Minutiae-Based Fingerprint Matching Decomposition: Methodology for Big Data Frameworks

Daniel Peralta[a,b], Salvador García[c], Jose M. Benitez[c], Francisco Herrera[c]

[a] *Data Mining and Modelling for Biomedicine group, VIB Center for Inflammation Research, Ghent, Belgium*
[b] *Department of Internal Medicine, Ghent University, Ghent, Belgium*
[c] *Department of Computer Science and Artificial Intelligence. University of Granada, 18071 Granada, Spain*

Abstract

Fingerprint recognition, and in particular minutiae-based matching methods, are ever more deeply implanted into many companies and institutions. As the size of their identification databases grows, there is a need of flexible, reliable structures for fingerprint recognition systems. In this paper, we propose a generic decomposition methodology for minutiae-based matching algorithms that splits the calculation of the matching scores into lower level steps that can be carried out in parallel in a flexible manner. The decomposition allows to adapt any minutiae-based algorithm to frameworks such as MapReduce or Apache Spark. General and specific guidelines to enhance the performance of the adapted matching algorithms are also described. The proposal is evaluated over two matching algorithms, two Big Data frameworks (Apache Hadoop and Apache Spark) and two large-scale fingerprint databases, with promising results concerning the identification time, in addition to the reliability, scalability, distribution and availability capabilities that are provided by such underlying frameworks.

Keywords: Biometrics, fingerprint recognition, fingerprint matching, Big Data, MapReduce, Apache Spark

1. Introduction

Automatic fingerprint recognition has been a hot research topic during the last two decades [17]. As specific hardware becomes more easily available, the number of institutions and companies that use recognition techniques steadily increases over the years, along with the number of people that are to be identified [26]. Fingerprints present various characteristics that make them a good feature for recognition purposes such as uniqueness, size and

Email addresses: daniel.peralta@irc.vib-ugent.be (Daniel Peralta), salvagl@decsai.ugr.es (Salvador García), J.M.Benitez@decsai.ugr.es (Jose M. Benitez), herrera@decsai.ugr.es (Francisco Herrera)

distinctiveness [29]. Despite the many features that can be derived from fingerprints for the identification, minutiae are the most widely used ones. Algorithms usually base their computations on a set of local structures extracted from the fingerprint minutiae [31].

The recognition problem can be tackled from two different points of view. Whilst verification aims to assess whether two images correspond to the same fingerprint or not [18], identification seeks for a given input fingerprint throughout a database of template fingerprints [19]. As many other image matching problems, verification presents a high complexity due to the deformations present in the images, missing parts, and the high dimensionality of the representation space [45, 46]. Furthermore, identification is intrinsically more complex than verification, as it involves multiple comparisons between identities. Several techniques have been proposed to overcome the difficulties of identification, such as classification [10] and indexing [39].

The complexity of the problem increases along with the size of the template database. The main bottleneck when dealing with very large numbers of individuals is the identification time, which grows linearly with the number of matchings [32]. Huge fingerprint databases pose a challenge for fingerprint identification.

High Performance Computing (HPC) has been widely used in many computationally intensive problems in order to accelerate the runtime by splitting the calculations among a set of computers and processors [36]. It has been applied to the fingerprint identification problem from several points of view. On the one hand, it allows a parallel search through the template database that reduces the identification time [32, 25, 4, 48]. On the other hand, HPC systems can be used to provide data redundancy and high availability [15, 28]. However, the solutions present in the literature tackle these two benefits separately, sacrificing one on behalf of the other. The approach presented in [4] is especially interesting, as it provides an outstanding throughput of 35 million matches per second with a thoroughly optimized binary version of Minutia-Cylinder Code [3] for GPUs. However, this centralized approach does not provide redundancy or robustness against failures. The presence of more computers increases the risk of failure during the identification. The access to the underlying data can also pose a problem when the hardware fails.

In the last few years, several frameworks for dealing with Big Data problems using large-scale HPC architectures have been proposed. They represent an intermediate layer between the basic parallelization structures (such as messages, processes or threads) and the user application, providing fault-tolerance, distributed file systems, parallel computation and high availability. Two of the most popular among those frameworks are Apache Hadoop [42] and Apache Spark [23]. The specialized literature already includes some proposals for applying these frameworks for the biometric identification problem, such as load balancing strategies [24], a parallelization of iris recognition [35] or proposals for enhancing security [47, 13].

In this paper, we tackle the scalable fingerprint identification problem in Big Data frameworks. To do so, we propose a generic, flexible decomposition methodology for minutiae-based fingerprint matching algorithms. This decomposition is directly applied to adapt such algorithms to the described Big Data frameworks, so as to take advantage from their fault-tolerance, reliability, scalability and parallel computing capabilities. The decomposition of the matching process allows for a better parallelism, as well as for discarding some subsets of

local structures that can be early detected as non-matching. Moreover, we describe various guidelines that can be applied to the matchers to maximize these benefits.

To evaluate the proposal, the decomposition methodology and the suggested enhancements are applied over two different matching algorithms (Jiang's algorithm [22] and Minutia-Cylinder Code [3]) for them to be executed on two frameworks: Apache Hadoop and Apache Spark. The resulting identification systems are applied over several large-scale fingerprint databases, proving the scalability and flexibility capacities of the proposed idea. The source code of all the implemented algorithms is publicly available [1].

This manuscript is organized as follows. Section 2 presents the background on fingerprint recognition. Section 3 details the proposed decomposition methodology, as well as several applied cases, which are experimentally tested in Section 4. Finally, Section 5 concludes the paper.

2. Background

This section explains the background behind the proposed fingerprint identification decomposition methodology. First, Section 2.1 introduces the fingerprint matching problem, explaining the problems encountered when these methods are applied over very large databases along with some of the solutions in the literature. Then, Section 2.2 describes some frameworks for processing large databases and Big Data.

2.1. Fingerprint verification and identification: minutiae-based matching

The patterns of valleys and ridges that form fingerprints offer a variety of features that can be used for discerning them [14]. Minutiae are the most used among these features, more concretely the endings and bifurcations of the fingerprint ridges [26]. This allows us to represent a fingerprint T_i as a set of minutiae M_{ik}^T, each of whom can be represented as a triplet $\{x, y, \theta\}$ containing its coordinates and direction.

From this point of view, the verification problem consists of comparing two sets of minutiae to determine if they represent the same fingerprint [18]. A matching function is therefore defined as $Q(T_i, I_j) = q_{ij}$, where q_{ij} is the matching score between a template fingerprint T_i and an input fingerprint I_j. Usually $q_{ij} \in [0, 1]$ or $q_{ij} \in \{0, 1\}$. Fingerprint matching methods are divided into two categories [31]:

- Global matching methods look for the best alignment of the two minutiae sets. They use the entire information of the fingerprints, which provides high distinctiveness, but they tend to be computationally expensive and sensitive to distortions of the fingerprint captures. Some of the most prominent global matching methods are published in [34, 18, 50, 20, 7].

- Local matching methods extract *local structures* from the minutiae sets. Then, these local structures are compared to determine their similarity. These methods are usually invariant to translation and rotation, and less sensitive to distortions. They are

[1] `https://github.com/dperaltac/bigdata-fingerprint`

also able to perform parts of the overall matching procedure separately using partial information. However, they lack a global view of the fingerprint information. Some examples of purely local methods are presented in [2, 37, 33].

Most of the recent proposals for fingerprint matching combine these two approaches into a first local matching procedure, followed by a global consolidation step from which they obtain the final matching score q_{ij} [31]. Let T_i be a template fingerprint, decomposed into a set of local structures L_i^T as shown in Eq. (1), where elements l_{ik}^T are local structures, and m_i^T is the number of local structures extracted from the fingerprint. An input fingerprint I_j is decomposed analogously. The overall matching procedure is depicted in Fig. 1.

$$\begin{aligned} L_i^T &= \{l_{ik}^T \quad | \quad k = 1, ..., m_i^T\} \\ L_j^I &= \{l_{jk}^I \quad | \quad k = 1, ..., m_j^I\} \end{aligned} \tag{1}$$

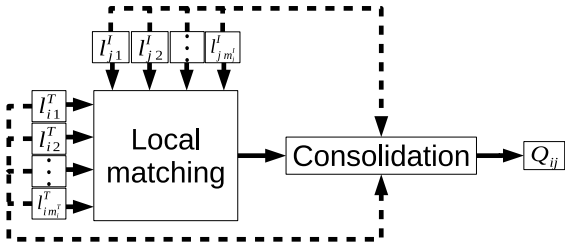


Figure 1: Workflow of a generic matching algorithm

The identification problem consists of searching for an input fingerprint I_j within a template database of n fingerprints $T = \{T_1, ..., T_n\}$ [21]. Most identification systems perform n verifications to compare I_j with every template fingerprint T_i, and return the identity that yields the maximum score:

$$\text{Identity} = \arg\max_i Q(I_j, T_i), \qquad i = 1, ..., n. \tag{2}$$

The identification time of such an identification system increases linearly with respect to n. Therefore, when n is very large (from tens of thousands onward) the identification time tends to be too long for most real applications.

