Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 382 (2007) 97–108
Theoretical Computer Science
www.elsevier.com/locate/tcs

On coding labeled trees[☆]

Saverio Caminiti[a,*], Irene Finocchi[b], Rossella Petreschi[a]

[a] *Dipartimento di Informatica, Università degli Studi di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy*
[b] *Dipartimento di Informatica, Sistemi e Produzione, Università degli Studi di Roma "Tor Vergata", Via del Politecnico 1, 00133 Roma, Italy*

Abstract

We consider the problem of coding labeled trees by means of strings of node labels. Different codes have been introduced in the literature by Prüfer, Neville, and Deo and Micikevičius. For all of them, we show that both coding and decoding can be reduced to integer (radix) sorting, closing several open problems within a unified framework that can be applied both in a sequential and in a parallel setting. Our sequential coding and decoding schemes require optimal $O(n)$ time when applied to n-node trees, yielding the first linear time decoding algorithm for a code presented by Neville. These schemes can be parallelized on the EREW PRAM model, so as to work in $O(\log n)$ time with cost $O(n)$, $O(n\sqrt{\log n})$, or $O(n \log n)$, depending on the code and on the operation: in all cases, they either match or improve the performances of the best ad hoc approaches known so far.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Graph algorithms; Labeled trees; Prüfer-like codes; Data structures

1. Introduction

Labeled trees are of interest in practical and theoretical areas of computer science. For example, Ethernet has a unique path between terminal devices, thus being a tree: labeling the tree nodes is necessary to identify without ambiguity each device in the network. An interesting alternative to the usual representations of tree data structures in computer memories is based on coding labeled trees by means of strings of node labels. This representation was first used by Prüfer in the proof of Cayley's theorem [2,22] to show a one-to-one correspondence between unrooted labeled trees on n nodes and strings of length $n-2$. In addition to this purely mathematical use, string-based codings of trees have many practical applications. For instance, they make it possible to generate random uniformly distributed trees and random connected graphs [16]: the generation of a random string followed by the use of a fast decoding algorithm is typically more efficient than generating a tree by adding edges randomly, since in the latter case one must

[☆] Work supported in part by the Italian Ministry of University and Scientific Research (project "ALINWEB: Algorithmics for Internet and the Web") and by the University of Rome "La Sapienza" (project "Parallel and Distributed Codes"). A preliminary version of this work appeared in the Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN'04) [S. Caminiti, I. Finocchi, R. Petreschi, A unified approach to coding labeled trees, in: Proceedings of the 6th Latin American Symposium on Theoretical Informatics, in: LNCS, vol. 2976, 2004, pp. 339–348].

[*] Corresponding author. Tel.: +39 06 4991 8514; fax: +39 06 8541 842.
E-mail addresses: caminiti@di.uniroma1.it (S. Caminiti), finocchi@disp.uniroma2.it (I. Finocchi), petreschi@di.uniroma1.it (R. Petreschi).

0304-3975/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2007.03.009

pay attention not to introduce cycles. Furthermore, constraints on the number and on the set of leaves, on the choice of the root, and on the degree of nodes can be easily imposed during the generation. Tree codes are also used for data compression, in the computation of the tree and forest volumes of graphs [15], in genetic algorithms over trees, where chromosomes in the population are represented as strings of integers, and in heuristics for computing minimum spanning trees with additional constraints, e.g., on the number of leaves or on the diameter of the tree [8,9,24].

1.1. Tree codes

We now survey the main tree codes known in the literature, referring the interested reader to the taxonomies in [7,18] for further details. We assume to deal with a rooted n-node tree T whose nodes have distinct labels from $[1, n]$. All the codes that we discuss are obtained by progressively *updating* the tree through the *deletion of leaves*: when a leaf is deleted, the label of its parent is appended to the code.

The oldest and most famous code is due to Prüfer [22]: the leaf deleted at each step is the one with smallest label. In 1953, Neville [20] presented three different codes, the first of which coincides with Prüfer's one. The second Neville code, before updating T, deletes *all* the leaves ordered by increasing labels. The third Neville code works by deleting chains. We call *pending chain* a path $u_1, \ldots, u_k$ of maximal length such that the starting point u_1 is a leaf, and, for each $i \in [1, k-1]$, the deletion of u_i makes u_{i+1} a leaf: the code works by iteratively deleting the pending chain with the smallest starting point. Quite recently, Deo and Micikevičius [6] suggested the following coding approach: at the first iteration, delete all tree leaves as in the second Neville code, then delete the remaining nodes in the order in which they become leaves. For brevity, we will denote the codes introduced above with PR, N2, N3, and DM, respectively. Examples of code computation are shown in Fig. 1.

All these codes have length $n-1$ and the last code element is the root of the tree. If the tree is unrooted, the deletion scheme implicitly defines a root for it. Actually, it is easy to see that the last element in the code is:

(1) the node with maximum label (i.e., n) for Prüfer code;
(2) the center of the tree with maximum label for the second Neville code;
(3) the leaf with maximum label for the third Neville code;
(4) any tree center for Deo and Micikevičius code.

We remark that a center of a graph is a node minimizing the maximum distance from the other nodes; a tree has at most two centers. In cases 1, 3, and 4, the value of the last element can be unequivocally determined from the code and thus the code length can be reduced to $n-2$ [7]. We remark that codes PR and DM have been originally presented for unrooted trees, while Neville's codes have been given for rooted trees and later generalized for unrooted trees by Moon [19]. In view of these considerations, in the rest of this paper we will focus on rooted trees.

