is found.

4 A Naive Implementation of the Lukáš *et al.* Algorithm on Hadoop

We developed in Java a *MapReduce* based implementation of the Lukáš *et al.* algorithm[1]. It was split in four different modules, each corresponding to the four processing steps of the Lukáš *et al.* algorithm. A preliminary image loading activity on the HDFS is also performed. This task was accomplished by copying and keeping the images as separate files.

In the following, we describe in details these four modules.

Step I: Calculating Reference Patterns. The aim of this step is to calculate the Reference Pattern of a camera C, by analyzing a set of enrollment images with the same spatial resolution and taken by using C. In the map phase, each processing node receives a set of images, extracts their corresponding residual noises and outputs them. In the reduce phase, the processing function (one for each camera C) uses the set of residual noises of C produced in the previous tasks and combines them, thus generating the RP_C. This operation is repeated for each input camera. During this step, the input *key* is derived from the image meta-data and *value* stores the URL of the image on HDFS. When a map function is activated, it receives this record, loads the corresponding image in memory from HDFS and, finally, extracts the RN from the image. As an output, this function produces a new $<key, value>$ pair, where *key* is the *camera id* and *value* is the URL of RN directly saved on HDFS. During the reduce phase, a function receives a tuple in the $<key, values>$ format, where *key* is the identifier of a camera, e.g., C, and *values* is a set of the URLs to RNs (saved on HDFS) for that camera, as calculated during the map tasks. All the RNs of the same camera are summed and then averaged to form the Reference Pattern for C. As an output, the function generates a new $<key, value>$ pair, where *key* is the identifier of C, and *value* is RP_C.

Step II: Calculating Correlation Indices. During this step, the algorithm extracts the RN of each calibration/testing image and correlates it with the RPs of all the input cameras. In the map phase, each processing node receives a list of input images to be correlated as $<key, value>$ records, where *key* is derived from the image meta-data and *value* stores the image URL on HDFS. For each URL, the corresponding image is (possibly) transferred to the slave node, and the RN is extracted and correlated with the RPs of all the input cameras calculated in the previous step. For each correlation, a map function generates a new pair, where *key* is the string *"Correlation"* and *value* consists of: the *image id*, the *camera id* used for shotting the image, the *RP id*, a value indicating the correlation preprocessing *type*, plus the three correlation indices (one for each color channel). Since each slave has to load the RP of all the input cameras, we used the Hadoop `DistributedCache` mechanism to make each node transfer to its local file system a copy of these files, before starting the Hadoop job. In this step, no reduce task is required.

[1] A copy of the source code of our implementation is available upon request.

Step III: Identification System Calibration. In this step a set of three acceptance thresholds (one for each color channel) is calculated for each of the input cameras. The thresholds are determined using the Neyman-Pearson approach and exploiting the correlation values of the calibration images computed in the previous step. The correlation values of a set of testing images, calculated during Step II, are then used to validate the identification system by comparing them to the aforementioned thresholds. Since this step is computationally cheap, it is run directly on the master node, without using any form of parallelization.

Step IV: Performing Source Camera Identification. The aim of this step is to establish which camera has been used for capturing an image I. The input of the Hadoop job is the directory where the RPs have been stored. The output is the id of the camera recognized as the originating camera for the input image. For each input RP, a new map function is invoked. This function uses a copy of I for extracting its residual noise and for calculating its correlation with the input RP. Then, the job returns a file containing the list of the correlation values needed to perform the recognition phase using the thresholds computed in the previous step. Finally, the predicted *camera id* is returned.

5 Experimental Analysis

In this section we discuss the results of a preliminary experimental analysis we have conducted. We compared the performance of our Hadoop-based implementation of the Lukáš *et al.* algorithm with its non-distributed counterpart. The discussion also includes a description of the experimental settings and the datasets used in our analysis.