2.2. Frameworks for Big Data

Big data has become one of the most challenging problems in information processing contexts [6]. The ever growing amount of potentially valuable data affects multiple research areas, such as decision making [38], data mining [43], biology [27] or physics [11]. In consequence, several frameworks have been designed over the last few years for dealing with Big Data problems over HPC systems. These frameworks usually provide the following main aspects simultaneously [8]:

- High availability, so that when a machine fails, the others can carry out its workload in a transparent way regarding the user.
- Distributed data storage, so that a disk failure in one machine is tolerable and does not involve any data loss. A side effect of this distribution is that data locality is increased, as several computers can process the same data locally.
- Massive parallelization, by splitting the input data into small chunks that can be loaded and processed separately by the machines. This also removes the necessity of storing all data into the memory at the same time, which is not possible when dealing with extremely large volumes of data.
- High data throughput, built upon the distributed data storage, which allows to load and write the data in parallel from all the machines.

Two of the most popular frameworks with these features are Apache Hadoop [42] and Apache Spark [23].

Apache Hadoop (along with its distributed file system HDFS) is an open-source implementation of the MapReduce paradigm, which is based on structuring the data into pairs of key and value. These pairs are processed into two phases: map and reduce. In particular, a typical MapReduce workflow involves three types of these pairs: the input to the map $\{k_1, v_1\}$, the pool between map and reduce $\{k_2, v_2\}$, and the output of the reduce $\{k_3, v_3\}$. In the map phase, each map operation receives a single pair, and outputs a set of pairs resulting from processing it. The intermediate pairs for all maps are shuffled and sorted, so that each reduce operation receives all the pairs with the same key. Therefore, there are as many reduce operations as there are different k_2 keys. Finally, the result of the reduce operations is also expressed as a set of key-value pairs, which are stored back to HDFS.

Apache Spark also uses HDFS as the underlying file system, but groups the data into Resilient Distributed Datasets (RDD). Spark focuses on keeping these RDD in memory as much as possible, so as to avoid the disk I/O overhead. As for the programming paradigm, Spark is not restricted to MapReduce. Instead, it provides a set of *transformations* (which convert an RDD into another RDD) and *actions* (which convert an RDD into some kind of result). It also provides the possibility to work with key-value pairs. These transformations and actions can be combined at will to create different workflows (one of which is MapReduce).

There are some works in the literature that use this kind of frameworks for biometric identification. Kitano and Su [24] present a load balancing strategy that distributes the template fingerprints evenly among the computing nodes for identification in a MapReduce-like environment, so that all worker nodes will have the same amount of work even if some of them are not operational. Their proposal loads the templates in memory prior the identifications, but does not modify or evaluate the matching algorithm itself. Shelly and Raghava [35] describe a MapReduce adaptation for iris recognition where each mapper compares the input sample with a subset of the template database and outputs the scores higher than a

threshold. Zhao et al. [49] propose a distributed and load-balancing framework for fingerprint identification, based on Hadoop and MongoDB. This proposal is two-fold: the feature extraction is carried out on the Hadoop Image Processing Interface, and the features are stored in a MongoDB cluster. Then, a global matching is carried out to search for an input fingerprint, which is efficient but not translation or rotation invariant. Peer and Bule [30] present some general guidelines for designing cloud-based fingerprint verification systems focusing on APIs, their operation methodology and their security. Other studies focus on the security of the biometric information, describing encryption methods that can be applied on the input fingerprint or biometric feature before it is transferred to the servers where the matching is actually performed [13, 9, 44].

This paper focuses on reaching a high-throughput exploration of very large databases, delving further into both the fingerprint matching process and the exploitation of existing Big Data frameworks. The main advantage of the proposal is its genericity: virtually any fingerprint matching algorithm can be decomposed following the proposed methodology, so as to benefit from the strengths of Big Data architectures such as Hadoop and Spark, including scalability, fault tolerance and reliable storage. The accuracy of the identification is also taken into account: very accurate matchers can be decomposed in this manner without any changes in the logic of the algorithm. Moreover, the proposal also enables the simultaneous search of multiple input fingerprints, increasing the parallelization level so that bulk searches can be carried out efficiently in very large template databases.

3. Minutiae-based fingerprint matching decomposition for Big Data frameworks

In this paper, we propose a generic decomposition methodology for adapting any minutiae-based fingerprint matching algorithm following the taxonomy in [31] to such Big Data frameworks. The resulting identification systems are expected to be fault-tolerant, reliable and scalable thanks to the underlying features of the Big Data frameworks.

For this purpose, the matching process is decomposed into smaller steps that can be computed in a parallel and flexible way. This decomposition also allows to discard some of the parts of the process in order to early detect non-matching fingerprints and accelerate the process.

This decomposition methodology is described in Section 3.1. Its application to MapReduce and Spark architectures is detailed in Sections 3.2 and 3.3, respectively. Then, this methodology is applied over two well-known matching algorithms: Minutia Cylinder-Code (Section 3.4) and Jiang's algorithm (Section 3.5). Finally, Section 3.6 describes some generic and specific enhancements that can be applied when decomposing these algorithms.

3.1. Matching decomposition

A classic matching process starts from two sets of local structures L_i^T and L_j^I, and compares them to compute a final score q_{ij}. In this proposal, we introduce the concept of *partial score*. A partial score contains the similarity information between two non-empty subsets of the original local structure sets: $L_{ik}^T \subseteq L_i^T$ and $L_{jk}^I \subseteq L_j^I$. The definition of the partial score function p is shown in Eq. (3), where $\mathcal{P}(S)$ is the set of all possible non-empty

subsets of a set S, and P is the space in which the partial scores are defined, which is dependent on the particular matching algorithm. Note that the cardinality of L_{ik}^T and L_{jk}^I is not necessarily the same: for instance, a partial score can be computed using a single template local structure ($L_{ik}^T = \{l_{ih}^T\}$ for some $h \in \{1, ..., m_i^T\}$) and all input local structures ($L_{jk}^I = L_j^I$).

$$\begin{aligned} p : \mathcal{P}(L_i^T) \times \mathcal{P}(L_j^I) &\to P \\ (L_{ik}^T, L_{jk}^I) &\mapsto p(L_{ik}^T, L_{jk}^I) \end{aligned} \tag{3}$$

Based on the concept of partial scores, we define two functions that aggregate them to compute the final matching score q_{ij} in a parallelizable and flexible manner:

- Function Q_p aggregates a set of k_p partial scores into a single new partial score, as shown in Eq. (4). Note that k_p is the number of partial scores that are being aggregated, which is not related to the number of local structures of either fingerprint. Therefore, the output of function Q_p is the partial score computed from k_p subsets of local structures obtained from each of the two fingerprints. These subsets are not necessarily disjoint, which allows for further flexibility in applying the decomposition scheme on a given matching algorithm.

$$\begin{aligned} Q_p : \mathcal{P}(P) &\to P \\ \left\{p(L_{ik}^T, L_{jk}^I) \mid k = 1, ..., k_p\right\} &\mapsto p\left(\bigcup_{k=1}^{k_p} L_{ik}^T, \bigcup_{k=1}^{k_p} L_{jk}^I\right) \end{aligned} \tag{4}$$

- Function Q_f, defined in Eq. (5), is applied on a single partial score—which contains the aggregated information of the similarity between all local structures of T_i and I_j—and computes the final matching score q_{ij}.

$$\begin{aligned} Q_f : P &\to \mathbb{R} \\ p(L_i^T, L_j^I) &\mapsto Q(T_i, I_j) = q_{ij} \end{aligned} \tag{5}$$

These two functions enable the fine-grain parallelization of the matching algorithm by allowing the aggregation of partial scores in a very flexible manner (by successively applying function Q_p to any combination of local structure subsets), while maintaining a fixed function Q_f to generate the final matching score. They also open the possibility of discarding partial scores that are not promising or do not contain relevant information for the matching, because Q_f is simply defined on the space of partial scores and requires no additional information.

