Sequential and parallel algorithms for the NCA problem on pure pointer machines

A. Dal Palú\textsuperscript{a}, E. Pontelli\textsuperscript{b,}\textsuperscript{*}, D. Ranjan\textsuperscript{b}

\textsuperscript{a}Dipartimento di Matematica e Informatica, Università degli Studi di Udine, Udine, Italy
\textsuperscript{b}Department of Computer Science, New Mexico State University, Box 30001/CS, Las Cruces, NM 88003, USA

Received 23 December 2003; received in revised form 9 June 2005; accepted 7 October 2005

Communicated by P. Spirakis

Abstract

We present a simple, arithmetic-free, efficient scheme to compress trees maintaining the NCA information. We use this compression scheme to provide an $O(n + q \log \log n)$ solution for solving the NCA problem on Pure Pointer Machines (PPMs)—i.e., pointer machines with no arithmetic capabilities—in both the static and dynamic case, where $n$ is the number of add-leaf/delete operations and $q$ is the number of NCA queries. This solution is optimal. We also extend the solution to a parallel pointer machine algorithm. The algorithm assumes that the tree $T$ is known in advance and it requires $O(\log n)$ parallel time and $O(n)$ processors for pre-processing where $n$ is the number of nodes in the tree. Thereafter, it can answer any NCA query in $O(\log \log n)$ time using a single processor. To our knowledge, this is the best known parallel pointer machine algorithm for the NCA problem. Our NCA algorithm requires an efficient parallel solution of the temporal precedence problem [Ranjan et al., The temporal precedence problem, Algorithmica 28 (2000) 288–306]. We provide an efficient parallel pointer machine algorithm to solve this problem as well.

© 2005 Elsevier B.V. All rights reserved.

Keywords: Nearest common ancestor; Dynamic data structures; Pointer machines

1. Introduction

The Nearest Common Ancestor (NCA) Problem can be broadly defined as follows: Given a tree $T$ and two nodes $x, y \in T$, find the nearest common ancestor of $x$ and $y$ in $T$. In the static version of the problem, $T$ is known in advance. In the dynamic version $T$ is modified via some pre-defined operations. The difficulty of the problem depends on what kind of operations are allowed for tree modification. Some typical operations considered in this context are add-leaf that allows addition of leaves to the tree, delete which allows deletion of a node, link which allows linking of a tree as a subtree of a node in another tree, etc. In the offline version, both $T$ and the NCA queries are known in advance.

* Corresponding author. Tel.: +1 5056466239; fax: +1 5056461002.
E-mail addresses: dalpalu@dimi.uniud.it (A. Dal Palú), epontell@cs.nmsu.edu (E. Pontelli), dranjan@cs.nmsu.edu (D. Ranjan).

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
The NCA problem has been studied extensively. For the static case, the original work by Harel and Tarjan [15] provides a constant-time algorithm for performing the \( \text{nca}(x, y) \) operation after a linear-time pre-processing of the tree. This result was later simplified and parallelized by Schieber and Vishkin [22] and Gusfield [14]. Bender and Farach-Colton [4] provided an effectively implementable algorithm which provides \( O(1) \) time execution of \( \text{nca}(x, y) \) operation with linear-time pre-processing of the tree. In all these algorithms, complexity analysis is done assuming the RAM model.

For the dynamic case, Tsakalidis [25] provides algorithms with \( O(\log h) \) worst-case time for the \( \text{nca} \) operation and almost amortized \( O(1) \) time for add-leaf and delete in a dynamic tree, where \( h \) is the height of the tree. The algorithm is developed for an Arithmetic Pointer Machine (APM) model under the uniform cost measure assumption (constant time arithmetic for \( \Theta(\log n) \)-size integers). This result on APMs has been recently improved in Alstrup and Thorup [1], where it is shown that the NCA problem can be solved in worst-case \( O(\log \log n) \) time per operation, and that it can be solved in \( O(n + q \log \log n) \) time on APMs, where \( n \) is the number of link operations and \( q \) is the number of NCA queries. The work by Cole and Hariharan [7] provides the ability to insert (leaves and internal nodes) and delete nodes (leaves and internal nodes with one child) in a tree, and execute the \( \text{nca}(x, y) \) operation in worst-case constant time. Both methods make use of arithmetic capabilities of the respective machine models. For the offline version, a linear-time algorithm on APMs was given by Buchsbaum et al. [6].

