Parallel Tree Kernel Computation

Souad Taouti[1], Hadda Cherroun[1] and Djelloul Ziadi[2]
[1]*LIM, Université UATL Laghouat, Algeria*
[2]*Groupe de Recherche Rouennais en Informatique Fondamentale, Université de Rouen Normandie, France*

Keywords: Kernel Methods, Structured Data Kernels, Tree Kernels, Tree Series, Root Weighted Tree Automata, MapReduce, Spark, Parallel Automata Intersection.

Abstract: Tree kernels are fundamental tools that have been leveraged in many applications, particularly those based on machine learning for Natural Language Processing tasks. In this paper, we devise a parallel implementation of the sequential algorithm for the computation of some tree kernels of two finite sets of trees (Ouali-Sebti, 2015). Our comparison is narrowed on a sequential implementation of SubTree kernel computation. This latter is mainly reduced to an intersection of weighted tree automata. Our approach relies on the nature of the data parallelism source inherent in this computation by deploying both MapReduce paradigm and Spark framework. One of the key benefits of our approach is its versatility in being adaptable to a wide range of substructure tree kernel-based learning methods. To evaluate the efficacy of our parallel approach, we conducted a series of experiments that compared it against the sequential version using a diverse set of synthetic tree language datasets that were manually crafted for our analysis. The reached results clearly demonstrate that the proposed parallel algorithm outperforms the sequential one in terms of latency.

1 INTRODUCTION

Trees are basic data structures that are naturally used in real world applications to represent a wide range of objects in structured form, such as XML documents (Maneth et al., 2008), molecular structures in chemistry (Gordon and Ross-Murphy, 1975) and parse trees in natural language processing (Shatnawi and Belkhouche, 2012).

In (Haussler et al., 1999), Haussler provides a framework based on convolution kernels, which find the similarity between two structures by summing the similarity of their substructures. Many convolution kernels for trees are presented based on this principle and have been effectively used to a wide range of data types and applications.

Tree kernels, which were initially presented in (Collins and Duffy, 2001; Collins and Duffy, 2002), as specific convolution kernels, have been shown to be interesting approaches for the modeling of many real world applications. Mainly those related to Natural Language Processing tasks, e.g. named entity recognition and relation extraction (Nasar et al., 2021), text syntactic-semantic similarity (Alian and Awajan, 2023), detection of text plagiarism (Thom, 2018), topic-to-question generation (Chali and Hasan, 2015), source code plagiarism detection (Fu et al., 2017), linguistic pattern-aware dependency which captures chemical–protein interaction patterns within biomedical literature (Warikoo et al., 2018).

Subtree (ST) kernel (Vishwanathan and Smola, 2002) and the subset tree (SST) (Collins and Duffy, 2001) were the initially kernels introduced in the context of trees. The compared segments in ST Kernel, are subtrees, a node and its entire descendancy. While in the SST Kernel, the considered segments are subset-trees, a node and its partial descendancy.

The principle of tree kernels, as initially presented, is to compute the number of shared substructures (subtrees and subset trees) between two trees t_1 and t_2 with m and n nodes, respectively. It may be computed recursively as follows:

$$K(t_1,t_2) = \sum_{(n_1,n_2)\in N_{t_1}\times N_{t_2}} \Delta(n_1,n_2) \qquad (1)$$

where N_{t_1} and N_{t_2} are the number of nodes in t_1 and t_2 respectively and $\Delta(n_1,n_2) = \sum_{i=1}^{|S|} I_i(n_1).I_i(n_2)$ for some finite set of subtrees $S = \{s_1, s_2, \dots\}$, and $I_i(n)$ is an indicator function which is equal to 1 if the subtree is rooted at node n and to 0 otherwise.

An approach for computing the tree kernels of two finite sets of trees was proposed in (Mignot et al.,

Taouti, S., Cherroun, H. and Ziadi, D.
Parallel Tree Kernel Computation.
DOI: 10.5220/0012386500003654

In *Proceedings of the 13th International Conference on Pattern Recognition Applications and Methods (ICPRAM 2024)*, pages 329-336
ISBN: 978-989-758-684-2; ISSN: 2184-4313

2015). That makes use of Rooted Weighted Tree Automata (RWTA) that are a class of weighted tree automata. Subtree, Subset tree, and Rooted tree kernels can be computed using a general intersection of RWTAs associated with the two finite sets of trees and then the computation of weights on the resulting automaton.

In this paper, we narrow our study to the parallel implementation using MapReduce paradigm and Spark framework of the construction of the RWTA from a finite set of trees (as this step is a common base for other trees kernels) and to the comparison with the sequential linear algorithm proposed in (L. Mignot and Ziadi, 2023) for SubTree kernel computation. The main motivation behind this proposal is that despite the linear complexity of the sequential version, it remains complex when we consider Machine Learning computational cost requirements.