As described so far, the decomposition proposed in this paper is composed by three main elements: partial scores, and two aggregation functions (Q_p and Q_f). Based on this

decomposition, the matching process becomes as depicted in Fig. 2. The process starts from the individual local structures, $l^T_{i1}, ..., l^T_{im^T_i}$ and $l^I_{j1}, ..., l^I_{jm^I_j}$, which are arbitrarily grouped into sets of the form $L^T_{ik} \subseteq L^T_i$ and $L^I_{jk} \subseteq L^I_j$. These sets are used to compute partial scores of the form $p(L^T_{ik}, L^I_{jk})$. The partial scores are aggregated by successive applications of function Q_p until a single partial score is obtained, which contains information of both entire local structure sets. Finally, function Q_f is applied on this partial score to obtain the final numeric score q_{ij}.

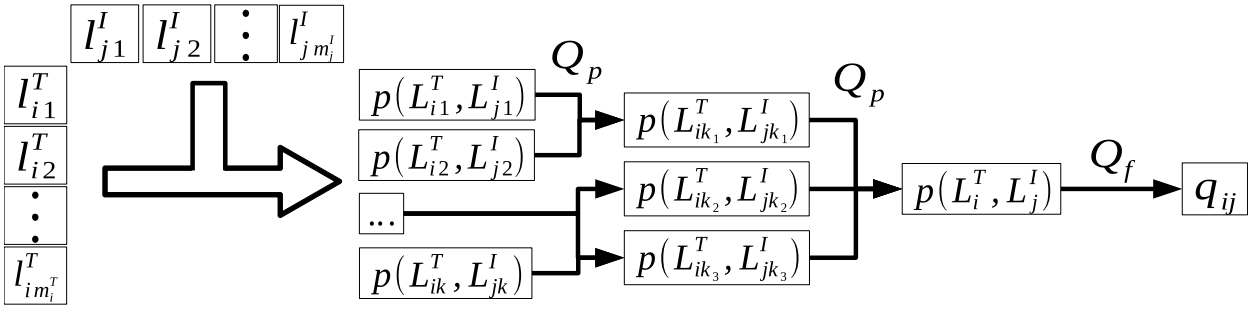


Figure 2: Workflow of the proposed matching decomposition

The adaptation of any minutiae-based matching algorithm to the proposed methodology only requires the design of the partial score structure and the aggregation functions Q_p and Q_f, which can often be directly derived from the original consolidation procedure.

A partial score can be safely discarded if it does not contain any useful information to compute q_{ij}. This is especially true in the case of algorithms that use simple consolidations, such as adding the highest similarities among local structures [31, 3].

3.2. Translation into MapReduce

The decomposition defined in the previous section can be used to formulate the problem in terms of the MapReduce paradigm. By embedding the decomposition into the key-value structure of MapReduce we can directly obtain all the benefits of such an architecture: fault-tolerance, distributed file system, massive parallelization, etc. Note that in this context MapReduce is just one case of use: the proposed decomposition methodology could be applied to adapt an algorithm to many other frameworks, such as Spark (Section 3.3).

The template database is composed by the local structures of all n_T template fingerprints. The local structures of some n_I input fingerprints that are to be identified are extracted (in general $n_I \ll n_T$). For the sake of simplicity, we consider that these local structures are stored in the distributed file system before the matching process in MapReduce starts.

The input of the map function is composed by the identifier of a fingerprint as the key and one of its local structures as the value. Then, for each input fingerprint I_j, the map computes the local matches between the local structure it received and all those in L^I_j, and aggregates them to produce a partial score:

$$
\begin{aligned}
&\text{Map}(k_1, v_1) \rightarrow list\left(\{k_2, v_2\},\ \forall j \in \{1, ..., n_i\}\right) \\
&\quad k_1 = i, \quad v_1 = l_{ik}^T \\
&\quad k_2 = \{i, j\}, \quad v_2 = p(\{l_{ik}^T\}, L_j^I)
\end{aligned}
\tag{6}
$$

Note that the map produces n_I output records for each input record. The partial scores produced thus are sent to the reduce phase, using the identifiers of the two fingerprints involved as key. Each reduce function merges all the partial scores for a given pair of template and input fingerprints to produce the final matching score as shown in Eq. (7). These scores are written to the distributed file system.

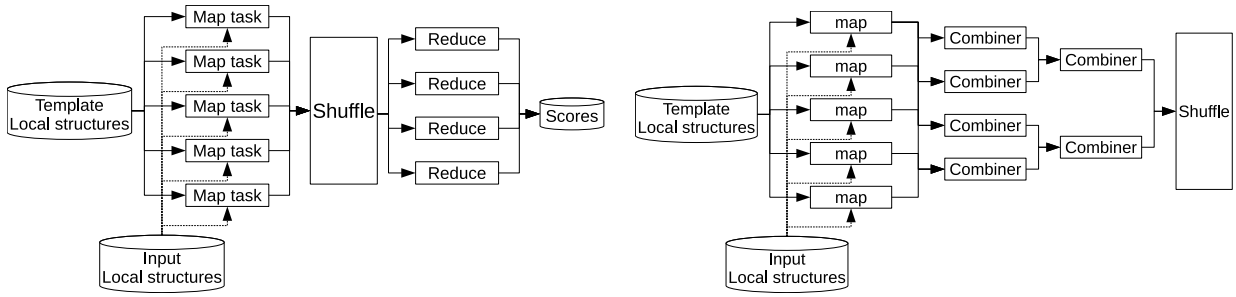
$$
\begin{aligned}
&\text{Reduce}(k_2, list(v_2)) \rightarrow \{k_2, v_3\} \\
&list(v_2) = \left\{ p(L_{ik}^T, L_j^I) \mid \bigcup_k L_{ik}^T = L_i^T \right\} \\
&\quad v_3 = Q_f(Q_p(list(v_2))) = Q(T_i, I_j)
\end{aligned}
\tag{7}
$$

Additionally, an intermediate combine phase aggregates sets of partial scores, so as to minimize the network and disk traffic between mappers and reducers and to maximize the parallelism. In MapReduce the combiner can be applied multiple times (or none) over the records, which implies it has to be associative and commutative [42]. The proposed decomposition complies naturally with these requirements thanks to the definition based on sets:

$$
\begin{aligned}
&\text{Combine}(k_2, list(v_2)) \rightarrow \{k_2, v_2'\} \\
&list(v_2) = \left\{ p(L_{ik}^T, L_j^I) \mid L_{ik}^T \subset L_i^T \right\} \\
&\quad v_2' = Q_p(list(v_2)) = p\left(\bigcup_k L_{ik}^T, L_j^I \right)
\end{aligned}
\tag{8}
$$

Fig. 3a depicts the overall workflow of our proposal within MapReduce. Fig. 3b focuses on what happens inside each map task. The local matching is performed within the maps whereas the consolidation step is split among the map, combine and reduce phases.

In addition to the optimization provided by the combiner, partial scores that are not of use for further aggregations can be discarded in the map or combine phases. This further decreases both the communication between the phases of MapReduce and the computational load of the reduce phase. The local structures of each input fingerprint are accessed from every map operation. Therefore, they should be accessible from the computing nodes in a fast manner. A broadcast or distributed cache support speeds up the entire matching process in such a case.



(a) Workflow of the complete process

(b) Workflow within each map

Figure 3: Workflows of the process in the MapReduce framework

3.3. Translation into Spark

The proposed decomposition methodology not only applies to MapReduce. It can also be used to design efficient an identification framework within Apache Spark [23], as described in Algorithm 1.

The first step of the identification within Spark is to broadcast the input local structures so that they are readily available in all the computing nodes. Then, the template local structures are grouped by their key, which allows all the local structures of the same template fingerprint to be located within the same node, to eliminate any further communication. For each template local structure l_{ik}^T and each input fingerprint I_j, the score $p(\{l_{ik}^T\}, L_j^I)$ is computed, forming an RDD of partial scores with keys i, j. The scores with the same key are aggregated one by one using the function Q_p defined in Eq. (4).

Once all the scores for the same key have been aggregated, the function Q_f is applied over each resulting partial score to obtain the set of final matching scores.