The best known parallel NCA algorithms on PRAMs (e.g., the algorithm by Schieber and Vishkin, [22]) require \( O(\log n) \) parallel time for pre-processing using \( O(n/\log n) \) processors, and answers the NCA queries in \( O(1) \) time. Berkman and Vishkin [5] also show that if an Euler tour of the tree and the levels of all nodes are known in advance, then the NCA problem can be solved with \( I_m(n) \) parallel pre-processing using an optimal number of processors, and \( O(m) \) query time, where \( I_m(n) = \alpha(m, n) = \min\{i \mid A(m, i) \geq n\} \) and \( A \) is the Ackermann’s function. If this extra information is not known in advance, then the parallel pre-processing time is increased to \( O(\log n) \) and each query can be processed in \( O(1) \) time. We are not aware of any parallel pointer machine algorithm for the NCA problem. It is possible to translate the known PRAM algorithms to Parallel pointer machine algorithms, although it is not clear how one will avoid a penalty of at least factor of \( \log n \) in this translation (recall that PRAMs allow constant-time arithmetic on numbers of size up to \( \log n \) and indexing into arrays using integer addresses, unlike pointer machines).

In this work we focus on solving the problem on Pure Pointer Machines (PPMs), i.e., pointer machines that do not allow constant-time arithmetic operations. We present a simple, arithmetic-free, efficient scheme to compress trees maintaining the NCA information. This compression scheme is different from the ones previously used in literature [15,7]. In particular, it does not make any use of arithmetic and it is very local in nature and hence seems eminently parallelizable. We use this compression scheme to provide an \( O(n + q \log \log n) \) solution for solving the NCA problem on PPMs in both the static and the dynamic case, where \( n \) is the number of add-leaf/delete operations and \( q \) is the number of queries. This solution is optimal because of a known matching lower bound [15]. Moreover, it has the same complexity as that of an optimal solution on APMs. Hence, our result shows that use of arithmetic is not essential for doing NCA calculations optimally. This is intellectually satisfying because, intuitively, NCA is a structural and not an arithmetic problem. The result is also interesting because it shows that, for the NCA problem, it is possible to totally avoid the poly-log penalty that one has to incur in a generic translation of an algorithm designed for pointer machines with arithmetic to PPMs.

The solution can be extended to provide an efficient parallel pointer machine algorithm for the NCA problem for trees in the static case. The algorithm assumes that the tree \( T \) is known in advance. It requires \( O(\log n) \) parallel time and \( O(n) \) processors for pre-processing the tree, where \( n \) is the number of nodes. Thereafter, the algorithm can answer any NCA query in \( O(\log \log n) \) time using a single processor. To our knowledge, this is the best known parallel pointer machine algorithm for the NCA problem. Our NCA algorithm requires an efficient parallel solution of the temporal precedence (\( TP \)) problem [21]. We provide an efficient parallel pointer machine algorithm to solve this problem as well.

The paper is organized as follows. Section 2 presents the computation model—the Pointer Machine—used in the context of this investigation. Section 3 introduces the notations and some basic definitions used throughout the paper. Section 4 presents the basic tree compression algorithm used to pre-process trees. Sections 5 and 6 present algorithms for answering NCA queries in the static and in the dynamic case. Section 7 illustrates how the compression algorithm can be modified to provide an optimal parallel scheme for NCA queries. Section 8 provides some conclusions and indications for future work.
2. Pointer machines

A PPM is a restricted version of a Pointer Machine, that does not allow constant time arithmetic operations. A PPM consists of a finite but expandable collection of records and a finite collection of registers. Each record is uniquely identified through an address. A special address nil is used to denote an invalid address. Each record is a finite collection of fields and all the records have the same structure. Each field may contain either a data or an address. The machine can execute programs; the instructions in a program allow one to move addresses and data between registers and between registers and records’ fields. Special instructions are used to create a new record and to perform conditional jumps. The only conditions allowed in the jumps are equality comparisons between pointer registers. Observe that the content of the data fields will never affect the behavior of the computation. In terms of analysis of complexity, it is assumed that each instruction has a unit cost. This model is essentially the Linking Automaton model proposed by Knuth [16] and is a representative of what has been called atomistic pointer machine model in [2].

PPMs provide a good base for modeling implementation of linked data structures, like trees and lists—indeed, a significant amount of research (e.g., [19,23,20,17]) has recently been proposed in this direction. The PPM model is also simpler, w.r.t. RAM and other models, thus making it more suitable for analysis of lower bounds of time complexity. Observe that while RAM commonly hides the actual cost of arithmetic operations, by allowing operations on numbers of size up to $\log n$ ($n$ being the size of the input) to be treated as constant time operations (uniform cost assumption), PPMs instead make these costs explicit. This model of computation is also close to the traditional method of data access used in logic programming languages—indeed, observe that a significant part of our research has been motivated by problems deriving from the field of logic programming (e.g., [19,20,26]). Note that the PPM model is similar to the Turing Machine model with respect to the fact that the complexity of the arithmetic operations has to be accounted for while analyzing the complexity of an algorithm. It is more powerful than the Turing machine model because it allows for “jumps” based on pointer comparisons in constant time, that is not possible in the Turing machine model. Further details on PPMs and on algorithms developed for PPMs can be found in [2,3,16,24].