We begin by defining RWTA. Next, we present the sequential proposed algorithms. Then we provide our parallel implementation based on MapReduce and Spark programming model.

The rest of the paper is organized as follows: Section 2 introduces Tree Kernels and Automata while presenting the sequential proposed algorithms. Section3 presents the details of the parallel implementation of the tree kernel computation based on MapReduce and Spark. Some experimental results and evaluations are shown in Section 4. Finally, the conclusion and perspectives are presented in Section 5

2 TREE KERNELS AND AUTOMATA

Let Σ be a graded alphabet, t is a tree over Σ, defined initially as $t = f(t_1,\ldots,t_k)$ where k can be any integer, f any symbol from Σ_k and $t_1,\ldots,t_k$ are any k trees over Σ. The set of trees over Σ is referred as T_Σ. A tree language over Σ is a subset of T_Σ.

Let $\mathbb{M} = (M,+)$ be a monoid with identity is 0, a formal tree series $\mathbb{P}$ (Collins and Duffy, 2002) (Ésik and Kuich, 2002) over a set S represents a mapping from T_Σ to S, where its support is the set $\text{Support}(\mathbb{P}) = \{t \in T_\Sigma | (\mathbb{P},t) \neq 0\}$. Any formal tree series is consistent to a formal sum $\mathbb{P} = \sum_{t\in T_\Sigma}(\mathbb{P},t)t$, which is in this case both associative and commutative.

Weighted tree automata can realize formal tree series. In this paper, we employ specific automata, and the weights just indicate the finality of states. As a result, the automata we use are a particular subclasses of weighted tree automata.

2.1 Root Weighted Tree Automata

Definition 1. *Let $\mathbb{M} = (M,+)$ be a commutative monoid. An $\mathbb{M}$-Root Weighted Tree Automaton ($\mathbb{M}$-RWTA) is a 4-tuple (Σ,Q,μ,δ) with the following properties:*

- *$\Sigma = \bigcup_{k\in\mathbb{N}}\Sigma_k$: a graded alphabet,*
- *Q: a finite set of states,*
- *μ: the root weight function, is a function from Q to M,*
- *δ: the transition set, is a subset of $Q\times\Sigma_k\times Q^k$.*

A $\mathbb{M}$-RWTA is referred as RWTA, if there is no ambiguity.
The root weight function μ is extended to $2^Q \to M$ for each subset S of Q by $\mu(S) = \Sigma_{s\in S}\mu(s)$. The function μ is equivalent to the finite subset of $Q\times M$ defined for any couple (q,m) in $Q\times M$ by $(q,m)\in\mu \iff \mu(q) = m$.
For any subset S of Q, the root weight function μ is expanded to $2^Q \to M$ by $\mu(S) = \Sigma_{s\in S}\mu(s)$. The function μ
The transition set δ corresponds to the function in $\Sigma_k \times Q^k \to 2^Q$ defined for any symbol f in Σ_k and for any k-tuple $(q_1,\ldots,q_k)$ in Q^k by

$$q\in\delta(f,q_1,...,q_k) \iff (q,f,q_1,\ldots,q_k)\in\delta.$$

The function δ is extended to $\Sigma_k\times(2^Q)^k \to 2^Q$ as follows: for any symbol f in Σ_k, for any k-tuple $(Q_1,\ldots,Q_k)$ of subsets of Q,

$$\delta(f,Q_1,\ldots,Q_k) = \bigcup_{(q_1,\ldots,q_k)\in Q_1\times\cdots\times Q_k}\delta(f,q_1,\ldots,q_k).$$

Finally, the function Δ is the function from T_Σ to 2^Q defined for any tree $t = f(t_1,\ldots,t_k)$ in T_Σ by

$$\Delta(t) = \delta(f,\Delta(t_1),\ldots,\Delta(t_k)).$$

A weight of a tree t in a $\mathbb{M}$-RWTA A is $\mu(\Delta(t))$. The formal tree series realized by A is the formal tree series over M denoted by $\mathbb{P}_A$ and defined by $\mathbb{P}_A = \sum_{t\in T_\Sigma}\mu(\Delta(t))$, with $\mu(\emptyset) = 0$ where 0 is the identity of $\mathbb{M}$.