Algorithm 1 Pseudocode of the adaptation of the decomposition to Spark

```
InputRDD ← ReadRDD(inputs)
Broadcast(InputRDD)
TemplateRDD ← ReadRDD(templates)
TemplateRDD.GroupByKey()
for all l_ik^T ∈ TemplateRDD do
  for all L_j^I ∈ InputRDD do
    ps ← p({l_ik^T}, L_j^I)
    psRDD.insert({{i, j}, ps)
  end for
end for
FinalpsRDD ← psRDD.reduceByKey(Q_p)
ScoresRDD ← FinalpsRDD.mapValues(Q_f)
```

3.4. Use case: Minutia Cylinder-Code

In order to demonstrate a practical application of the proposed generic methodology, Section 3.4.1 describes the well-known MCC matching algorithm [3]. Then, Section 3.4.2

details how it can be decomposed into the terms of the proposed methodology.

3.4.1. The Minutia Cylinder-Code matching algorithm

MCC uses local structures called cylinders. A cylinder contains the information about a minutia neighborhood; therefore, a fingerprint contains as many cylinders as minutiae. Ultimately, the cylinder is encoded as a vector of real numbers of size $N_s N_s N_d$. Note that whilst the original authors of MCC suggest the use of binary vectors for more efficiency, we choose the real-coded version as it provides more accurate results.

The local matching consists of computing the similarity between each pair of cylinders (one of T_i and one of I_j), obtaining a matrix Γ, where γ is a similarity function:

$$\Gamma \in [0,1]^{m_i^T \times m_j^I} \quad \Gamma[r,s] = \gamma(l_{ir}^T, l_{js}^I) \tag{9}$$

Cappelli et al. [3] propose four consolidation techniques to compute the global score. In this paper we focus on two of them: Local Similarity Sort (LSS) and Local Similarity Sort with Relaxation (LSSR).

LSS consists of averaging the n_P best local similarities within Γ. The global score $Q_{LSS}(T_i, I_j)$ is computed as defined in Eq. (10), where Γ' is a vector containing the local similarities in Γ in decreasing order, $\lfloor \cdot \rceil$ is the rounding operator, and min_{n_P}, max_{n_P}, μ_P and τ_P are constants.

$$Q_{LSS}(T_i, I_j) = \frac{\sum_{h=1}^{n_P} \Gamma'_h}{n_P} \tag{10}$$

$$n_P = min_{n_P} + \lfloor (Z(n_R, \mu_P, \tau_P)) \cdot (max_{n_P} - min_{n_P}) \rceil \tag{11}$$

$$n_R = \min\{m_i^T, m_j^I\} \tag{12}$$

$$Z(v, \mu, \tau) = \frac{1}{1 + e^{-\tau(v-\mu)}} \tag{13}$$

This consolidation is very simple as it does not require of any information from the local structures besides their number (m_i^T and m_j^I).

LSSR is a more complex approach, involving n_{rel} relaxation steps. First, the most similar n_R pairs of cylinders are selected. Then, each relaxation step k computes a new similarity λ_t^k for each pair $t = (r_t, s_t)$ as shown in Eq. (14), where $w_R \in [0,1]$ is a weighting factor, ρ is a function measuring the compatibility between two pairs of local structures and $\lambda_t^0 = \Gamma[r_t, s_t]$.

$$\lambda_t^k = w_R \cdot \lambda_t^{k-1} + \frac{(1 - w_R)}{n_R - 1} \cdot \left(\sum_{\substack{s=1 \\ s \neq t}}^{n_R} \rho(t,s) \cdot \lambda_s^{k-1} \right) \tag{14}$$

Once the relaxed similarity values $\lambda_t^{n_{rel}}$ have been computed, the efficiency ϵ_t of the relaxation is defined:

$$\epsilon_t = \frac{\lambda_t^{n_{rel}}}{\lambda_t^0} \tag{15}$$

The global score is computed by averaging the relaxed similarities that have the n_P highest efficiencies.

3.4.2. MCC decomposition

The local structures used by MCC are the cylinders. Therefore, both the template database and the input fingerprints are represented by sets of cylinders.

The LSS consolidation consists of averaging the n_P best local similarities. In a simplistic approach, a partial score $p(L_{ik}^T, L_{jk}^I)$ would be a subset of Γ (denoted Γ_k) that only contains the rows and columns related to the local structures in L_{ik}^T and L_{ik}^I. However, as we are only interested in the n_P best similarities the formulation can be optimized as shown in Eq. (16), where Γ'_k is the set of n'_P best similarities within Γ_k and n'_P is an upper bound of n_P. The number of template local structures that are used to compute Γ'_k is needed in order to eventually compute the actual value of n_P within Q_f.

$$p_{\text{LSS}}(L_{ik}^T, L_j^I) = \{|L_{ik}^T|, \Gamma'_k\} \tag{16}$$

$$n'_P = min_{n_P} + \lfloor (Z(m_j^I, \mu_P, \tau_P)) \cdot (max_{n_P} - min_{n_P}) \rceil \tag{17}$$

The aggregation function Q_p for a set of partial scores consists of selecting the n'_P best similarities among all those included in the partial scores. The number of template local structures involved is obtained by adding the first term of all partial scores. Note that this requires the aggregated L_{ik}^T to be disjoint, which complies with the proposed MapReduce and Spark implementations. Finally, once a partial score that involves all the template local structures is obtained, the actual n_P value can be computed and the best n_P similarities can be averaged to produce the matching score.

$$Q_{f_{\text{LSSR}}}\left(\{m_i^T, \Gamma'_k\}\right) = \frac{\sum_{h=1}^{n_P} \Gamma'_{kh}}{n_P} \tag{18}$$

Note that similarities that are equal to zero do not have any influence on the final score. The partial scores that only contain zero-valued similarities can be safely removed to reduce the network overhead and simplify the process.

The decomposition approach for LSSR is similar. In this case not only the similarity values are needed to compute the matching score, but also the minutiae themselves. Additionally, n_R local similarities are needed instead of n_P ones (note that in general $n_P \ll n_R$). Therefore, the partial score is defined to contain a set of m_j^I local similarities (m_j^I is an upper bound of n_R) along with the involved minutiae (M_{ir}^T and M_{js}^I, for the template and input fingerprints respectively) and the value $|L_{ik}^T|$:

$$p_{\text{LSSR}}(L_{ik}^T, L_j^I) = \{|L_{ik}^T|, \{\{M_{ir}^T, M_{js}^I, \Gamma[r,s]\} \mid \forall \Gamma[r,s] \in \Gamma'_k\}\} \tag{19}$$

The aggregation function is analogous to that of LSS, with the exception that the minutiae are included into the partial score. Once the partial score $p_{\text{LSSR}}(L_i^T, L_j^I)$ is obtained,

the matching score can be computed from the n_R best local similarities and the associated minutiae as described in Section 3.4.1.

3.5. Use case: Jiang's algorithm

The proposed methodology has also been applied to the matching algorithm proposed by Jiang and Yau [22], which is described in Section 3.5.1. Section 3.5.2 explains the decomposition for this algorithm.

3.5.1. Jiang's matching algorithm

This algorithm uses local structures based on the N_n nearest neighbors of each minutia. Each local structure is described as a vector of real numbers, which can be compared by a function γ. The similarity matrix between all pairs of local structures follows the form of Eq. (9). In this work, we do not use the type and ridge count of the minutiae so as to only use the mandatory parts of the ISO 19794-2 standard [1]. The use of minutiae type can often be misleading in matching, and the accuracy loss caused by not using the ridge count is compensated by increasing the number of neighboring minutiae.

The global matching consists of fixing the best-matched pair of local structures, and aligning all the other minutiae according to this pairing. A rotation and translation invariant vector Fg_k is obtained from every minutia. Then, a matching certainty level $ml(r, s)$ is computed for every pair of aligned minutiae:

$$ml(r,s) = \begin{cases} 0.5 + 0.5\Gamma[r,s], & \text{if } |Fg_r^I - Fg_s^T| < bl \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

The final matching score is computed as follows:

$$Q_{\text{Jiang}}(T_i, I_j) = \frac{\sum_{r,s} ml(r,s)}{\max\{m_i^T, m_j^I\}} \tag{21}$$

To improve the accuracy, this procedure is repeated for the n_{BEST} best local matches considered for the alignment and the maximum score is returned, as suggested in [26].