5.1 Experimental Settings

All the experiments were conducted on a homogeneous cluster of 33 PCs equipped with 4 GB of RAM, an *Intel Celeron G530* dual-core processor, *Windows 7* host operating system and a 100 Mbps Ethernet card. In this environment, we installed on each computer a virtual machine running the *Ubuntu 12.10 64-bit* guest operating system, and equipped with 3,100 MB of RAM and 2 CPUs. Our cluster included 32 slave nodes and a master node, and the Hadoop version was 1.0.4. On each slave node, at most one map or reduce task was run. In addition, on each slave node, we set the properties that allow the framework to wait the end of all map tasks, before starting the reduce tasks, due to memory limits. According to our preliminary results, the HDFS replication factor was set to 2 and the HDFS block size was set to 64 MB.

The dataset used in our experiments is the same presented in [7]. It consists of 5160 JPEG images, shot using 20 different Nikon D90 digital cameras. This model has a CMOS image sensor and maximum image size of 4288×2848 pixels. 258 JPEG images were taken for each camera at the maximum resolution and with a very low JPEG compression. The images were organized in 130 enrollment images, 64 calibration images and 64 testing images for each camera. Enrollment images were taken from a *ISO Noise Chart 15739* [18], instead, calibration and testing images portray different types of scenes. The overall dataset is about 20 GB large, and about 40% of that size is due to enrollment images.

5.2 Preliminary Experimental Results

We developed Hadoop-based variants of the original Lukáš *et al.* algorithm. The first variant, here denoted `HSCI`, is the vanilla implementation of the algorithm described in Sect. 4. We focus on Step I and Step II only, because they are, by far, the most computationally expensive. In `HSCI` all the image files to be processed are initially loaded on HDFS. The files containing the *RN*s and the *RP*s obtained during the execution of the algorithm are also loaded on HDFS, as soon as they become available. As a consequence, map and reduce tasks take as an input (or provide as an output) a URL pointing at them.

We made a preliminary and coarse comparison between the performance of `HSCI` and the implementation running as a stand-alone (non-parallel) application, here named `SCI`, by measuring the overall execution time of the different steps of the algorithm in both settings. The results, available in Table 1, show that, when processing the second step of the algorithm, `HSCI` exhibits approximately a 16× speed up. On the contrary, the performance gain on the first step of the algorithm is almost negligible. Such a result is due to the reduce phase of this step. Each reduce task, in fact, has to collect from HDFS all the *RN*s generated during the map phase (in our experiments, 130 *RN*s for each *RP* file to generate, with the average size of a *RN* file of approximately 140 MB). This activity puts a heavy burden on the running time of the first step, as implemented by `HSCI`.

Table 1. Execution times, in minutes, of different preliminary distributed variants of the Lukáš *et al.* algorithm on a Hadoop cluster of 32 slave nodes, compared to the sequential counterpart, i.e., `SCI`, run on a single node.

Variant	Step I	Step II
`SCI`	888	5,257
`HSCI`	750	334
`HSCI_Seq`	290	304

In order to investigate the poor performance of `HSCI` during the first step of the Lukáš *et al.* algorithm, we analyzed the CPU and the network usage of slave nodes when running this step. The obtained results report that the CPU is mostly unused. Conversely, the network activity dominates both the map and reduce phases: the map phase, because of the time required to download from HDFS the input images and to write on HDFS the resulting *RN* files; the reduce phase, because of the time required to collect all the *RN* files produced in the map phase. A possible explanation for such long times is related to the problems of managing a very large number of small files [21].

A more efficient solution would be to fully exploit data local computation by further reducing the number of files to be processed and by placing the data on the slave nodes running the tasks in charge to process them. The solution we found consists in maintaining only two very large files containing all the image files. They have been coded as Hadoop `SequenceFile` objects and are: `EnrSeq`, used for storing a set of enrollment images, and `TTSeq`, used for storing a set of calibration and testing images. In both files, the images are ordered according to their originating *camera id*. Then, we used the input split capability available with sequence files for partitioning these two files among the different computing nodes, with the aim of promoting data local execution. Notice that the residual noises calculated during the first step of the algorithm are still written as separate files on HDFS as they become available, while the images are no longer directly downloaded from HDFS as individual files. In this case, in the sequence file input, the *key* of each pair is the meta-data of an image, while the *value* of that pair stores a binary copy of that image. The experimental performance of this implementation, labeled as `HSCI_Seq`, when running the first step of the Lukáš *et al.* algorithm, is much better than `HSCI` variant, with an execution time that is approximately 2.6× faster than `HSCI`. Also the second step of the algorithm seems to take advantage of this solution, as it is slightly faster than the `HSCI` solution.