1.2. Coding algorithms

Since the introduction of Prüfer code in 1918, a linear time algorithm for its computation was given for the first time only in the late 70's [21], and has been later rediscovered several times [3,10]. The approach that we describe here, according to our knowledge, is due to Kilingsberg (see, e.g., [21], page 271). Let $L[x]$ denote the adjacency list of a node x of a tree T. Let us assume that the degree of each node is known. The Prüfer code of T can be computed as follows:

```
1. for each node $v = 1$ to $n$ do
2.    if $degree[v] = 1$ then
3.       let $u$ be the unique node in $L[v]$
4.       append $u$ to the code and decrease its degree by 1
5.       while ($degree[u] = 1$ and $u < v$) do
6.          let $z$ be the unique node in $L[u]$
7.          append $z$ to the code and decrease its degree by 1
8.          $u \leftarrow z$
```

As for the running time, the dominant operation consists of decreasing the node degree, which is done in total as many times as the number of arcs. With respect to the correctness, indexes v and u are used to scan nodes forward and backward, respectively. The choice of the leaf with smallest label is guaranteed by the while loop: when a node u becomes a leaf, if its label is smaller than those of the leaves existing at that time (see the test $u < v$), u is immediately appended to the code. The third Neville code can be computed in a similar way, by just omitting the test $u < v$. As shown in [6], the code of Deo and Micikevičius requires a quite different approach, but their algorithm maintains

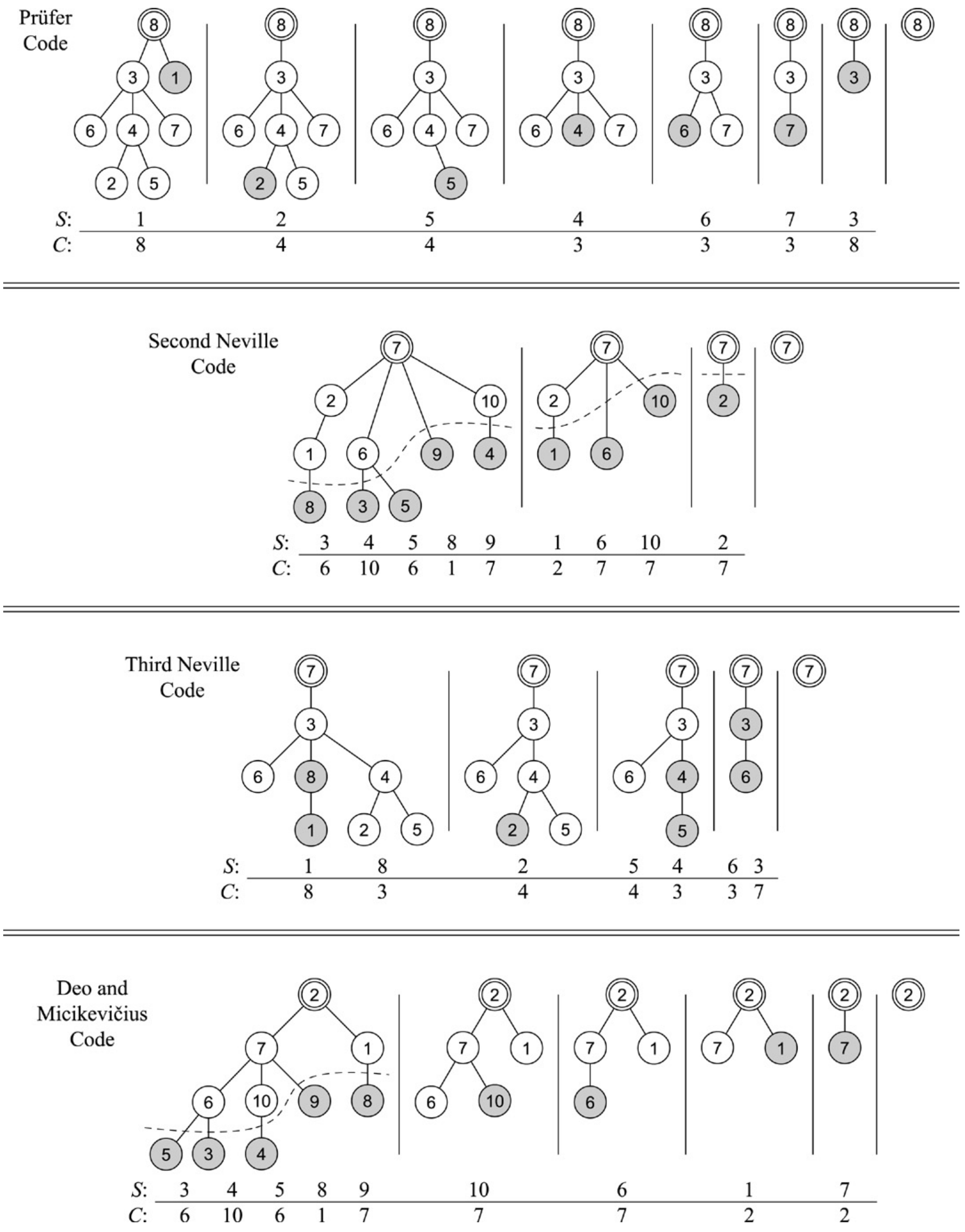


Fig. 1. Step by step computation of different codes. `PR` deletes the leaf with smallest label; `N2` deletes all the leaves ordered by increasing labels; `N3` deletes pending chains; `DM` deletes nodes as they become leaves. S and C denote the deleted leaves and the code, respectively.

linear running time:

```
1. insert the leaves of $T$ in a queue $Q$ in increasing order
2. while ($Q$ is not empty) do
3.    let $v$ be the head of $Q$
4.    let $u$ be the unique node in $L[v]$
5.    append $u$ to the code and decrease its degree by $1$
6.    if ($degree(u) = 1$) then
7.       insert $u$ in $Q$
```

As stated in [7], this algorithm cannot be adapted for computing the second Neville code: indeed, in this case we can not add nodes to the queue as soon as they become leaves, because code N2 requires sorting the leaves before each tree update. A linear time coding algorithm for N2 has been presented only recently in [18].