Example 1. *Let us consider the graded alphabet Σ defined by $\Sigma_0 = \{a,b\}$, $\Sigma_1 = \{g,h\}$ and $\Sigma_2 = \{f\}$. Let $\mathbb{M} = (\mathbb{N},+)$. The RWTA $A = (\Sigma,Q,\mu,\delta)$ defined by*

- $Q = \{1,2,3,4,5,6\}$,
- $\mu = \{(1,1),(2,4),(3,3),(4,1),(5,2),(6,3)\}$,
- $\delta = \{(1,a),(3,b),(2,h,1),(4,g,3),(5,f,2,3),(6,f,4,5)\}$,

This RWTA is represented in Figure 1. It realizes the following tree series: $\mathbb{P}_A = 1a+3b+4h(a)+1g(b)+2g(h(a))+3f(g(h(a)),g(b))$

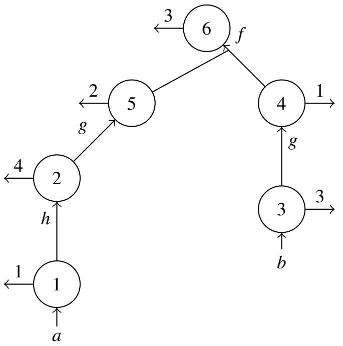

Figure 1: The RWTA A.

A RWTA appears to be a prefix tree constructed in the context of words. A finite set of trees could be represented by this compact structure.

In addition, many substructures' tree can be computed through this compact structure. In what follow, we introduce Subtree, Rooted tree and SubSet tree and their related automata (Ouali-Sebti, 2015). However, in this paper, we narrow our Kernel computation proposal to the SubTree case.

2.2 Subtree

Definition 2. *Let Σ be a graded alphabet and $t = f(t_1,\ldots,t_k)$ a tree in T_Σ. The set* $\text{SubTree}(t)$ *is the set defined inductively by:*

$$\text{SubTree}(t) = \{t\} \cup \bigcup_{1\leq j\leq k} \text{SubTree}(t_j).$$

Let L be a tree language over Σ. The set $\text{SubTreeSet}(L)$ is the set defined by:

$$\text{SubTreeSet}(L) = \bigcup_{t\in L} \text{SubTree}(t).$$

The formal tree series $\text{SubTreeSeries}(t)$ is the tree series over $\mathbb{N}$ inductively defined by:

$$\text{SubTreeSeries}(t) = t + \sum_{1\leq j\leq k} \text{SubTreeSeries}(t_j).$$

If L is finite, the rational series $\text{SubTreeSeries}(L)$ is the tree series over $\mathbb{N}$ defined by:

$$\text{SubTreeSeries}(L) = \sum_{t\in L} \text{SubTreeSeries}(t).$$

Example 2. *Let Σ be the graded alphabet defined by $\Sigma_0 = \{a,b\}$, $\Sigma_1 = \{g,h\}$ and $\Sigma_2 = \{f\}$.*

Let t be the tree $f(h(a),g(b))$, we have:
$\text{SubTree}(t) = \{a,b,h(a),g(b),f(h(a),g(b))\}$
$\text{SubTreeSet}(t) = t + h(a) + g(b) + a + b$

Definition 3. *Let Σ be a graded alphabet. Let t be a tree in T_Σ. The Subtree automaton associated with t is the RWTA $A_t = (\Sigma, Q, \mu, \delta)$ defined by:*

- $Q = \text{SubTreeSet}(t)$,
- $\forall s \in Q, \quad \mu(s) = (\text{SubTreeSeries}(t), s)$,
- $\forall f \in \Sigma, \forall s_1,\ldots,s_{k+1} \in Q, s_{k+1} \in \delta(f,s_1,\ldots,s_k) \iff s_{k+1} = f(s_1,\ldots,s_k)$.

This Weighted Tree Automaton (RWTA) requires less storage space because its states are exactly its subsets.

Example 3. *Given the tree defined by $t = f(f(h(a),b),g(b))$. The RWTA A_t associated with the tree t is represented in Figure 2.*

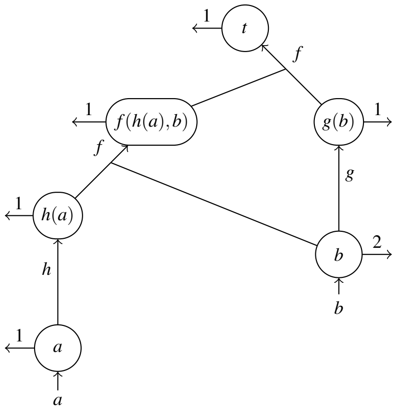

Figure 2: The RWTA A_t associated with the tree $t = f(f(h(a),b),g(b))$.