6 Advanced Experimental Analysis

As already discussed in Sect. 5, a preliminary round of experimentations led us to develop a Hadoop-based variant of the Lukáš *et al.* algorithm, named `HSCI_Seq`, whose performance met enough our expectations. The same experiments revealed that the performance of this algorithm in a distributed setting is strongly influenced by the network activity required to load and/or to save files (RNs) on the underlying distributed file system.

In this section, we further analyze these phenomena. The results of a thorough profiling activity aimed at charactering the behavior of the `HSCI_Seq` implementation will be presented, in order to improve our understanding about the way an algorithm, such as the one by Lukáš *et al.*, performs when adapted to run on the Hadoop framework. We also assess the possibility of achieving further performance improvements.

6.1 Profiling `HSCI_Seq` Implementation

We recall from the previous section that the input dataset for our tests contains two sequence files: `EnrSeq` and `TTSeq`. During Step I, the processing of the `EnrSeq` file requires the creation of 130 map tasks, i.e., one for each HDFS block of input file, where the size of `EnrSeq` is about 8 GB and the HDFS block size is set to 64 MB. The average amount of data exchanged between map and reduce tasks is approximately 355 GB (without considering tasks and data replicas). In our experimental analysis of `HSCI_Seq`, the framework ran, in the average, 141 map tasks: 130 completed successfully and the remaining 11 killed by the framework. The existence of these additional tasks is due to the Hadoop speculative execution. In the reduce phase, we set the number of reduce tasks to 20, that is the number of *RP*s to be calculated. In our profiling experiment, 29 reduce tasks were launched by the Hadoop framework, with only 20 completing their execution and the other ones being killed by the framework. Our result shows that about 75% of the running time of `HSCI_Seq` during Step I is spent in the reduce phase. On one side, we suppose that this overhead is due to the time consumed by each reduce function to retrieve the corresponding *RN* files to sum from the HDFS, i.e., 130 *RN*s for each *RP* to be calculated. On the other side, we expect that this second phase would have lasted lesser as it performed simple computational operations such as multiple sums of matrices.

In order to clarify this behavior, we traced the start and the end execution time of each task, both map and reduce phase, in our experiment. In Fig. 1, we show an overview of the map and reduce tasks used by Hadoop when running the Step I of the `HSCI_Seq` algorithm. In some cases, the Hadoop framework may decide to issue a same task a second time (e.g., for recovering a task that has been assigned to a free slave node, without being completed). These cases are highlighted in the figure by coloring black the tasks that are killed when their twin tasks complete their executions. As it can be seen in the figure, the overall time spent by each slave node for processing map tasks is almost the same. In the reduce phase, we observe that some reduce tasks end as soon as they start or are killed immediately. That can be explained by the fact that these tasks have not been assigned a *RP* to be calculated. In fact, the overall number of slave nodes completing a reduce task and computing at least one *RP* is 12 against a total number of 20 *RP*s. This unbalanced assignment is due to the standard hash function used by the Hadoop partitioner service for the distribution of the keys (in our case, the id of the cameras) to be processed in the reduce tasks.

We further analyzed the behavior of tasks of Step I by profiling the CPU usage and the network activity. In Fig. 2 we report, for example, the CPU activity of *slave1*. During the first 60 min, spent on processing map tasks, a single core of the node was used almost at its maximum. Notice that, in our case, it is not possible to run two distinct map tasks on the same node because the amount of memory in it would not be enough. Instead, the second significant activity, i.e., that related to the execution of a reduce task covering about 65 min, featured a 10% average CPU usage. This seems to confirm that, during the reduce phase, the CPU of the involved slave nodes is nearly unused, as this phase is dominated by the network activity related to the retrieval from HDFS of the *RN* files to sum. This observation is also supported by the analysis of the incoming network throughput for *slave1* node during Step I, as illustrated in Fig. 3. The figure shows that there is an intense network activity for *slave1* along all the map phase and the reduce phase.