An optimal parallel algorithm for computing Prüfer codes, which improves over a previous result due to Greenlaw and Petreschi [12], is given in [11]. A few simple changes make the algorithm work also for the third Neville code. Efficient, but not optimal, parallel algorithms for codes N2 and DM are presented in [8].

1.3. Decoding algorithms

A simple – non-optimal – scheme for constructing a tree T from a Prüfer code C is presented in [7]. The algorithm scans the labels in the code from left to right, working as follows:

```
1.  let $L$ be the set of labels not appearing in $C$ (leaves of $T$)
2.  for $i = 1$ to $n - 1$ do
3.      $u = C[i]$
4.      let $v$ be the smallest label in $L$
5.      add edge $(u, v)$ to tree $T$
6.      delete $v$ from $L$
7.      if ($C[i]$ is the rightmost occurrence of $u$ in $C$) then
8.          add $u$ to the label set $L$
```

This scheme can be promptly generalized for building T starting from any of the other codes: the only difference is in the choice of node v in step 4, according to the description of the codes. Different data structures can be used for L in order to implement the different choices of v. A priority queue is well suited for extracting the smallest label node, as required by Prüfer code. Since the second Neville code deletes all the leaves ordered by increasing labels before updating the tree, reconstructing T implies maintaining two disjoint sets of nodes: nodes are extracted from set L while a new set L' is being built; when L is empty, it is assigned with L' and a new round starts. Both L and L' can be implemented as priority queues. The third Neville code and the code by Deo and Micikevičius can be decoded using a stack and a queue, respectively. This scheme implies linear time decoding algorithms only for codes N3 and DM. The use of a priority queue, indeed, yields a running time $O(n \log n)$ for codes PR and N2. Linear time decoding algorithms for code PR are presented in [21,11].

In a parallel setting, Wang, Chen, and Liu [23] propose an $O(\log n)$ time decoding algorithm for Prüfer code using $O(n)$ processors on the EREW PRAM computational model. At the best of our knowledge, parallel decoding algorithms for the other codes were not known in the literature until this paper.

1.4. Results and techniques

We show that both coding and decoding can be reduced to integer (radix) sorting. Based on this reduction, we present a unified approach that works for all the codes introduced so far and can be applied both in a sequential and in a parallel setting. The coding scheme is based on the definition of pairs associated to the nodes of T according to criteria dependent on the specific code: the coding problem is then reduced to the problem of sorting these pairs in lexicographic order. The decoding scheme is based on the computation of the rightmost occurrence of each label in the code: this is also reduced to integer radix sorting.

Concerning coding, our general sequential algorithm requires optimal linear time for all the presented codes. The algorithm can be parallelized, and its parallel version either matches or improves by a factor $O(\sqrt{\log n})$ the performances of the best ad hoc approaches known so far. Concerning decoding, we design the first parallel algorithm for codes N2, N3, and DM: our algorithm works on the EREW PRAM model in $O(\log n)$ time with cost $O(n\sqrt{\log n})$ (with respect to PR, the cost is $O(n \log n)$ and matches the performances of the best previous result). We remark that the problem of finding an optimal sequential decoding algorithm for code N2 was open, and our general scheme solves it. In summary, our results show that labeled trees can be coded and decoded in linear sequential time independently of the specific code. Our parallel results both for coding and for decoding are summarized in Table 1.

Table 1
Summary of our results on the EREW PRAM model. Costs are expressed in terms of number of operations

	Coding		Decoding	
	Before	This paper	Before	This paper
PR	$O(n)$ [11]	$O(n)$	$O(n \log n)$ [23]	$O(n \log n)$
N2	$O(n \log n)$ [8]	$O(n\sqrt{\log n})$	Open	$O(n\sqrt{\log n})$
N3	$O(n)$ [8,11]	$O(n)$	Open	$O(n\sqrt{\log n})$
DM	$O(n \log n)$ [8]	$O(n\sqrt{\log n})$	Open	$O(n\sqrt{\log n})$

2. A unified coding algorithm

Many sequential and parallel coding algorithms have been presented in the literature [3,6,7,11,12,23], but all of them strongly depend on the properties of the code which has to be computed and thus are very different from each other. In this section we show a unified approach that works for all the codes introduced in Section 1.1 and can be used both in a sequential and in a parallel setting. Namely, we associate each tree node with a pair of integer numbers and we sort nodes using such pairs as keys. The obtained ordering corresponds to the order in which nodes are deleted from the tree and can thus be used to compute the code. In the rest of this section we show how different pair choices yield Prüfer, Neville, and Deo and Micikevičius codes, respectively. We then present a linear time sequential coding algorithm and its parallelization on the EREW PRAM model. The parallel algorithm works in $O(\log n)$ time and requires either $O(n)$ or $O(n\sqrt{\log n})$ operations, depending on the code.

2.1. Coding by sorting pairs

Let T be a rooted labeled n-node tree. If T is unrooted, we assume the root r to be chosen as in points (1)–(4) in Section 1.1. Let u, v be any two nodes of tree T. Let us call:

- T_v, the subtree of T rooted at v;
- $d(u, v)$, the distance between nodes u and v (we assume that $d(v, v) = 0$);
- $l(v)$, the (bottom-up) level of node v, i.e., the maximum distance of v from a leaf in T_v;
- $\mu(v)$, the maximum label among all nodes in T_v;
- $\lambda(v)$, the maximum label among all leaves in T_v;
- $\gamma(v)$, the maximum label among the leaves in T_v that have maximum distance from v;
- (x_v, y_v), a pair associated to node v according to the specific code as shown in Table 2;
- P, the set of pairs (x_v, y_v) for each v in T.

The following lemma establishes a correspondence between the set P of pairs and the order in which nodes are deleted from the tree.