3.5.2. Jiang's algorithm decomposition

The local structures used by the algorithm are real vectors. As the minutiae information itself is also needed for the global consolidation, the coordinates and angle of each minutia are also included within the corresponding local structure when stored into the system.

The partial score for this algorithm can be defined as:

$$p_{\text{Jiang}}(L_{ik}^T, L_j^I) = \{\{M_{ik}^T, ...\}, \Gamma_k\} \tag{22}$$

where Γ_k is the set of n_{BEST} maximum local similarities among those obtained by comparing the local structures in L_{ik}^T with those in L_j^I. Therefore, the aggregation function Q_p simply returns the union of the L_{ik}^T sets it receives, along with the n_{BEST} maximum local similarities in Γ_k.

The final aggregation function Q_f performs the n_{BEST} global matchings and returns the maximum matching score, as defined in Eq. (20) and Eq. (21).

3.6. Enhancing the performance of the matching decomposition

Once the decomposition scheme has been designed, it is possible to analyze the expected performance of the matching and to provide guidelines for its enhancements.

Table 1 shows the value of the different factors that determine the performance of the approach both for MapReduce and Spark, denoting the average number of local structures per fingerprint by $\overline{m}$. Note that in MapReduce, the number of combiners (and therefore the number of applications of Q_p) depends on many internal parameters and cannot be determined *a priori*. Nevertheless, the complexity of Q_p usually depends on the size of its parameters. If the function is applied few times each application would aggregate many partial scores, which can be slow, while a large number of applications would involve small sets of partial scores, each of which can be computed faster. The main bottleneck for the MapReduce approach would be the total size of the key-value pairs after the combiner, which are transmitted to the reducers via HDFS. For Spark the bottleneck mainly involves the number of operations that are performed.

Table 1: Factors that determine the performance of the approach for MapReduce and Spark. The specific parameters of Jiang's algorithm and MCC are extracted from their original publications.

Factor	**Value**
Bytes of Jiang local structure	$8 \cdot 3 \cdot N_n$
Bytes of Jiang local structure (enhanced)	$4 \cdot 2 \cdot N_n + N_n$
Bytes of cylinder	$8 \cdot N_s \cdot N_s \cdot N_d$
Bytes of cylinder (improved)	$4 \cdot N_s \cdot N_s \cdot N_d + 4$
Database key-value pairs	
Number of maps	$n_T \overline{m}$
Partial score creations	
Key-value pairs after map	$n_T n_I \overline{m}$
Key-value pairs after combiner	Between $n_T n_I \overline{m}$ and $n_T n_I$
Number of reduces	
Output key-value pairs	$n_T n_I$
Number of applications of Q_f	

The enhancements must focus on optimizing the removal of non-promising partial scores and on reducing the size of local structures and partial scores. Along these lines, our enhanced implementation used single precision floating point numbers and angles quantized to 256 values as in [4]. Further specific optimizations have been applied to the two matchers that constitute the cases of use for this paper. In particular, the resulting local structure size is shown in Table 1.

For MCC, the LSS consolidation can be further simplified by setting the value of n_P to a fixed value, eliminating the need for $|L_{ik}^T|$ in Eq. (16). The value n_R was also divided by two, to reduce the size of partial scores for LSSR. Furthermore, the norm of the feature vector (used to compute the similarity) was stored within the local structure.

For Jiang's algorithm, the size of the neighborhood was increased from 2 to 4, and a bounding box was applied when computing the local similarity, so that two local structures with very different angles or locations are never matched. From the implementation point of view, the feature vector was split in two, as now the angle differences can be efficiently encoded in a vector of bytes. Finally, the global consolidation step was substituted by a sum of the local similarities so that the partial score in Eq. (22) can become simply $\{\Gamma_k\}$, and ml is replaced by Γ in Eq. (21).

4. Experiments and analysis

This section presents the experimental results that verify the performance of the proposal. Sections 4.1 and 4.2 perform the analysis of the decomposition scheme for two different databases. Finally, Section 4.3 presents a study on the scalability of the proposal.

The two described matchers have been implemented both in Apache Hadoop (in Java) and Apache Spark (in Scala). The MPI-based parallel system[1] presented in [32], implemented in C++, is used as a reference to compare the results.

All the experiments described in this section were carried out on a cluster of 20 computing nodes (2 Intel Xeon E5-2600 2.00GHz, 64GB RAM each) plus a header node (2 Intel Xeon E5-2600 2.00GHz, 32GB RAM), connected by a QDR Infiniband network. Cloudera Hadoop 5.3.1 (Hadoop 2.5), Spark 1.5.2 and OpenMPI 1.8.7 were used for the performed experiments. All fingerprint minutiae were extracted using NIGOS *mindtct* [41], with the parameters presented in Table 2. We used the parameters provided in the respective original publications for both algorithms. Note that all accuracy results are computed for a 0% False Positive Rate (FPR).

Table 2: Parameters for the methods used in the experimentation

	Parameters	**Ref.**
Jiang	$w_d = 1, w_\theta = 0.3\frac{180}{\pi}, w_\phi = 0.3\frac{180}{\pi}$ $w_n = 0, w_t = 0, n_{\text{BEST}} = 5$ $N_n = 2, BG_1 = 8, BG_2 = \frac{\pi}{6}, BG_3 = \frac{\pi}{6}$ Enhanced version: $N_n = 4$, Local bounding box: $\{250, 250, 96\}$	[22]
MCC	$R = 70, N_s = 8, N_d = 6, \sigma_s = \frac{28}{3}, \sigma_d = \frac{2\pi}{9}$ $\mu_\Psi = 0.01, \tau_\Psi = 400, \omega = 50, min_{VC} = 0.75$ $min_M = 4, min_{ME} = 0.60, \sigma_\theta = \frac{\pi}{2}, max_{n_p} = 12$ Floating-point-based version: enabled, $\mu_P = 20$ $w_R = 0.5, \mu_1^\rho = 5, \tau_P = 0.6, min_{n_p} = 4, \tau_1^\rho = -1.6$ $\mu_2^\rho = \frac{\pi}{12}, \tau_2^\rho = -30, \mu_3^\rho = \frac{\pi}{12}, \tau_3^\rho = -30, n_{rel} = 5$ Enhanced version: $n_P = 10$	[3]
mindtct	output format = ANSI INCITS 378-2004 image enhancement = enabled	[41]

4.1. SFinGe large database

A database of 400 000 template fingerprints was generated by means of the SFinGe software [5, 26] in order to test the proposal over a large database. A set of 10 000 input fingerprints was built by taking one additional impression of 5000 of the template fingerprints,

[1] `https://github.com/dperaltac/mpi-afis`

plus an additional set of 5000 fingerprints with no match in the database. The parameters used to generate the database are listed in Table 3. Note that no other fingerprints were added to the database, so as to obtain a homogeneous set of fingerprints that will allow us to compute meaningful statistics, as shown in Table 4.

Table 3: Parameter specification to generate the SFinGe database

Scanner parameters	**Generation parameters**
Acquisition area: 14.6mm x 19.6mm.	Impressions per finger: 25.
Resolution: 500 dpi.	Class distribution: Natural.
Image size: 288 x 384.	Varying quality and perturbations.
Background type: Optical.	Generate pores: enabled.
Background noise: Default.	Save ISO templates: enabled.
Crop borders: 0 x 0.	Output file type: WSQ.

Table 4: Statistics of the SFinGe database

		Template	**Input**
Number of template fingerprints (n_T and n_I)		400 000	10 000
Average number of local structures ($\overline{m}$)		55.47	50.14
Size in bytes	**Jiang**	3.25E+09	2.38E+07
	Jiang (enhanced)	2.75E+09	1.76E+07
	Cylinders	6.41E+10	7.92E+08
	Cylinders (enhanced)	3.29E+10	3.81E+08

Even though the same algorithms were implemented in the three compared frameworks (MPI, Hadoop and Spark), the accuracy results differ slightly. This is caused by the floating point operations errors that may accumulate along the matching computations. Table 5 shows how the TPR (True Positive Rate) obtained for the SFinGe database is very similar for the three compared frameworks, even though not exactly equal. However, these differences can safely be considered as negligible in any practical context. The enhancements applied to the matchers do have effects on the accuracy: Jiang's algorithm suffers a loss of TPR in this case, while LSS improves its accuracy, thanks to the larger value used for n_P.