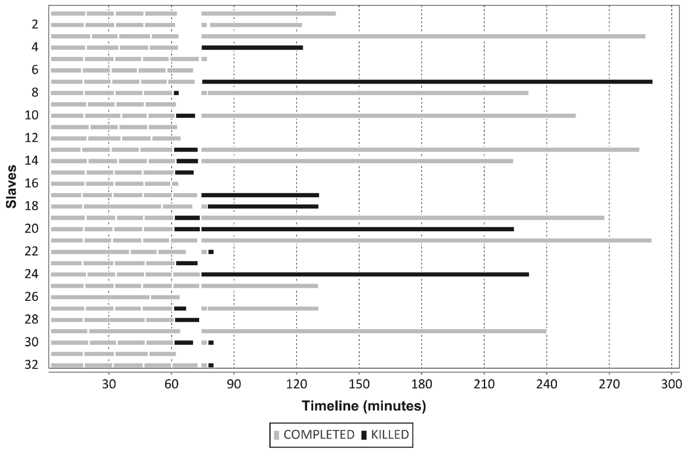

Fig. 1. `HSCI_Seq` implementation - An overview of map and reduce tasks launched during Step I. Reduce tasks start only after the termination of all map tasks.

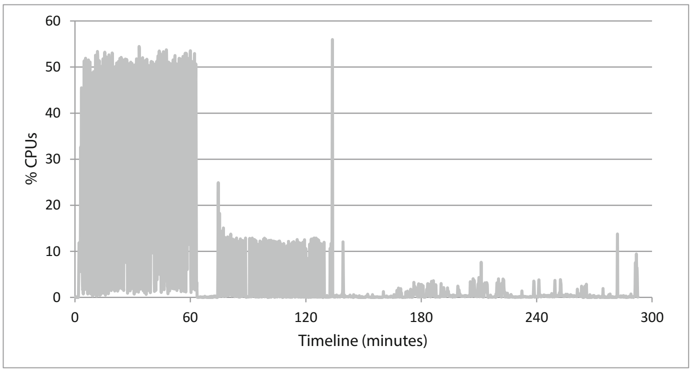

Fig. 2. CPU usage of *slave1*, in percentage, when running Step I of `HSCI_Seq`.

During Step II, at least 194 map tasks are created using the testing and calibration images available in the `TTSeq` file. In this experiment, the framework ran 210 map tasks: 194 completed successfully and the remaining 16 killed by the framework. Of all these tasks, 187 were data local map tasks. We recall that the Step II of the `HSCI_Seq` does not make use of the reduce phase, thus its execution time is approximately equal to the execution time of the map phase. An in-depth analysis of the map tasks revealed that they are characterized by an intense I/O activity, needed to load the Reference Patterns. However, these tasks also feature a very intense CPU activity, due to the work required to perform the correlations on big input files, as shown in Fig. 4 (the average CPU usage stays around 40%). That indicates, on one hand, that the CPU does not suffer much from delays due to I/O activity, and, on the other hand, that there is a margin

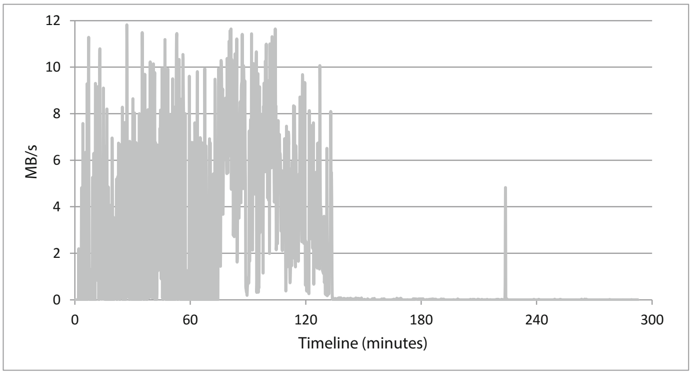

Fig. 3. Incoming network throughput of *slave1*, in MB/s, when running Step I of `HSCI_Seq`.