Lemma 1. *For each code, the lexicographic ordering of the pairs (x_v, y_v) in set P corresponds to the order in which nodes are deleted from tree T according to the code definition.*

Proof. We discuss each code separately:

PR: before selecting a node v, the entire subtree T_v has been deleted. Furthermore, according to the definition of Prüfer code, when the node $\mu(v)$ is chosen for deletion, the only remaining subtree of T_v consists of a *chain* from $\mu(v)$ to v. All the nodes in such a chain have label smaller than $\mu(v)$ and thus will be chosen in the steps immediately following the deletion of $\mu(v)$. The tree is therefore partitioned into chains containing nodes with the same value of $\mu(v)$ and the rank of each node v in the chain is $d(v, \mu(v))$. Prüfer code deletes all the chains, in increasing order, with respect to $\mu(v)$.

N2: the code deletes at each iteration all the leaves of T, and thus nodes are deleted starting from smaller to higher levels. Nodes within the same level are deleted by increasing label. Hence the pair choice.

Table 2
Pair (x_v, y_v) associated to node v for the different codes

Code	$(x_v\,,\ y_v)$
PR	$(\mu(v)\,,\ d(\mu(v), v))$
N2	$(l(v)\,,\ v)$
N3	$(\lambda(v)\,,\ d(\lambda(v), v))$
DM	$(l(v)\,,\ \gamma(v))$

N3: it is sufficient to use the definition of *pending chain* given in Section 1.1 and to observe that, for each node v, $\lambda(v)$ is the head of the unique pending chain containing v.

DM: similarly to code N2, code DM deletes nodes from smaller to higher levels. As proved in [8], nodes within the same level ℓ are deleted in increasing order of their γ values. The proof given by Deo and Micikevičius is by induction on ℓ.
Nodes within level 0 (i.e., the leaves of T) are such that $\gamma(v) = v$ and are deleted by increasing label order. Let u and v be two arbitrary nodes at level ℓ. According to the code definition, the order in which u and v become leaves is strictly related to the deletion order of nodes at level $\ell - 1$. Let u' and v' be the last deleted nodes of T_u and T_v respectively. It is easy to see that $l(u') = l(v') = \ell - 1$. Furthermore, by definition of γ, it holds $\gamma(u') = \gamma(u)$ and $\gamma(v') = \gamma(v)$. Since by inductive hypothesis u' is deleted before v' if and only if $\gamma(u') < \gamma(v')$, the same holds for nodes u and v. □

Before describing the sequential and parallel algorithms, note that it is easy to sort the pairs (x_v, y_v) used in the coding scheme. Indeed, independently of the code, each element in such pairs is in the range $[1, n]$. A radix-sort-like approach [5] is thus sufficient to sort them according to y_v, first, and x_v, later. In Fig. 2 the pairs relative to the four codes are presented. The tree used in the example is the same in the four cases and is rooted according to points (1)–(4) in Section 1.1. Bold arcs in the trees related to codes PR and N3 indicate chains and pending chains, respectively; dashed lines in the trees related to codes N2 and DM separate nodes at different levels. In each figure the string representing the generated code, the pairs sorted in increasing order, and the node corresponding to each pair are also shown.

2.2. Sequential algorithm

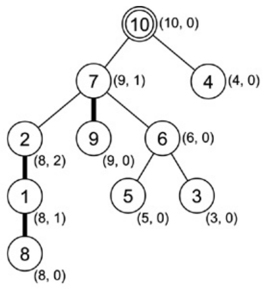
Our sequential coding algorithm works on rooted trees and hinges upon the pairs defined in Section 2.1:

UNIFIED CODING ALGORITHM:
```
1.  for each node $v$, compute the pair $(x_v, y_v)$ according to Table 2
2.  sort the tree nodes according to pairs $(x_v, y_v)$
3.  for $i = 1$ to $n - 1$ do
4.      let $v$ be the $i$-th node in the ordering
5.      append $parent(v)$ to the code
```

Theorem 1. *Let T be a n-node tree and let the pair (x_v, y_v) associated to each node v of T be defined as in Table 2. The* UNIFIED CODING ALGORITHM *computes codes* PR, N2, N3, *and* DM *in $O(n)$ running time.*

Proof. The correctness of the UNIFIED CODING ALGORITHM follows from Lemma 1. The set of pairs can be easily computed in $O(n)$ time using a post-order traversal of the tree, and two counting-sorts can be used to implement step 2. Hence the linear running time. □

We remark that the UNIFIED CODING ALGORITHM works on rooted trees and generates strings of length $n - 1$. As observed in Section 1.1, the codes can be also defined for unrooted trees. In this case, if we root the tree at a node r chosen according to points (1)–(4) (see Section 1.1), our algorithm returns the concatenation of C and r, where C is the string of length $n - 2$ associated to the unrooted tree by code definition.

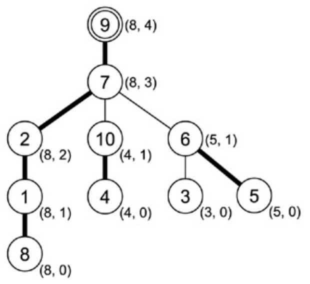


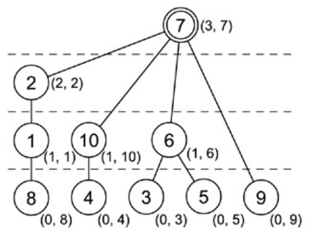
Prüfer code

Pairs:	(3,0)	(4,0)	(5,0)	(6,0)	(8,0)	(8,1)	(8,2)	(9,0)	(9,1)
Node:	3	4	5	6	8	1	2	9	7
Code:	6	10	6	7	1	2	7	7	10

Third Neville code

Pairs:	(3,0)	(4,0)	(4,1)	(5,0)	(5,1)	(8,0)	(8,1)	(8,2)	(8,3)
Node:	3	4	10	5	6	8	1	2	7
Code:	6	10	7	6	7	1	2	7	9