Table 5: TPR obtained with the SFinGe database (for 0% FPR)

	Jiang		**MCC (LSS)**		**MCC (LSSR)**	
	Orig.	Enhan.	Orig.	Enhan.	Orig.	Enhan.
MPI	0.710	–	0.669	–	0.949	–
Hadoop	0.711	0.507	0.681	0.751	0.952	0.951
Spark	0.712	0.507	0.677	0.766	0.950	0.948

Table 6 shows the average identification times for the three frameworks, and matchers, along with their enhancements. Fig. 4 displays the number of thousands of matches per second performed with each configuration. Hadoop turns out to be the slowest option, due to the amount of disk accesses performed between the map and reduce phases.

However, the times obtained with Spark are lower than those obtained with MPI, which is a very noticeable result. Even though the template database resides in memory for both frameworks, Scala is considered to be slower than C++ because of its use of the bytecode

Table 6: Average identification times in seconds with the SFinGe database

	Jiang		MCC (LSS)		MCC (LSSR)	
	Orig.	Enhan.	Orig.	Enhan.	Orig.	Enhan.
MPI	0.5003	–	2.5872	–	3.4591	–
Hadoop	3.7925	0.2005	7.4802	3.5902	14.9461	5.8687
Spark	0.3928	0.1661	2.0968	1.8003	3.5360	3.1624
Ratios						
Hadoop/MPI	7.5802	0.4008	2.8552	1.3877	4.2259	1.6966
Spark/MPI	0.7851	0.3320	0.8045	0.6958	0.9782	0.9142

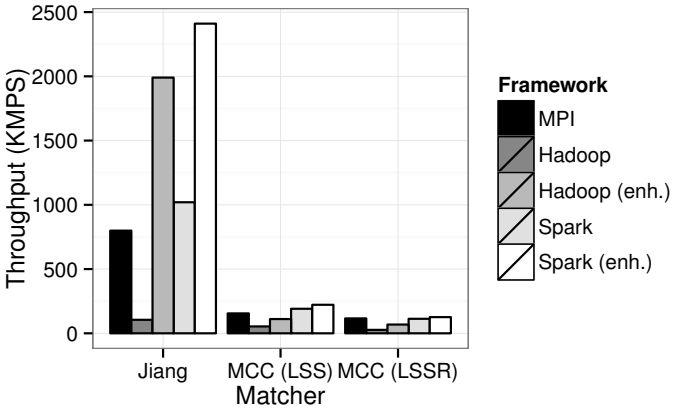


Figure 4: Throughput for the three tested frameworks.

over the Java Virtual Machine [16]. This improvement in the computing time is produced by the proposed decomposition methodology. The use of partial scores increases the level of parallelism, so that the overall matching process is composed by many small computations that can be performed independently. This structure allows to identify bunches of input fingerprints in a flexible way, reusing efficiently the information of the template local structures for several sets of input local structures. Finally, the possibility of safely discarding partial scores enhances the matching process as it allows the early detection of non-matching fingerprints or non-similar fingerprint parts, reducing the overall computation.

An additional factor to take into account is that the flexibility of the Spark RDD model allows the machines to divide the data into small chunks and process them in parallel, so that when a core finishes processing a chunk it can tackle a new one. In other terms, the partitioning of the database performed by the MPI approach is more static than the one performed by Spark. Therefore, the idle time of the cores is reduced in the latter framework.

As for the enhancements carried out on the matchers, there is a significant gain of time in all cases, especially when Hadoop is used, assessing an important reduction of the network and storage workload. The enhanced version of Jiang's matcher attained more than 2 million matches per second.

GPUs have been used by other researchers to develop thoroughly optimized versions of the matching algorithms. Table 7 shows some of the best results reported in the literature so far on SFinGe fingerprints, to the best of the authors' knowledge. Both the hardware

and the parameters used to generate the database vary among these works, which prevents a fair comparison. However, the table gives us an outline of what might be expected from fine-grain optimizations of the matchers in high performance architectures. In general, our proposal has a throughput higher than that obtained by a single server with several GPUs, except in the case of the binary version of MCC implemented in [4].

Table 7: Throughput (in thousands of matches per second) obtained by GPU implementations of the matchers.

	Hardware	Average minutiae	Templates	Cores	Jiang	MCC (LSS)	MCC (LSSR)
GPU-based proposals							
Gutiérrez et al. [12]	2 × Nvidia Tesla M2090	40.7	100 000	1024	–	97.7	54.4
Lastra et al. [25]	2 × Nvidia Tesla K20m 2 × Nvidia Tesla M2090	51.8	800 000	6016	1500.00	–	–
Cappelli et al. [4]	4 × Tesla C2075 GPUs	32.3	250 000	1792	–	35 221.40	–
Our proposal							
Hadoop	20 × 2 × Intel Xeon E5-2620	55.5	400 000	480	105.47	53.47	26.76
Spark					1018.30	190.77	113.12
Hadoop (enhanced)					1994.95	111.41	68.16
Spark (enhanced)					2407.88	222.19	126.49

In order to carry out an objective comparison of these results, some of the approaches listed in Table 7 have been applied to the same SFinGe database used to test our proposal, which contains 400 000 templates. The results obtained, which can be seen in Table 8, show a reduction of the throughput with respect to that presented in the original papers. In the case of MCC, this difference accounts for the higher number of minutiae of the fingerprints in our SFinGe dataset, in combination with the quadratic complexity order of MCC with respect to the minutiae set size. The throughput drop of Jiang's algorithm is larger and cannot be explained exclusively by the increase of minutiae: the offline pre-processing of the local matching performed in the original paper was not carried out in our experiments to keep the experimental environment consistent with the rest of algorithms.

Naturally, the throughput achieved by the GPU algorithms is higher than that obtained with a single CPU. However, one of the strengths of the Big Data frameworks tested with our decomposition proposal is their scalability in terms of number of computing nodes. The obtained results reaffirm this point: the approach described in this paper can obtain a throughput significantly higher than that of the state-of-the art GPU implementations due to its multi-node scaling capabilities.

Table 8: Throughput (in thousands of matches per second) obtained by GPU implementations of the matchers over the SFinGe database.

Hardware	Cores	Jiang [25]	MCC (LSS) [12]	MCC (LSSR) [12]
2 × Nvidia Tesla M2090	1024	174.50	36.33	26.98
2 × Nvidia Tesla K20m 2 × Nvidia Tesla M2090	6016	345.92	51.85	41.19

Table 9 presents some statistics about the executions in Hadoop for the three tested matchers. For the sake of clarity, only the enhanced versions are shown. The table shows that the combine phase is well optimized, as the number of partial scores that are passed to the reducer is very low, requiring less network traffic during the shuffle phase. Most of the

computing time is spent on the map phase, which computes all the local similarities. The computation of the partial scores is efficiently performed within the combiners and reducers.

Table 9: Statistics of the executions with Hadoop

	Jiang	MCC (LSS)	MCC (LSSR)
Number of maps	2.00E+07	1.99E+07	1.99E+07
Number of combines	9.26E+09	1.02E+10	9.65E+09
Number of reduces	4.00E+09	4.00E+09	4.00E+09
Combine inputs	2.21E+01	2.00E+01	2.12E+01
Reduce inputs	1.00E+00	2.56E+00	2.53E+00
Map time	8.38E+01	4.14E+02	6.63E+02
Combine time	2.83E-02	4.93E-02	8.36E-02
Reduce time	3.04E-03	3.40E-06	6.64E-03

Table 10 shows the time of each phase of the Spark workflow. Note that the phases are executed concurrently, so that the overall time is far lower than the sum of the times of the individual phases. First, the database loading is performed in parallel throughout the computing nodes, so that it can be done quickly from the distributed file system, taking into account the size of the data that is loaded. The broadcast of the input fingerprints consumes more time for each fingerprint, but is still acceptable for any practical purpose. The main part of the time is taken by the computation of the matching and the writing of the result to HDFS, both of which consume a similar amount of time.

Table 10: Average time (in seconds) for each step in Spark

	Jiang	MCC (LSS)	MCC (LSSR)
Load templates (total)	6.3490	37.9890	36.4620
Load + broadcast inputs	0.0011	0.0208	0.0251
Matching	0.1631	1.7802	3.0211
Writing	0.1674	1.7951	3.0382

4.2. NIST-SD14 database

The NIST-SD14 database [40] is composed by two impressions of 27 000 rolled fingerprints. In these experiments, the first impressions of the fingerprints were used as templates, whilst 1000 second impressions were randomly selected to be used as input fingerprints. The statistics of the database are shown in Table 11.