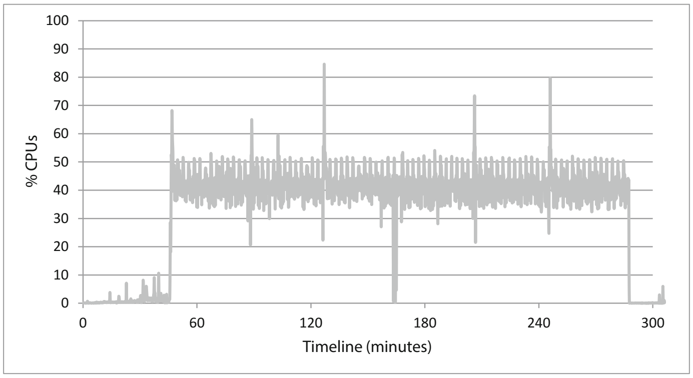

Fig. 4. CPU usage of *slave1*, in percentage, when running Step II of `HSCI_Seq`.

for optimization by taking advantage of the second core of the CPU, actually unused. As already stated above, in fact, the available memory in each node is likely to be insufficient to run two tasks at the same time. In addition, Fig. 4 shows that the CPU of *slave1* remains unused while waiting for the framework to copy the *RP*s from HDFS to the local file system.

6.2 Further Optimizations and Results

Following the profiling activity we pinpointed two issues affecting the performance of `HSCI_Seq` and we developed some practical optimizations to solve them.

Excessive network traffic. The excessive network traffic arising in Step I is mostly due to the transfer of a large number of *RN*s from map tasks to reduce tasks. Consequently, we required each map task to aggregate all the *RN*s generated for a same camera into one *RN* file, before sending it to the corresponding reduce task. The aggregation is done by summing all the *RN* files produced by a same node for a same camera during a map task. To facilitate this operation, the enrollment images are ordered by the *camera id* and the partial sum of the *RN* files is kept in memory by the node, without involving any I/O operation. Instead of using the standard Hadoop Combiner, we implemented an ad-hoc solution that does not require to store all the *RN* files in memory, but just their sum (this solution is denoted *in-map aggregation*). In addition, we used Hadoop implicit mechanism for directly passing this sum as *value* to the pair output by the node, rather than saving it on HDFS. We named `HSCI_Sum` the variant of `HSCI_Seq` featuring this optimization.

Poor CPU usage. The map phase during Step II is characterized by an intense CPU activity, but it is not able to take advantage of the availability of an additional CPU core. The standard behavior of the map task during Step II requires the loading from the local file system of a camera *RP*, followed by the calculation of its correlation with an input *RN*. While carrying out the first activity, the CPU is almost unused, as it is essentially an I/O-intensive operation. The second activity, instead, is CPU-intensive and makes no use of the file system. A possible intra-parallelization of this task, allowing for the usage of a second CPU core, consists in modelling the loading and the correlation activities on the producer-consumer paradigm, then to be implemented as a multi-threaded application. A first thread would be in charge of loading *RP* files from the local file system and adding them to an in-memory shared queue. In the meanwhile, the second thread would load *RP* files from the shared queue and would use them to calculate the correlation with an input *RN*. Notice that it is not possible to maintain in memory the *RP* of all the cameras because of their large size. The implementation of this strategy, here denoted `HSCI_PC`, also includes the optimizations introduced by `HSCI_Sum`.

Table 2. Execution times, in minutes, of the different steps of the variants of the Lukáš *et al.* algorithm on a Hadoop cluster of 32 slave nodes. For a comparison, see Table 1.

Variant	Step I	Step II
`HSCI_Seq`	290	304
`HSCI_Sum`	49	276
`HSCI_PC`	50	236

After developing the optimizations presented above, we performed another round of experiments in order to compare the optimized implementations to `HSCI`. The results, available in Table 2, report a significant performance improvement on `HSCI_Seq`. The first optimized code we consider is `HSCI_Sum`. This algorithm differs from `HSCI_Seq` in the way *RN*s are transmitted from map tasks to reduce tasks. Namely, it implements an aggregation strategy that drastically reduces the amounts of data exchanged between map and reduce tasks.