Second Neville Code

Pairs:	(0,3)	(0,4)	(0,5)	(0,8)	(0,9)	(1,1)	(1,6)	(1,10)	(2,2)
Node:	3	4	5	8	9	1	6	10	2
Code:	6	10	6	1	7	2	7	7	7

Deo and Micikevičius code

Pairs:	(0,3)	(0,4)	(0,5)	(0,8)	(0,9)	(1,4)	(1,5)	(1,8)	(2,9)
Node:	3	4	5	8	9	10	6	1	7
Code:	6	10	6	1	7	7	7	2	2

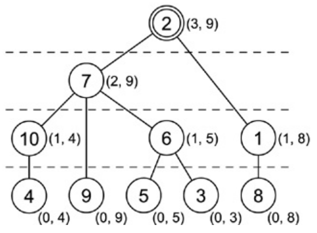


Fig. 2. Pair associated to each tree node as specified in Table 2.

2.3. *Parallel algorithm*

We now show how to parallelize each step of the sequential algorithm presented in Section 2.2. We work in the simplest PRAM model with exclusive read and write operations (EREW [14]). If the tree is unrooted, the Euler tour technique makes it possible to root it at the node r specified in Section 1.1 in $O(\log n)$ time with cost $O(n)$ [14]. The node r can be easily identified; in particular, we refer to the approach described in [17] for computing a center of the tree. Before analyzing the UNIFIED CODING ALGORITHM on the EREW PRAM, we discuss how to compute the pair components in parallel.

Lemma 2. *The pairs given in Table 2 can be computed on the EREW PRAM model in $O(\log n)$ time with cost $O(n)$.*

Proof. We discuss separately the components of each pair.

$\mu(v)$: the maximum node in each subtree can be computed in $O(\log n)$ time with cost $O(n)$ using the Rake technique [14]. In order to avoid concurrent reading during the Rake operation, the tree T must be preliminarily transformed into a binary tree T_R as follows: for each node v with $k > 2$ children, v is replaced by a complete binary tree of height $\lceil \log k \rceil$ having v as root and v's children as the k leftmost leaves. This transformation can be also done in $O(\log n)$ time with cost $O(n)$ [11].

$d(\mu(v), v)$: we partition T into chains by labeling each node v with the value $\mu(v)$ and by deleting edges between nodes with different labels. Now, the rank of node v in its chain is exactly $d(\mu(v), v)$. In order to compute the chains, each node links itself to its parent if $\mu(v) = \mu(\mathit{parent}(v))$. A list ranking then gives the position of each node in its chain in $O(\log n)$ time with cost $O(n)$ [14]. The use of the binary tree T_R guarantees that no concurrent read is necessary for accessing $\mu(\mathit{parent}(v))$.

$l(v)$: an Euler tour gives the distance $d(v, r)$ of each node v from the root r of tree T. Then, $l(v) = d(f, r) - d(v, r)$, where f is a leaf of T_v at maximum distance from r. We remark that f can be easily computed using the Rake technique [14].

$\lambda(v)$: the same techniques used for computing $\mu(v)$ can be adapted to obtain the maximum leaf of each subtree with the same performances.

$d(\lambda(v), v)$: analogous considerations as for computing $d(\mu(v), v)$ hold.

$\gamma(v)$: given the distance of each node from the root, $\gamma(v)$ is the node $u \in T_v$ such that $(d(u, r), u)$ is maximum and can be computed with the Rake technique. □

The following theorem summarizes the performances of the UNIFIED CODING ALGORITHM in a parallel setting.

Theorem 2. *Let T be a n-node tree and let the pair (x_v, y_v) associated to each node v of T be defined as in Table 2. On the EREW PRAM model, the* UNIFIED CODING ALGORITHM *computes codes* PR *and* N3 *optimally, i.e., in $O(\log n)$ time with cost $O(n)$, and codes* N2 *and* DM *in $O(\log n)$ time with cost $O(n\sqrt{\log n})$.*

Proof. By Lemma 2, step 1 of the UNIFIED CODING ALGORITHM requires $O(\log n)$ time with cost $O(n)$. Step 3 can be trivially implemented in $O(1)$ time with cost $O(n)$. The sorting in step 2 is thus the most expensive operation. Following a radix-sort-like approach and using the stable integer-sorting algorithm presented in [13] as a subroutine, step 2 would require $O(\log n)$ time with cost $O(n\sqrt{\log n})$ on an EREW PRAM.[1] This gives the stated running time and cost for codes N2 and DM. For codes PR and N3, we can further reduce the cost of our algorithm to $O(n)$ by using an ad hoc sorting procedure that benefits from the partition into chains.

Let us consider Prüfer code first. As observed in [12], the final node ordering can be obtained by sorting chains among each other and nodes within each chain. In our framework, the chain ordering is given by the value $\mu(v)$, and the position of each node within its chain by the distance $d(\mu(v), v)$. Instead of using a black-box integer sorting procedure, we exploit the fact that we can compute optimally the size of each chain, i.e., the number of nodes with the same $\mu(v)$, by means of prefix sums. Another prefix sum computation can then be used to obtain, for each chain head, the number of nodes in the preceding chains, i.e., its final position. At last, the position of the remaining nodes is unequivocally determined by summing up the position of the chain head $\mu(v)$ with the value $d(\mu(v), v)$. Similar considerations can be applied to the third Neville code. □

We remark that our algorithm solves within a unified framework the parallel coding problem. With respect to codes N3 and PR, it matches the performances of the (optimal) algorithms known so far [8,11]. With respect to codes N2 and DM, it improves of an $O(\sqrt{\log n})$ factor over the best approaches known in the literature [8].