2.3 Rooted Tree

Definition 4. *Let Σ be a graded alphabet and $t = f(t_1,\ldots,t_k)$ a tree in T_Σ. We indicate with Σ', the set $\Sigma \cup \{\bot\}$, where $\bot \in \Sigma'_0$ and $\bot \notin \Sigma$.* $\text{PrefixSet}(t)$ *is the set of trees on T'_Σ inductively defined by:*

$$\text{PrefixSet}(t) = \{t\} \cup f(\bot,\ldots,\bot)$$
$$\cup\, f(\text{PrefixSet}(t_1),\ldots,\text{PrefixSet}(t_k))$$

It should be noted that $\bot$ is not a prefix of t.

Let L be a tree language over Σ. The set $\text{PrefixSet}(L)$ is the set defined by:

$$\text{PrefixSet}(L) = \bigcup_{t\in L} \text{PrefixSet}(t).$$

The formal tree series $\text{PrefixSeries}(t)$ is the tree series over $\mathbb{N}$ inductively defined by:

$$\text{PrefixSeries}(t) = \sum_{t'\in \text{PrefixSet}(t)} t'$$

If L is finite, the series $\text{PrefixSeries}(L)$ is the tree series over $\mathbb{N}$ defined by:

$$\text{PrefixSeries}(L) = \sum_{t'\in L} \text{PrefixSeries}(t')$$

If L is not finite, as Σ is a finite set of symbols, there exists a symbol f in Σ^k such that $f(\bot,\ldots,\bot)$ occurs as a prefix infinite number of times in L. Therefore, $\text{PrefixSeries}(L)$ is a series of trees over $\mathbb{N} \cup \{+\infty\}$.

Definition 5. *Let Σ be a graded alphabet. Let t be a tree in T_Σ. The automaton of prefixes associated with t is the RWTA $A_t = (\Sigma', Q, \mu, \delta)$ defined by:*

- $Q = \mathrm{SubTreeSet}(t) \cup \{\perp\}$,
- $\forall t' \in Q, \quad \mu(t') = \begin{cases} 1, & if\, t' = t, \\ 0, & else, \end{cases}$
- $\forall t' = f(t_1, \dots, t_k) \in Q, \delta(f, t_1, \dots, t_k) = t'$
- $\forall f \in \Sigma^k, \delta(f, \perp, \dots, \perp) = \{f(t_1, \dots, t_k) \in Q\}$.

2.4 SubSet Tree (SST)

Definition 6. *Let Σ be a graded alphabet and $t = f(t_1, \dots, t_k)$ a tree in T_Σ. We indicate with Σ', the set $\Sigma \cup \{\perp\}$, where $\perp \in \Sigma'_0$ and $\perp \notin \Sigma$.*
The set SSTSet(t) is the set of trees on $T_{\Sigma'}$ defined by: $SSTSet(t) = PrefixSet(SubtreeSet(t))$

Let L be a tree language over Σ. The set SSTSeries(L) is the set defined by:

$$SSTSeries(L) = \textstyle\sum_{t' \in L} SSTSeries(t)t'.$$

The formal tree series SSTSeries(t) is the tree series over $\mathbb{N}$ inductively defined by:

$$SSTSeries(t) = SubtreeSeries(prefixSet(t))$$

Definition 7. *Let Σ be a graded alphabet. Let t be a tree in T_Σ. The SST automaton associated with t is the RWTA $A_t = (\Sigma', Q, \mu, \delta)$ defined by:*

- $Q = \mathrm{SubTreeSet}(t) \cup \{\perp\}$,
- $\forall t' \in Q, \quad \mu(t') = \begin{cases} 1, & if\, t' = t, \\ 0, & else, \end{cases}$
- $\forall t' = f(t_1, \dots, t_k) \in Q, \delta(f, t_1, \dots, t_k) = t'$
- $\forall f \in \Sigma^k, \delta(f, \perp, \dots, \perp) = \{f_j(t_1, \dots, t_k) \in Q | h(f_j) = f\}$.

2.5 Sequential Kernel Computation

In order to compute the kernel of two finite tree languages X and Y, we act in three steps:

1. First, we construct both RWTAs A_X and A_Y.
2. Then we compute the intersection of A_X and A_Y;
3. Finally, the kernel is simply computed through a sum of all of the root weights of this RWTA.

One can easily observe that the set of states, denoted by Q, is equal to SubTreeSet(t) in ST, plus $\{\perp\}$ for both Rooted Tree and SubSet Tree. However, their root weight function (μ) are different. In addition, their δ, denoting the transition functions, are defined from the ST transition table. Consequently, the RWTA construction step remains the same for the three tree substructures while the RWTA intersection step is distinct for each tree substructure.

Let us recall that in this paper we narrow the RWTA intersection and kernel computation to the ST kernel case.