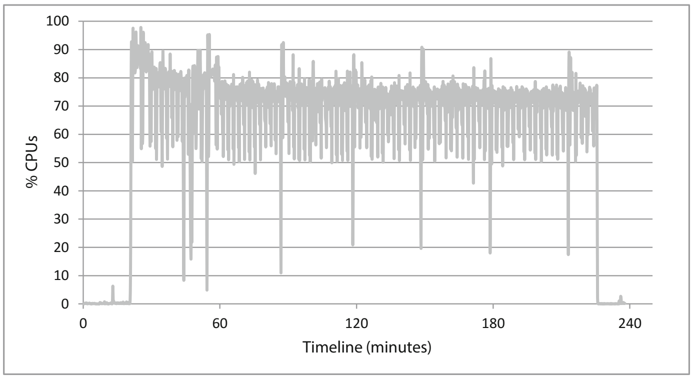

Fig. 5. CPU usage of *slave1*, in percentage, when running Step II of `HSCI_PC`.

For instance, in our experiments, the amount of data exchanged during Step I by `HSCI_Sum` is about 6% of that exchanged by `HSCI_Seq` in the same phase. This led to a consistent performance improvement in our experiments, since the Step I of `HSCI_Sum` required 49 min, in the average, to be accomplished against the 290 min required for the same step by `HSCI_Seq`. It is interesting to note that smaller amounts of data to exchange not only imply faster communications but could also result in a much smaller number of tasks being replicated and re-run by the Hadoop framework, thanks to shorter network congestions. In addition, the reduce phase takes just a few minutes.

`HSCI_PC` implementation uses the producer-consumer paradigm to evaluate correlations during the map phase of Step II, by means of a multi-threaded architecture. This approach brought a consistent performance gain compared to `HSCI_Seq` and `HSCI_Sum`, as the overall execution time of Step II dropped from 304 (`HSCI_Seq`) and 276 (`HSCI_Sum`) to 236 min (`HSCI_PC`). The result also includes a consistent increasing in the CPU usage, exhibited by `HSCI_PC` when processing the map phase of Step II and shown in Fig. 5 (for a comparison see Fig. 4).

7 Conclusion

In this paper, we discussed the engineering of an efficient Hadoop-based implementation of the Lukáš *et al.* algorithm, in order to solving the Source Camera Identification problem. We were able to quickly obtain a running distributed implementation for this algorithm, by leveraging the standard facilities available with the Hadoop framework. The vanilla distributed implementation exhibited a very poor performance. This motivated us to perform a thorough profiling activity which led, first, to pinpoint several performance issues and, then, to develop several both theoretical and practical optimizations, thus achieving a much better performance than the vanilla distributed implementation. In addition, other optimizations should be considered, for example, developing a custom partitioner in Step I to obtain a good balancing of the workload in reduce phase.

Software frameworks like Hadoop are attractive because they offer the possibility of coding full-fledged distributed applications with very low efforts. However, such an easiness of use implies a cost, as the resulting implementations may not be able to fully exploit the potential of a distributed system. In these cases, an engineering methodology based on the implementation of smart optimizations driven by a careful profiling activity may lead to a much better experimental performance, as demonstrated in this paper. To this end, we notice that the application pattern we considered in our paper, characterized by the interleaving of I/O-intensive and CPU-intensive tasks, is not only required by the Lukáš *et al.* algorithm but is an instance of a more general problem that is often found also in other application fields. As a consequence of this, the optimizations we developed in our case study are likely to improve in a systematic way the performance of Hadoop-based implementation of other algorithms as well. Along this line, an interesting future direction for our work would be the formalization of this methodology and its experimentation with other case studies, such as [1,3,5,11,12].