3. Decoding algorithms

In this section we present sequential and parallel algorithms for decoding, i.e., for building the tree T corresponding to a given code C. As far as C is computed, each node label in it represents the parent of a leaf deleted from T. Hence, in order to reconstruct T, it is sufficient to compute the ordered sequence of labels of the deleted leaves, say S: for each $i \in [1, n-1]$, the pair $(C[i], S[i])$ will thus be an arc in the tree. Before describing the algorithms, we argue that computing the rightmost occurrence of a node in the code is very useful for decoding, and we show how to obtain such an information both in a sequential and in a parallel setting.

[1] The result on parallel integer sorting [13] holds when the machine word length is $O(\log n)$. Under the more restrictive hypothesis that the word length is $O(\log^2 n)$, the cost of sorting can be reduced to $O(n)$, and so does the cost of our coding algorithm.

Table 3
Condition on node v that is checked in the UNIFIED DECODING ALGORITHM and position of v as a function of $rightmost(v, C)$

	$test(v)$	$position(v)$
PR	$rightmost(v, C) > prev(v, C)$	$rightmost(v, C) + 1$
N3	$rightmost(v, C) > 0$	$rightmost(v, C) + 1$
DM	$rightmost(v, C) > 0$	$\|leaves(T)\| + \sigma(rightmost(v, C))$

3.1. Decoding by rightmost occurrence computation

We first observe that the leaves of T are exactly those nodes that do not appear in the code, as they are not parents of any node. Each internal node, say v, in general may appear in C more than once; each appearance corresponds to the deletion of one of its children, and therefore to decreasing the degree of v by 1. After the rightmost occurrence in the code, v is clearly a leaf and thus becomes a candidate for being deleted. More formally:

$$\forall\, v \neq r, \quad \exists \text{ unique } j > rightmost(v, C) \text{ such that } S[j] = v$$

where r is the tree root (i.e., the last element in C) and $rightmost(v, C)$ denotes the index of the rightmost occurrence of node v in C. We assume that $rightmost(v, C) = 0$ if v does not appear in C. It is easy to compute the rightmost occurrence of each node sequentially by simply scanning C. The following lemma analyzes the rightmost computation in parallel.

Lemma 3. *The rightmost occurrences of nodes in a code C of length $n - 1$ can be computed in $O(\log n)$ time with cost $O(n\sqrt{\log n})$ on the EREW PRAM model.*

Proof. We reduce the rightmost occurrence computation to a pair sorting problem: we sort in increasing order the pairs $(C[i], i)$, for $i \in [1, n-1]$. Indeed, in each sub-sequence of pairs with the same first element $C[i]$, the second element of the last pair is the index of the rightmost occurrence of node $C[i]$ in the code. Since each pair value is an integer in $[1, n]$, we can use twice the stable integer-sorting algorithm of [13]: this requires $O(\log n)$ time and $O(n\sqrt{\log n})$ cost in the EREW PRAM model. Then, each processor p_i in parallel compares the first element of the i-th pair in the sorted sequence to the first element of the $(i + 1)$-th pair, deciding whether this is the end of a sub-sequence. This requires additional $O(1)$ time and linear cost with exclusive read and write operations. □

3.2. A unified decoding algorithm

In this section we describe a decoding algorithm for codes PR, N3, and DM that is based on the rightmost occurrences and can be used both in a sequential and in a parallel setting. Differently from the other codes, in code N2 the rightmost occurrence of each node in C gives only partial information about sequence S. Thus, we treat N2 separately in Section 3.3. We need the following notation. For each $i \in [1, n - 1]$, let $\rho(i)$ be 1 if i is the rightmost occurrence of node $C[i]$, and 0 otherwise. Let $\sigma(i)$ be the number of internal nodes whose rightmost occurrence is at most i, i.e.

$$\sigma(i) = \sum_{j \le i} \rho(j).$$

Similarly to [23], let $prev(v, C)$ denote the number of nodes with label smaller than v that become leaves before v, i.e.

$$prev(v, C) = |\{u \text{ s.t. } u < v \text{ and } rightmost(u, C) < rightmost(v, C)\}|.$$

The following lemma shows, for each code, how the position of a node in the sequence S that we want to construct can be expressed as a function of *rightmost*.

Lemma 4. *Let C be a string of $n - 1$ integers in $[1, n]$. For each of the codes* PR, N3, *and* DM, *let* test *and* position *be defined as in Table 3. Let S be the sequence of leaves deleted from the tree while building the code. The proper position in S of any node v that satisfies* test(v) *is given by* position(v).

	1	2	3	4	5	6	7	8	9
C:	6	10	6	7	1	2	7	7	10

	1	2	3	4	5	6	7	8	9	10
rightmost:	5	6	0	0	0	3	8	0	0	9
prev:	0	1	0	0	0	3	6	0	0	9
test:	t	t	f	f	f	f	t	f	f	f

	1	2	3	4	5	6	7	8	9
S (after line 3):	-	-	-	-	-	1	2	-	7
S (after line 7):	3	4	5	6	8	1	2	9	7

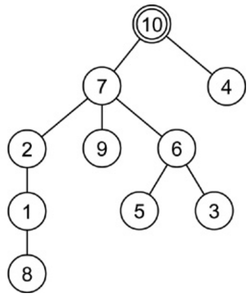


Fig. 3. An example of execution of the UNIFIED DECODING ALGORITHM in the case of Prüfer code: content of the main data structures and output tree.

Proof. We discuss each code separately, starting from the simplest one.

N3: each internal node v is deleted as soon as it becomes a leaf. Thus, the position of v in sequence S is exactly $rightmost(v, C) + 1$.

PR: differently from code N3, in code PR an internal node v is deleted as soon as it becomes a leaf if and only if there is no leaf with label smaller than v. In order to test this condition, following [23], we use information given by $prev(v, C)$: the position of v in S is $rightmost(v, C) + 1$ if and only if $rightmost(v, C) \geq prev(v, C)$.