Table 11: Statistics of the NIST-SD14 database

		Template	Input
Number of template fingerprints (n_T and n_I)		27 000	1000
Average number of local structures ($\overline{m}$)		213.31	199.253
Size in bytes	**Jiang**	8.96E+08	8.55E+06
	Jiang (enhanced)	7.50E+08	6.48E+06
	Cylinders	1.84E+10	3.31E+08
	Cylinders (enhanced)	9.25E+09	1.60E+08

Table 12 shows the TPR for this database. Again, the floating point operations cause slight differences in the accuracy of the algorithms, although they can be safely ignored.

However, in this case the enhanced version of Jiang's algorithm obtains much better accuracy, assessing its value for rolled prints. The same behavior can be seen with MCC: this indicates that the enhanced approaches, which privilege the local phase of the matching over the global consolidation, work better with rolled prints than with the plain ones generated by SFinGe.

Table 12: TPR obtained with the NIST-SD14 database (for 0% FPR)

	Jiang		**MCC (LSS)**		**MCC (LSSR)**	
	Orig.	Enhan.	Orig.	Enhan.	Orig.	Enhan.
MPI	0.259	–	0.315	–	0.799	–
Hadoop	0.258	0.511	0.341	0.348	0.799	0.845
Spark	0.258	0.482	0.341	0.352	0.803	0.837

The average identification times and the throughput, shown in Table 13 and Fig. 5 respectively, corroborate the results obtained with the artificially generated database. The enhancement has been able to significantly decrease the execution time in all cases. Hadoop is still the slowest framework, but now Spark is significantly faster than MPI, with a much larger difference than with SFinGe. This behavior is due to the higher number of minutiae (and therefore local structures) present in the rolled fingerprints of NIST-SD14. This increases the amount of pairings between the local structures of both templates and input fingerprints, which in turn results in a more fine-grained decomposition of the matching process that allows a better parallelism. As the computation of a single matching between two fingerprints now requires more aggregations of partial scores, the partial scores that are discarded cause a larger saving of computational resources.

Table 13: Average identification times in seconds with the NIST-SD14 database

	Jiang		**MCC (LSS)**		**MCC (LSSR)**	
	Orig.	Enhan.	Orig.	Enhan.	Orig.	Enhan.
MPI	1.1206	–	3.0010	–	4.7101	–
Hadoop	3.6140	1.0031	7.6840	3.6608	14.5230	6.9196
Spark	0.3280	0.1319	2.3338	2.0315	4.1742	2.8834
Ratios						
Hadoop/MPI	3.2250	0.8951	2.5605	1.2199	3.0833	1.4691
Spark/MPI	0.2927	0.1167	0.7777	0.6770	0.8862	0.6122

4.3. Study on the scalability of the proposal

A key objective in the design of the proposed decomposition scheme is to allow for scalable identification systems. This section evaluates the scalability with Hadoop and Spark, using subsets of several sizes of the SFinGe database. The number of input fingerprints was set to 10% the number of templates in each case, half of them being impostor identities, up to a maximum of 10 000. Therefore, $4 \cdot 10^9$ matches were computed for the largest database.

Figure 6 shows the throughput with Hadoop and Spark and the three matchers, as a function of the size of the template database. The plot shows that although Spark has a throughput higher than that of Hadoop, both frameworks present the same behavior when the number of templates is increased: as the computing time becomes larger in proportion

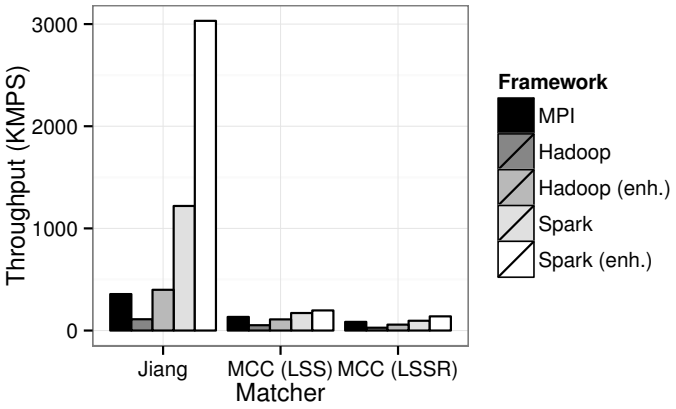


Figure 5: Throughput for the three tested frameworks.

to the communication times, the throughput steadily increases until it remains constant for very large databases. These results assess the scalability of the proposed decomposition scheme in accordance with the tested Big Data frameworks.

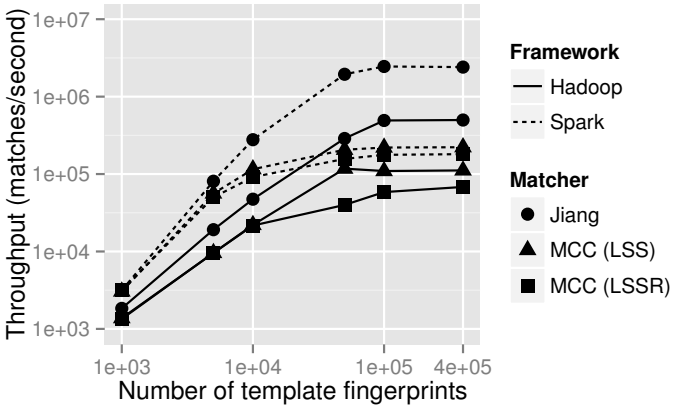


Figure 6: Throughput of the enhanced matchers for different database sizes.

5. Conclusion

Fingerprint identification within very large databases has become a challenging problem for many companies and institutions. There is a need of scalable, flexible methodologies for dealing with this problem. With the raise of Big Data oriented technologies in the last years, new environments highlight as suitable for implanting efficient, highly reliable fingerprint identification systems.

A generic decomposition methodology for fingerprint matching has been proposed in this paper, along with its application on two of the most popular frameworks for Big Data computing: Apache Hadoop and Apache Spark. The computation of the final matching score

for two fingerprints is split at a finer level, defining partial scores between subsets of local structures. These partial scores can be computed independently and merged afterwards, which defines a very flexible procedure and allows some of the partial scores to be discarded. Furthermore, generic and specific guidelines to enhance the performance of the matchers are provided. This methodology was applied over two well-known matching algorithms of the scientific literature.

The experimental results over two large databases reveal a very promising behavior of the proposed decomposition, which is further improved by the application of the aforementioned guidelines. The possibility to discard parts of the matching process enhances the identification time by reducing the communication and synchronization necessities of the system.

Finally, it is noteworthy that the proposed decomposition is generic enough so as not to be limited to minutiae-based fingerprint matching algorithms. It only requires an adequate definition of local structures, partial scores and aggregation functions; thus, other types of matching can be adapted to the proposal as well. Moreover, the approach can also be applied on biometric features other than fingerprints.

Acknowledgements

This work was supported by the research projects TIN2014-57251-P, TIN2013-47210-P and P12-TIC-2958.

References

[1] ISO 19794-2:2011, Biometric data interchange formats – Part 2: Finger minutiae data, Standard, International Organization for Standardization, 2011.

[2] S. Bistarelli, F. Santini, A. Vaccarelli, An Asymmetric Fingerprint Matching Algorithm for Java Card TM, Pattern Analysis and Applications (2006) 359–376.

[3] R. Cappelli, M. Ferrara, D. Maltoni, Minutia cylinder-code: A new representation and matching technique for fingerprint recognition, IEEE Transactions on Pattern Analysis and Machine Intelligence 32 (2010) 2128–2141.

[4] R. Cappelli, M. Ferrara, D. Maltoni, Large-scale fingerprint identification on GPU, Information Sciences 306 (2015) 1–20.

[5] R. Cappelli, D. Maio, D. Maltoni, Synthetic fingerprint-database generation, in: Proceedings of the 16th International Conference on Pattern Recognition, volume 3, pp. 744–747.

[6] C.L.P. Chen, C.Y. Zhang, Data-intensive applications, challenges, techniques and technologies: A survey on Big Data, Information Sciences 275 (2014) 314–347.

[7] F. Chen, X. Huang, J. Zhou, Hierarchical minutiae matching for fingerprint and palmprint identification, IEEE Transactions on Image Processing 22 (2013) 4964–4971.