Initially, we introduce the sequential step-by-step procedure that enable us to efficiently calculate tree kernels by utilizing the intersection of tree automata.

2.5.1 RWTA Construction

In this section, we describe through an algorithm the construction of an RWTA from a finite set of trees. Consider the finite set of trees X, first, we extract all the prefixes of each tree in X and sum their number of occurrences of each tree in X, which is equivalent to the sum of the subtrees series.

$$\mathrm{SubTreeSeries}(X) = \textstyle\sum_{t \in X} \mathrm{SubTreeSeries}(t)$$

Algorithm 1 constructs an RWTA from a finite set of trees.

Algorithm 1: Computation of Automaton A_X from X.

```
Input  : $X$: Set of trees
Output: RWTA $A_X$ = $(\Sigma, Q, \mu, \delta)$
$Q_X = \emptyset$;
foreach $t \in X$ do
    if $t \notin Q_X$ then
        Add($Q_X$, $t$);
        $\mu_X(t) \leftarrow 1$;
    else
        $\mu_X(s) \leftarrow \mu_X(s) + 1$;
    end
end
```

2.5.2 RWTA Intersection

Definition 8. *Let $\sum$ be an alphabet. Let X and Y be two finite tree languages over Σ. The Tree Series of (X, Y) is defined by:*

$$\mathrm{SubTreeSeries}((X,Y)) = \textstyle\sum_{t \in T_\Sigma} (\mathrm{SubTreeSeries}(X), t) \times (\mathrm{SubTreeSeries}(Y), t).$$

Example 4. *Let Σ be the graded alphabet defined by $\Sigma_0 = \{a, b\}$, $\Sigma_1 = \{g, h\}$ and $\Sigma_2 = \{f\}$. Let us consider the three trees $t_1 = f(f(h(a), b), g(b))$, $t_2 = f(h(a), g(b))$ and $t_3 = f(f(h(a), b), f(h(a), g(b)))$.*
We have:

$\mathrm{SubTreeSeries}(t_1) = t_1 + f(h(a), b) + h(a) + g(b) + a + 2b$

$\mathrm{SubTreeSeries}(t_2) = t_2 + h(a) + g(b) + a + b$

SubTreeSeries(t_3) $= t_3 + f(h(a),b) + t_2 + 2h(a) + g(b) + 2a + 2b$

SubTreeSeries$(\{t_1,t_2\})$ $= t_1 + t_2 + 2f(h(a),b) + 2h(a) + 2g(b) + 2a + 3b$

SubTreeSeries$((\{t_1,t_2\},\{t_3\}))$ $= t_2 + f(h(a),b) + 4h(a) + 4g(b) + 4a + 3b$

By definition, any t in T_Σ, is in Q if and only if $t \in$ SubTreeSet$(X) \cap$ SubTreeSet(Y). Moreover, by definition of μ, for any tree t, $\mu(t) = \mu_X(t) \times \mu_Y(t)$, as for any tree t, if $t \notin$ SubTreeSet(X) (resp. $t \notin$ SubTreeSet(Y)), then $\mu_X(t) = 0$ (resp. $\mu_Y(t) = 0$).

In order to compute the kernel from two RWTA, Algorithm 2 below, loops through the states of A_X, if any state q is present in A_Y then it is added to the automaton $A_{(X,Y)}$ with $\mu_{(X,Y)}(q) = \mu_X(q) \times \mu_Y(q)$, respectively to A_Y.

Algorithm 2: Computation of Automaton $A_{(X,Y)} = A_X \times A_Y$.

```
Input  : 2 RWTA $A_X$ and $A_Y$
Output: an RWTA $A_{(X,Y)} = A_X \times A_Y$
$A_{(X,Y)} \leftarrow A_Y$;
foreach $s \in Q_X$ do
    if $s \in Q_Y$ then
        $\mu_{(X,Y)}(s) \leftarrow \mu_X(s) \times \mu_Y(s)$;
    else
        $\mu(s) \leftarrow 0$;
    end
end
```

2.5.3 Subtree Kernel Computation

The RWTA for both tree sets and their intersection is constructed, allowing for the computation of the tree kernel in the subtree case. Two finite tree languages, X and Y, are given, and Z is the accessible part of their intersection tree automaton $A_{(X,Y)}$. Then, the kernel is simply computed through the sum of the weights:

$$\text{TreeKernel}(X,Y) = \sum_{q \in Z} \mu(q).$$

3 PARALLEL TREE KERNEL COMPUTATION

In light of the inherent data parallelism within Tree Kernel Computation, we have developed and implemented a parallel adaptation of the previously sequential SubTree Kernel Computation, leveraging both MapReduce and Spark paradigms. The frameworks facilitate parallel execution in a distributed environment and offer advanced features for distributed computing, eliminating the need for manual task coordination. By breaking down large tasks into smaller, concurrently executable chunks, it streamlines job scheduling, bolsters fault tolerance, enhances distributed aggregation, and simplifies other management tasks. Kernel Computations bear a striking resemblance to the Big Data paradigm, which poses significant challenges compared to traditional data processing methods. While numerous solutions have been proposed to address the computational and storage challenges of Big Data, the MapReduce and Spark frameworks stand out as prominent methods. Before delving into our parallel implementation, we will provide an explanation of the MapReduce and Spark frameworks in the following sections.