References

1. Bayram, S., Sencar, H.T., Memon, N., Avcibas, I.: Source camera identification based on CFA interpolation. In: IEEE International Conference on Image Processing (ICIP), vol. 3, pp. 69–72. IEEE (2005)
2. Cattaneo, G., Ferraro Petrillo, U., Giancarlo, R., Roscigno, G.: An effective extension of the applicability of alignment-free biological sequence comparison algorithms with Hadoop. J. Supercomput., 1–17 (2016). http://dx.doi.org/10.1007/s11227-016-1835-3
3. Cattaneo, G., Ferraro Petrillo, U., Roscigno, G., Fusco, C.: A PNU-based technique to detect forged regions in digital images. In: Battiato, S., Blanc-Talon, J., Gallo, G., Philips, W., Popescu, D., Scheunders, P. (eds.) ACIVS 2015. LNCS, vol. 9386, pp. 486–498. Springer, Cham (2015). doi:10.1007/978-3-319-25903-1_42
4. Cattaneo, G., Roscigno, G.: A possible pitfall in the experimental analysis of tampering detection algorithms. In: 17th International Conference on Network-Based Information Systems (NBiS), pp. 279–286, September 2014
5. Cattaneo, G., Roscigno, G., Bruno, A.: Using PNU-based techniques to detect alien frames in videos. In: Blanc-Talon, J., Distante, C., Philips, W., Popescu, D., Scheunders, P. (eds.) ACIVS 2016. LNCS, vol. 10016, pp. 735–746. Springer, Cham (2016). doi:10.1007/978-3-319-48680-2_64
6. Cattaneo, G., Roscigno, G., Ferraro Petrillo, U.: Experimental evaluation of an algorithm for the detection of tampered JPEG images. In: Linawati, M.M.S., Neuhold, E.J., Tjoa, A.M., You, I. (eds.) CT-EurAsia 2014. LNCS, vol. 8407, pp. 643–652. Springer, Heidelberg (2014). doi:10.1007/978-3-642-55032-4_66
7. Cattaneo, G., Roscigno, G., Ferraro Petrillo, U.: A scalable approach to source camera identification over Hadoop. In: IEEE 28th International Conference on Advanced Information Networking and Applications (AINA), pp. 366–373. IEEE (2014)

8. Choi, J., Choi, C., Ko, B., Choi, D., Kim, P.: Detecting web based DDoS attack using MapReduce operations in cloud computing environment. J. Internet Serv. Inf. Secur. (JISIS) **3**(3/4), 28–37 (2013)
9. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
10. Ferraro Petrillo, U., Roscigno, G., Cattaneo, G., Giancarlo, R.: FASTdoop: a versatile and efficient library for the input of FASTA and FASTQ files for MapReduce Hadoop bioinformatics applications. Bioinformatics (2017). https://dx.doi.org/10.1093/bioinformatics/btx010
11. Fridrich, J., Lukáš, J., Goljan, M.: Detecting digital image forgeries using sensor pattern noise. In: SPIE, Electronic Imaging, Security, Steganography, and Watermarking of Multimedia Contents VIII, vol. 6072, pp. 1–11 (2006)
12. Gloe, T.: Feature-based forensic camera model identification. In: Shi, Y.Q., Katzenbeisser, S. (eds.) Transactions on Data Hiding and Multimedia Security VIII. LNCS, vol. 7228, pp. 42–62. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31971-6_3
13. Goljan, M., Fridrich, J., Filler, T.: Large scale test of sensor fingerprint camera identification. In: IS&T/SPIE, Electronic Imaging, Security and Forensics of Multimedia Contents XI, vol. 7254, pp. 1–12. International Society for Optics and Photonics (2009)
14. Goljan, M., Fridrich, J., Filler, T.: Managing a large database of camera fingerprints. In: SPIE Conference on Media Forensics and Security, vol. 7541, pp. 1–12. International Society for Optics and Photonics (2010)
15. Golpayegani, N., Halem, M.: Cloud computing for satellite data processing on high end compute clusters. In: IEEE International Conference on Cloud Computing, pp. 88–92. IEEE (2009)
16. Lukáš, J., Fridrich, J., Goljan, M.: Digital camera identification from sensor pattern noise. IEEE Trans. Inf. Forensics Secur. **1**, 205–214 (2006)
17. McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M., et al.: The genome analysis toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. Genome Res. **20**(9), 1297–1303 (2010)
18. Precision Optical Imaging: ISO Noise Chart 15739 (2011). http://www.precisionopticalimaging.com/products/products.asp?type=15739
19. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. IEEE (2010)
20. The Apache Software Foundation: Apache Hadoop (2016). http://hadoop.apache.org/
21. White, T.: The small files problem. Cloudera (2009). http://www.cloudera.com/blog/2009/02/the-small-files-problem/