DM: by definition of code DM, all the leaves of T, sorted by increasing labels, are at the beginning of sequence S. Then, all the internal nodes appear in the order in which they become leaves, i.e., sorted by increasing *rightmost*. Thus, the position of node v is given by $|leaves(T)| + \sigma(rightmost(v, C))$. □

We remark that some entries of S may be still empty after positioning nodes according to Lemma 4. Using the codes' definitions, all the nodes not positioned by Lemma 4, except for the root, can be assigned to the empty entries of S by increasing label order. In particular, for codes N3 and DM, only the leaves of T are not positioned and, in the case of DM, all of them will appear at the beginning of S. We are now ready to describe our unified decoding algorithm:

```
UNIFIED DECODING ALGORITHM:
1.  for each node $v$ compute $rightmost(v, C)$
2.  for each node $v$ except for the root do
3.      if ($test(v)$ = true) then $S[position(v)] \leftarrow v$
4.  let $L$ be the list of (non-root) nodes not yet assigned to a
          position and considered in increasing order
5.  let $P$ be the set of positions of $S$ which are still empty
6.  for each $i = 1$ to $|L|$ do
7.      $S[P[i]] \leftarrow L[i]$
```

where $test(v)$ and $position(v)$ are specified in Table 3. An example of execution of the UNIFIED DECODING ALGORITHM in the case of Prüfer code is shown in Fig. 3.

As observed in Section 1.3, a linear sequential decoding algorithm for Prüfer code is presented in [11], while the straightforward sequential implementation of our algorithm would require $O(n \log n)$ time due to the computation of *prev*. This can be reduced to $O(n)$ time by adapting the UNIFIED DECODING ALGORITHM in such a way that the *prev* computation can be avoided. Namely, lines 2–3 can be omitted (considering the $test(v)$ as false for each node v), and lines 6–7 can be replaced as follows:

```
6.  for each $i = 1$ to $|L|$ do
7.      $position \leftarrow \max\{first\_empty\_pos(S), rightmost(L[i], C) + 1\}$
8.      $S[position] \leftarrow L[i]$
```

where $first\_empty\_pos(S)$ returns the smallest empty position in S. In this implementation, nodes are considered in increasing label order: node v is assigned to position $rightmost(v) + 1$ of S if this position is still empty, and to the leftmost empty position otherwise. In order to see that this is equivalent to the UNIFIED DECODING ALGORITHM, observe that nodes for which $rightmost(v) > prev(v)$ (see the test in line 3) will always find the position $rightmost(v) + 1$ empty, due to the definition of *prev*. Hence, they will be inserted in S exactly as in line 3 of the UNIFIED DECODING ALGORITHM.

The performances of the UNIFIED DECODING ALGORITHM in sequential and parallel models of computation are described by the following theorem.

Theorem 3. *Let C be a string of $n-1$ integers in $[1, n]$. For each of the codes* `PR`*,* `N3`*, and* `DM`*, let* test *and* position *be defined as in Table 3. The* UNIFIED DECODING ALGORITHM *computes the tree corresponding to string C in $O(n)$ sequential time. On the EREW PRAM model, it requires $O(\log n)$ time with cost $O(n \log n)$ for code* `PR` *and $O(n\sqrt{\log n})$ for codes* `N3` *and* `DM`*.*

Proof. The correctness of the UNIFIED DECODING ALGORITHM follows from Lemma 4. The sequential running time can be easily proved in view of the previous considerations. With respect to Prüfer code, the parallel version of the UNIFIED DECODING ALGORITHM yields essentially the same algorithm described in [23]. Its bottleneck is the *prev* computation that can be reduced to a dominance counting problem and can be solved on the EREW PRAM in $O(\log n)$ time with cost $O(n \log n)$ [1,4]: we refer to [23,11] for a detailed analysis. For the other codes, $\sigma(i)$ can be computed for each i using a prefix sum operation [14]. In order to get set L in step 4, we can mark each node not yet assigned to S and obtain its rank in L by computing prefix sums. Similarly for set P. Hence, the most expensive step is the rightmost computation, which requires integer sorting as in Lemma 3. □

3.3. Second Neville code

We first observe that if all nodes were assigned with a level, an ordering with respect to pairs $(l(v), v)$ would give sequence S, and thus the tree related to code `N2`. We refer to Section 2.1 for details on the correctness of this approach. We now show how to compute $l(v)$.

Let x be the number of leaves of T: these nodes have both level and rightmost 0. Consider the first x elements of code C, say $C[1], \dots, C[x]$. For each i, $1 \le i \le x$, such that i is the rightmost occurrence of $C[i]$, we know that node $C[i]$ has level 1. The same reasoning can be applied to get level-2 nodes from level-1 nodes, and so on. With respect to the running time, a sequential scan of code C is sufficient to compute the level of each node in linear time. Unfortunately, this approach is inherently sequential and thus inefficient in parallel. We now discuss an alternative approach for computing $l(v)$ that can be easily parallelized.

Lemma 5. *Let C be a string of $n-1$ integers in $[1, n]$ representing the code* `N2` *associated to a tree T. The level of each node in T can be computed from C on the EREW PRAM model in $O(\log n)$ time with cost $O(n\sqrt{\log n})$.*

Proof. Let T' be the tree obtained by interpreting C as the code by Deo and Micikevičius and let S' be the corresponding sequence: although T and T' are different, they have the same nodes at the same levels since $x_v = l(v)$ as shown in Table 2. Thus, both in S and S', nodes at level $i+1$ appear after nodes at level i, but are differently permuted within the level. We use T' to get missing information: after building T' using the UNIFIED DECODING ALGORITHM, we compute node levels applying the Euler tour technique. We remark that the Euler tour technique requires a particular data structure [14] that can be built as described in [12]. The bottleneck of this procedure is sorting of pairs of integers in $[1, n]$, and thus we can use the parallel integer sorting presented in [13]. □