3.1 MapReduce Framework

Map-Reduce was created by Google as a parallel distributed programming approach that works on a cluster of computers due to large-scale data (Dayalan, 2004). It is termed parallel because tasks are executed by dedicating multiple processing units in a parallel environment, and distributed over distinct storage. Hadoop is among most popular open-source MapReduce implementation created primarily by the Apache Software Foundation.

MapReduce is a popular framework for proposing programs without infrastructure complexity due to its stable, easy-to-use, abstract, and scalable environment, with its programming paradigm outlined through basic MapReduce jobs.

3.1.1 Map Function

A Map-Reduce task involves mapping input data to specific reducers, which generate a list of key, value pairs for each unit of data based on a mapping schema. Key pairs from the same list are collected in the same reducer. The mapping schema is the most crucial element, affecting precision, time complexity, and space complexity.

3.1.2 Reduce Function

The output of mappers serves as input for the reducer function, which receives a key associated with a list of records. The output of each reducer is pairs of $< key, value >$, which can produce multiple values with the same keys. The reducer function is applied in parallel to the input list of data and written to the Distributed File System.

3.1.3 MapReduce Implementations

Hadoop Framework. Hadoop is an open-source software framework that enables the efficient storage and processing of large volumes of data across clusters of computers. It consists of several key components, such as the Hadoop Distributed File System (HDFS) and the MapReduce programming model, which enable parallel processing across the cluster. Hadoop also provides tools for data ingestion, processing, analysis, and resource management, making it a popular choice for big data analytics and machine learning applications.

Spark Framework. Apache Spark is an open-source distributed data processing framework, renowned for its speed, adaptability, and versatility in handling big data tasks. It supports various data processing tasks, including batch processing, real-time stream processing and machine learning. Spark's in-memory processing enhances computation speed.

3.2 Parallel RWTA Construction

Let X be a finite tree language. To construct the RWTA A_X from X based on MapReduce paradigm, we have to determine the Map and Reduce jobs. The prefixes of X are listed in a file which is used as an input.

First, the Map function (Algorithm 3) splits the prefixes list into subtrees. Next, the Map function distributes the key-value pairs as follows: $< key, 1 >$, where the key represents the subtree, and the value is 1 indicating the number of occurrence. At this step it represents its presence.

Then, the Map function sends the key-value pairs to reducers by key, so if a subtree appears multiple times, it will be sent to the same reducer multiple times. This principle is similar to the popular word count program where words are subtree.

Next, the reducer (Algorithm 4) sums the value which is the number of occurrences and is equivalent to the weight of the subree. Finally, the subtrees are merged in tree series.

3.3 Parallel RWTA Intersection

Let X and Y be two finite tree languages, A_X and A_Y are their respective RWTA. In order to compute the intersection of automata A_X and A_Y using MapReduce programming model, we have to identify the Map and Reduce jobs. Let us mention that we have both RWTA details saved, from the last Construction step, as an input file, each of them in a separate line.

First, the Map function (Algorithm 5) splits the tree series for X and Y resp. into subtrees with their weights in list. Then, the Map function distributes the key-value pairs as follows: $<$subtree$, (1, weight)>$, where the key represents the subtree, and the value is composed of the weight of the subtree and the number 1 that acts as a Boolean indicating whether or not the subtree is present in the RWTA.

Next, the Map function distributes its output to the reducers by key i.e if any subtree is present in different RWTA, it will be sent to the same reducer.

After that, every reducer (Algorithm 6) sums the first part of the value, which indicates the presence of the subtree in the RWTA to check if it is present in both RWTAs. Then, if the presence is equal to 2 the reducer multiplies the weights of the received subtree from A_X and A_Y.