[8] A. Fernández, S. del Río, V. López, A. Bawakid, M.J. del Jesus, J.M. Benitez, F. Herrera, Big Data with Cloud Computing: an insight on the computing environment, MapReduce, and programming frameworks, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 4 (2014) 380–409.

[9] A. Fiandrotti, M. Mattelliano, E. Baccaglini, P. Vergori, CDVSec: Privacy-preserving biometrical user authentication in the cloud with CDVS descriptors, Pattern Recognition Letters (2017) in press.

[10] M. Galar, J. Derrac, D. Peralta, I. Triguero, D. Paternain, C. Lopez-Molina, S. García, J.M. Benitez, M. Pagola, E. Barrenechea, H. Bustince, F. Herrera, A survey of fingerprint classification Part I:

Taxonomies on feature extraction methods and learning models, Knowledge-Based Systems 81 (2015) 76–97.
[11] L.M. Ghiringhelli, J. Vybiral, S.V. Levchenko, C. Draxl, M. Scheffler, Big Data of Materials Science: Critical Role of the Descriptor, Physical Review Letters 114 (2015) 105503.
[12] P.D. Gutierrez, M. Lastra, F. Herrera, J.M. Benitez, A high performance fingerprint matching system for large databases based on GPU, IEEE Transactions on Information Forensics and Security 9 (2014) 62–71.
[13] M. Haghighat, S. Zonouz, M. Abdel-Mottaleb, CloudID: Trustworthy cloud-based and cross-enterprise biometric identification, Expert Systems with Applications 42 (2015) 7905–7916.
[14] H. Hasan, S.A. Kareem, Fingerprint image enhancement and recognition algorithms: a survey, Neural Computing and Applications 23 (2013) 1605–1610.
[15] M. Hulea, A. Atilean, T. Leia, R. Miron, S. Folea, Fingerprint recognition distributed system, in: Proceedings of the 16th IEEE International Conference on Automation, Quality and Testing, Robotics, volume 3, pp. 423–428.
[16] R. Hundt, Loop recognition in C++/Java/Go/Scala, Proceedings of Scala Days 2011 (2011) 38.
[17] A.K. Jain, J. Feng, Latent fingerprint matching, IEEE Transactions on Pattern Analysis and Machine Intelligence 33 (2011) 88–100.
[18] A.K. Jain, L. Hong, R. Bolle, On-line fingerprint verification, IEEE Transactions on Pattern Analysis and Machine Intelligence 19 (1997) 302–314.
[19] A.K. Jain, L. Hong, S. Pankanti, R. Bolle, An identity-authentication system using fingerprints, Proceedings of the IEEE 85 (1997) 1365–1388.
[20] J. Jia, L. Cai, P. Lu, X. Liu, Fingerprint matching based on weighting method and the SVM, Neurocomputing 70 (2007) 849–858.
[21] X. Jiang, M. Liu, A.C. Kot, Fingerprint retrieval for identification, IEEE Transactions on Information Forensics and Security 1 (2006) 532–542.
[22] X. Jiang, W.Y. Yau, Fingerprint minutiae matching based on the local and global structures, in: Proceedings of the 15th International Conference on Pattern Recognition, pp. 1038–1041.
[23] H. Karau, A. Konwinski, P. Wendell, M. Zaharia, Learning Spark: Lightning-Fast Big Data Analysis, O'Reilly Media, Inc., 2015.
[24] T. Kitano, L. Su, SPOAN: Load Balancing Replica Placement Strategy for Large Scale Biometric Identification Service, in: IEEE International Congress on Big Data, IEEE, 2013, pp. 326–333.
[25] M. Lastra, J. Carabaño, P.D. Gutierrez, J.M. Benitez, F. Herrera, Fast fingerprint identification using GPUs, Information Sciences 301 (2015) 195–214.
[26] D. Maltoni, D. Maio, A.K. Jain, S. Prabhakar, Handbook of fingerprint recognition, Springer, New York, 2009.
[27] V. Marx, The big challenges of Big Data, Nature 498 (2013) 255–260.
[28] R.F. Miron, T.S. Letia, M. Hulea, Two server topologies for a distributed fingerprint-based recognition system, in: 15th International Conference on System Theory, Control and Computing, pp. 1–6.
[29] S. Pankanti, S. Prabhakar, A.K. Jain, On the Individuality of Fingerprints, IEEE Transactions on Pattern Analysis and Machine Intelligence 24 (2002) 1010–1025.
[30] P. Peer, J. Bule, J.Z. Gros, V. Struc, Building cloud-based biometric services, Informatica (Slovenia) 37 (2013) 115–122.
[31] D. Peralta, M. Galar, I. Triguero, D. Paternain, S. García, E. Barrenechea, J.M. Benitez, H. Bustince, F. Herrera, A survey on fingerprint minutiae-based local matching for verification and identification: Taxonomy and experimental evaluation, Information Sciences 315 (2015) 67–87.
[32] D. Peralta, I. Triguero, R. Sanchez-Reillo, F. Herrera, J.M. Benitez, Fast Fingerprint Identification for Large Databases, Pattern Recognition 47 (2014) 588–602.
[33] J. Qi, Y. Wang, A robust fingerprint matching method, Pattern Recognition 38 (2005) 1665–1671.
[34] N.K. Ratha, K. Karu, S. Chen, A.K. Jain, A real-time matching system for large fingerprint databases, IEEE Transactions on Pattern Analysis and Machine Intelligence 18 (1996) 799–813.
[35] Shelly, N.S. Raghava, Iris recognition on Hadoop: A biometrics system implementation on cloud com-

puting, in: IEEE International Conference on Cloud Computing and Intelligence Systems, IEEE, 2011, pp. 482–485.
[36] H.S. Stone, High-performance computer architecture, Addison-Wesley, Reading, USA, 1992.
[37] X. Tan, B. Bhanu, A Robust Two Step Approach for Fingerprint Identification, Pattern Recognition Letters 24 (2003) 2127–2134.
[38] H. Wang, Z. Xu, H. Fujita, S. Liu, Towards felicitous decision making: An overview on challenges and trends of Big Data, Information Sciences 367 (2016) 747–765.
[39] Y. Wang, L. Wang, Y.M. Cheung, P.C. Yuen, Learning compact binary codes for hash-based fingerprint indexing, IEEE Transactions on Information Forensics and Security 10 (2015) 1603–1616.
[40] C.I. Watson, NIST Special Database 14, Technical Report, NIST, 1993.
[41] C.I. Watson, M.D. Garris, E. Tabassi, C.L. Wilson, R.M. Mccabe, S. Janet, K. Ko, User's Guide to NIST Biometric Image Software (NBIS), Technical Report, NIST, 2010.
[42] T. White, Hadoop: The Definitive Guide, O'Reilly Media, Inc., 3rd edition, 2012.
[43] X. Wu, X. Zhu, G.Q. Wu, W. Ding, Data mining with big data, IEEE Transactions on Knowledge and Data Engineering 26 (2014) 97–107.
[44] Z. Wu, L. Tian, P. Li, T. Wu, M. Jiang, C. Wu, Generating stable biometric keys for flexible cloud computing authentication using finger vein, Information Sciences (2017) in press.
[45] J. Yu, D. Liu, D. Tao, H.S. Seah, Complex Object Correspondence Construction in Two-Dimensional Animation, IEEE Transactions on Image Processing 20 (2011) 3257–3269.
[46] J. Yu, D. Tao, J. Li, J. Cheng, Semantic preserving distance metric learning and applications, Information Sciences 281 (2014) 674–686.
[47] J. Yuan, S. Yu, Efficient privacy-preserving biometric identification in cloud computing, in: Proceedings of the IEEE INFOCOM, pp. 2652–2660.
[48] Y. Zhao, W. Zhang, D. Li, Z. Huang, DFIS: A scalable distributed fingerprint identification system, Proceedings of the 15th International Conference on Algorithms and Architectures for Parallel Processing 9530 (2015) 162–175.
[49] Y.X. Zhao, W.X. Zhang, D.S. Li, Z. Huang, M.N. Li, X.C. Lu, Pegasus: a distributed and load-balancing fingerprint identification system, Frontiers of Information Technology and Electronic Engineering 17 (2016).
[50] E. Zhu, J. Yin, G. Zhang, Fingerprint matching based on global alignment of multiple reference minutiae, Pattern Recognition 38 (2005) 1685–1694.