Given level information, the sequence S corresponding to tree T can be easily obtained by sorting the pairs $(l(v), v)$. We can summarize the results concerning code `N2` as follows:

Theorem 4. *Let C be a string of $n-1$ integers in $[1, n]$. The tree corresponding to C according to code* `N2` *can be computed in $O(n)$ sequential time and in $O(\log n)$ time with cost $O(n\sqrt{\log n})$ on the EREW PRAM model.*

Proof. The correctness follows from the definition of code `N2` and from Lemma 1. The running time is guaranteed by Lemma 5 and by the bounds on integer sorting [13]. □

4. Concluding remarks

In this paper we have presented a unified approach for coding labeled trees by means of strings of node labels and have applied it to four well-known codes due to Prüfer [22], Neville [20], and Deo and Micikevičius [6]. The coding scheme is based on the definition of pairs associated to the nodes of the tree according to criteria dependent on the specific code: the coding problem is then reduced to the problem of sorting these pairs in lexicographic order.

The decoding scheme is based on the computation of the rightmost occurrence of each label in the code: this is also reduced to pair sorting.

We have applied these approaches both in a sequential and in a parallel setting. Namely, we have presented the first optimal sequential decoding algorithm for code N2: the sequential coding and decoding problem is thus completely closed, as we have shown that both operations in all the four codes can be done in linear time.

The parallel version of our coding algorithm works on the EREW PRAM model in $O(\log n)$ time and either $O(n)$ or $O(n\sqrt{\log n})$ operations, depending on the code: it is therefore optimal for codes PR and N3, while it improves by a factor $O(\sqrt{\log n})$ the performances of the best ad hoc algorithms for codes N2 and DM.

With respect to parallel decoding, our general scheme yields the first algorithms for N2, N3, and DM and matches the performances of the best known algorithm for PR. Namely, all the codes, except for PR, can be decoded in $O(\log n)$ time and $O(n\sqrt{\log n})$ operations, since integer sorting is the most costly operation: any improvement on the computation of integer sorting would thus yield better results for our parallel algorithms. As discussed in [11], a dominance counting problem is instead the bottleneck of the decoding algorithm for PR: in [1] a lower bound $\Omega(n \log n)$ has been shown for the dominance counting problem. At the best of our knowledge, it is an open question to understand if it is possible to improve this result when the input is limited to integer values in a small range or to avoid the *prev* computation in the decoding algorithm for PR.

References

[1] M.J. Atallah, R. Cole, M.T. Goodrich, Cascading divide-and-conquer: A technique for designing parallel algorithms, SIAM Journal of Computing 18 (3) (1989) 499–532.

[2] A. Cayley, A theorem on trees, Quarterly Journal of Mathematics 23 (1889) 376–378.

[3] H.C. Chen, Y.L. Wang, An efficient algorithm for generating Prüfer codes from labelled trees, Theory of Computing Systems 33 (2000) 97–105.

[4] R. Cole, Parallel merge sort, in: Proceedings of 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 511–516.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, McGraw-Hill, 2001.

[6] N. Deo, P. Micikevičius, A new encoding for labeled trees employing a stack and a queue, Bulletin of the Institute of Combinatorics and its Applications 34 (2002) 77–85.

[7] N. Deo, P. Micikevičius, Prüfer-like codes for labeled trees, Congressus Numerantium 151 (2001) 65–73.

[8] N. Deo, P. Micikevičius, Parallel algorithms for computing Prüfer-like codes of labeled trees, Computer Science Technical Report CS-TR-01-06, 2001.

[9] W. Edelson, M.L. Gargano, Feasible encodings for GA solutions of constrained minimal spanning tree problems, in: Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann Publishers, 2000, p. 754.

[10] W. Edelson, M.L. Gargano, Modified Prüfer code: $O(n)$ implementation, Graph Theory Notes of New York 40 (2001) 37–39.

[11] R. Greenlaw, M.M. Halldorsson, R. Petreschi, On computing Prüfer codes and their corresponding trees optimally, in: Proceedings Journees de l'Informatique Messine, Graph algorithms, 2000.

[12] R. Greenlaw, R. Petreschi, Computing Prüfer codes efficiently in parallel, Discrete Applied Mathematics 102 (2000) 205–222.

[13] Y. Han, X. Shen, Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMS, SIAM Journal of Computing 31 (6) (2002) 1852–1878.

[14] J. Jájá, An Introduction to parallel algorithms, Addison-Wesley, 1992.

[15] A. Kelmans, I. Pak, A. Postnikov, Tree and forest volumes of graphs, DIMACS Technical Report 2000-03, 2000.

[16] V. Kumar, N. Deo, N. Kumar, Parallel generation of random trees and connected graphs, Congressus Numerantium 130 (1998) 7–18.

[17] W.T. Lo, S. Peng, The optimal location of a structured facility in a tree network, Journal of Parallel Algorithms and Applications 2 (1994) 43–60.

[18] P. Micikevičius, Parallel graph algorithms for molecular conformation and tree codes, Ph.D. Thesis, University of Central Florida, Orlando, Florida, 2002.

[19] J.W. Moon, Counting Labeled Trees, William Clowes and Sons, London, 1970.

[20] E.H. Neville, The codifying of tree structures, Proceedings of Cambridge Philosophical Society 49 (1953) 381–385.

[21] A. Nijenhuis, H.S. Wilf, Combinatorial Algorithms, Academic Press, New York, 1978.

[22] H. Prüfer, Neuer Beweis eines Satzes über Permutationen, Archiv für Mathematik und Physik 27 (1918) 142–144.

[23] Y.L. Wang, H.C. Chen, W.K. Liu, A parallel algorithm for constructing a labeled tree, IEEE Transactions on Parallel and Distributed Systems 8 (12) (1997) 1236–1240.

[24] G. Zhou, M. Gen, A note on genetic algorithms for degree-constrained spanning tree problems, Networks 30 (1997) 91–95.