Algorithm 3: Map Function for the construction of A_X from X.

```
Input  : ($X$: Set of trees) file
while $file \neq empty$ do
    Split(line, $S$);
    foreach $s \in S$ do
        Emit($s$, 1);
    end
end
```

Algorithm 4: Reduce Function for the construction of A_X from X.

```
Input  : mapped $<s, 1>$
Output: RWTA $A_X$
Weight$_s \leftarrow 0$;
forall all mapped $s$ do
    Weight$_s \leftarrow$ Weight$_s + 1$;
end
Add($A_X$, ($s$, Weight$_s$));
```

Algorithm 5: Map Function for the computation of the automaton $A_X \times A_Y$.

```
Input  : (RWTA $A_X$, $A_Y$) file
while $file \neq empty$ do
    Split(line, $Q$);
    for $s \in Q$ do
        Emit($s$, (1, $\mu(s)$));
    end
end
```

Algorithm 6: Reduce Function for the computation of the intersection automaton $A_X \times A_Y$.

```
Input  : $<s,(1,\mu(s))>$
Output: RWTA $A_X \times A_Y$
Presence$_s \leftarrow 0$;
Weight$_s \leftarrow 1$;
forall mapped $s$ do
    Presence$_s \leftarrow$ Presence$_s + 1$;
    Weight$_s \leftarrow$ Weight$_s \times \mu(s)$;
end
if $Presence_s = 2$ then
    Add($A_X \times A_Y, (s,$ Weight$_s)$);
end
```

4 EXPERIMENTS AND RESULTS

To analyse our parallel RWTA-based SubTree Kernel computation we have performed a batch of comparative experiments in order to demonstrate the difference in terms of latency between our parallel algorithm and the sequential one using MapReduce and Spark frameworks.

Additionally, we use the absolute acceleration metric defined by $A^{abs} = T_{seq}/T_{par}$ where T_{seq} and T_{par} are the running times of the sequential and the parallel algorithms respectively.
Prior to presenting our findings, let us initially provide a description of the benchmark we constructed and outline the implementation details.

4.1 Dataset

In order to perform the comparative study of both variants of algorithms, we need a testbed of multiple datasets that cover the variety wide of tree characteristics in order to have a deep algorithm analysis, which is not the case in the real world datasets that are standard benchmarks for learning on relatively small trees. For that purpose, in our experiments, we are brought in generating synthetic datasets.

For this building dataset task, we have considered into account mainly three criteria: i) the alphabet size (varying between 2 and 12), ii) the range of the maximal alphabet arity (between 1 and 5), and iii) the tree depth (TD) that we have varied within the range of 10 to 50.

The constructed tree datasets are divided into two batches according to the alphabet size in [2,12]. The first batch gathered four datasets $D1$, $D2$, $D3$ and $D4$. Each of them contains two tree sets generated using this above principle. Into each batch, each dataset has different size that we have classified according to their average tree size in three classes (small: less than 2GB, medium: less than 2.5GB, and large: more than 4GB). Table 1 illustrates more details on the generated datasets.

Table 1: Details on the generated datasets.

	Trees	Σ	Arity	TD	size Gb
$D1$	500	[2, 12]	[1, 5]	[10,20]	1.5
$D2$	800	[2, 12]	[1, 5]	[10, 50]	2.5
$D3$	3000	[2, 12]	[2, 5]	[10, 50]	4.4
$D4$	4500	[2, 12]	[2, 5]	[10, 50]	7

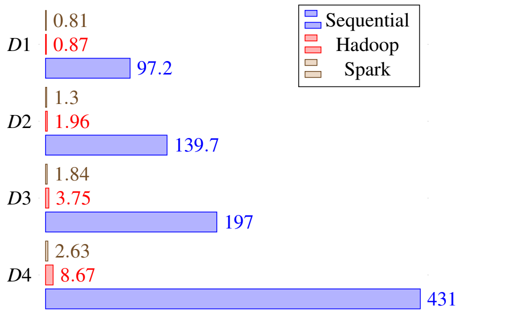

Figure 3: Performances of Parallel algorithm vs the Sequential one in terms of running time (minutes).

Both sequential and parallel algorithms are implemented in Java 11. All experiments were performed on a server equipped with an Intel(R) Xeon(R) Silver 4216 CPU (2.10GHz) processor with 32 cores and 128GB of RAM running Linux.

Sequential and parallel codes in addition to the generated datasets are available on Github.

In this study, we have established a fixed cluster architecture, utilizing Docker containers for conducting all tests. Our cluster comprises one container designated as the Master, and five containers serving as Slaves. It is important to note that this choice of cluster configuration was made arbitrarily for the purpose of this research. The Hadoop [1] V 3.3.0 is installed as MapReduce implementation platform on our cluster and Spark [2] V 3.4.1 to serve as the infrastructure for running Spark.

4.2 Results

Figure 3 reports the performances of our parallel SubTree Kernel computation versus the sequential algorithm in terms of running time on the generated datasets. Let us mention that the sequential time is obtained on one node (container) of our cluster.

[1] https://hadoop.apache.org/

[2] https://spark.apache.org/

It is obvious to observe that our parallel computation is significantly faster compared to the sequential version across all dataset instances. Furthermore, our analysis reveals that the average absolute acceleration achieved using MapReduce is 50.8 times (Table 2), and the absolute acceleration obtained through Spark is 94.3 times (Table 3). This substantial acceleration is notable, which reflects the effectiveness of our parallel computation approach.

Table 2: Acceleration of Parallel SubTree Kernel computation on different datasets using MapReduce.

	$D1$	$D2$	$D3$	$D4$	**Average**
A^{abs}	57.2	62.9	42.6	40.7	50.8

Table 3: Acceleration of Parallel SubTree Kernel computation on different datasets using Spark.

	$D1$	$D2$	$D3$	$D4$	**Average**
A^{abs}	61.4	95	86.8	134.3	94.3

5 CONCLUSION

The prefix tree automaton constitutes a common base for the computation of different tree kernels: SubTree, RootedTree, and SubSequenceTree kernels (Ouali-Sebti, 2015). In this paper, we have shown a parallel Algorithm that efficiently compute this common structure (RWTA automaton) and we have used it for the computation of the SubTree Kernel using MapReduce and Spark frameworks.

Our parallel implementation of the SubTree kernel computation has been tested on synthetic datasets with different parameters. The results showed that our parallel computation is by far more speed than the sequential version for all instances of datasets. Despite that this work has shown the efficiency of the parallel implementation compared to the sequential algorithms, three main future works are envisaged. Firstly, we have to devise some algorithms that generalise the computation of others kernels such RootedTree, and SubSequenceTree .... Some of them will deploy tree automata intersection in addition to the associated weights computation. In fact, while the subtree kernel is a simple summation of weights, the SubSequenceTree needs more investigation on the weight computations using the resulted RWTAs intersection. Secondly, more large datasets have to be generated and tested to confirm the output-sensitive results of our solutions. Finally, one can investigate different cluster architectures in order to give more insights and recommendations on the cluster' parameters tuning.

REFERENCES

Alian, M. and Awajan, A. (2023). Syntactic-semantic similarity based on dependency tree kernel. *Arabian Journal for Science and Engineering*, pages 1–12.

Chali, Y. and Hasan, S. A. (2015). Towards topic-to-question generation. *Computational Linguistics*, 41(1):1–20.

Collins, M. and Duffy, N. (2001). Convolution kernels for natural language. In Dietterich, T., Becker, S., and Ghahramani, Z., editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press.

Collins, M. and Duffy, N. P. (2002). New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Annual Meeting of the Association for Computational Linguistics*.

Dayalan, M. (2004). Mapreduce: simplified data processing on large clusters. In *CACM*.

Ésik, Z. and Kuich, W. (2002). Formal tree series. *BRICS Report Series*, (21).

Fu, D., Xu, Y., Yu, H., and Yang, B. (2017). Wastk: An weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming*, 2017.

Gordon, M. and Ross-Murphy, S. B. (1975). The structure and properties of molecular trees and networks. *Pure and Applied Chemistry*, 43(1-2):1–26.

Haussler, D. et al. (1999). Convolution kernels on discrete structures. Technical report, Citeseer.

L. Mignot, F. O. and Ziadi, D. (2023). New linear-time algorithm for subtree kernel computation based on root-weighted tree automata.

Maneth, S., Mihaylov, N., and Sakr, S. (2008). Xml tree structure compression. In *2008 19th International Workshop on Database and Expert Systems Applications*, pages 243–247.

Mignot, L., Sebti, N. O., and Ziadi, D. (2015). Root-weighted tree automata and their applications to tree kernels. *CoRR*, abs/1501.03895.

Nasar, Z., Jaffry, S. W., and Malik, M. K. (2021). Named entity recognition and relation extraction: State-of-the-art. *ACM Comput. Surv.*, 54(1).

Ouali-Sebti, N. (2015). Noyaux rationnels et automates d'arbres.

Shatnawi, M. and Belkhouche, B. (2012). Parse trees of arabic sentences using the natural language toolkit.

Thom, J. D. (2018). *Combining tree kernels and text embeddings for plagiarism detection*. PhD thesis, Stellenbosch: Stellenbosch University.

Vishwanathan, S. V. N. and Smola, A. (2002). Fast kernels for string and tree matching. In *NIPS*.

Warikoo, N., Chang, Y.-C., and Hsu, W.-L. (2018). Lptk: a linguistic pattern-aware dependency tree kernel approach for the biocreative vi chemprot task. *Database*, 2018.