As with sequential pointer machine models, various versions of parallel pointer machines have been proposed. They all share the common characteristic that no pointer arithmetic is allowed; these models commonly differ in the way interprocessor communication is realized (see [8] for an extensive discussion). All models rely on the presence of a number of processors; each processor is essentially a sequential pointer machine, and all processors execute the same program in a synchronous fashion. At one end of the spectrum we have the CREW Parallel Pointer Machine [13], where arbitrary (concurrent) read and write operations on a shared memory are allowed (although the shared memory cannot be accessed as an array). At the other end of the spectrum, we have the Parallel PPM model [8]. The Parallel PPM is defined by a collection of finite state synchronous machines (thus ruling out the use of constant time arithmetic), each of which can rearrange its communication links by a bounded amount in one step. Each finite state machine has an ordered set of input lines (also called links), that can be viewed as taps on other processors’ outputs. The usual parallel PPM model allows for unbound fan-out but only constant fan-in. Each finite state machine has the ability to change its links in a restricted way: a finite state machine may redirect one of its links to point to another unit at a “pointer distance” no more than two from it. It has been shown that Parallel PPMs are surprisingly powerful. The details of what exactly constitutes a parallel PPM can be found in Cook and Dymond [8].

There is a number of models whose computational power lies between that of the two models defined above, e.g., the CREW/EREW Parallel pointer machines, the Concurrent-Read Owner-Write model (CROW), and the SIMDAG model with its variants [12]. Several interesting results regarding their computational power have been established. In particular, an $n$-processor CROW PRAM running in time $O(\log n)$ can be simulated by a Parallel PPM in time $O(\log n \log \log n)$ using polynomially many processors. In addition any step-by-step simulation of an $n$ processors CROW PRAM by a Parallel PPM requires time $\Omega(\log \log n)$ per step [9].

3. Notation and definitions

All the problems considered in this paper relate to dynamic binary trees. The following operations are used to manipulate the structure of a dynamic tree:

- **create_tree**($v$) creates a new tree containing a single vertex (the root) labeled $v$; the operation returns as result the node representing the root of the tree;
We will indicate with \( ≼ \) a partial but not a total order.

The problem considered in this context is the problem of determining nearest common ancestors in a dynamic tree. This problem introduces an additional operation to manipulate dynamic trees, called \( nca(x, y) \), where \( x \) and \( y \) are two nodes in the tree. Given two nodes \( x, y \), the operation \( nca(x, y) \) returns a pointer to the nearest common ancestor of the nodes \( x \) and \( y \). More formally,

\[
\text{nca}(x, y) = z \iff (z ≼ x \wedge z ≼ y \wedge \forall w (w ≼ x \wedge w ≼ y \Rightarrow w ≼ z)).
\]

We will indicate with \( NCA \) the problem of executing an arbitrary, correct, on-line sequence of create_tree, expand, remove, and nca operations.

4. A compression scheme for trees

The PPM algorithm to solve the NCA problem we propose is based on a compression scheme, aimed at creating a new tree with logarithmic depth that preserves the ancestor structure of the original tree. The compression algorithm we propose starts from the initial tree \( T = T_0 \) and repeatedly performs two types of compressions, generating a sequence of trees: \( T_0, T_1, T_2, \ldots \) until a tree \( T_k \) containing a single node is obtained. The trees in this sequence are used to build a second tree structure \( (H\text{-tree}), \) that summarizes the NCA information of \( T \). The key property of the \( H\text{-tree} \) is that its depth is at most logarithmic in the number of nodes \( T \). This allows us to produce fast answers to NCA queries.

Given \( T_j, T_{i+1}^L \) the result of leaf-compression of \( T_i \), is obtained by merging each leaf of \( T_i \) with its parent. If a leaf \( \ell \) is merged with its parent \( \text{parent}(\ell) \), then \( \text{parent}(\ell) \) is said to be the direct representative of \( \ell \). A path-compression of a tree \( T_{i+1}^L \) returns a tree \( T_i+1 \), where each path containing only nodes with a single child and ending in a leaf of \( T_i+1 \) is replaced by the head of such path. If a path containing nodes \( v_0, v_1, \ldots, v_k \) is compressed to the node \( v_0 \), then \( v_0 \) is said to be the direct representative of \( v_0, v_1, \ldots, v_k \). A compression of a tree \( T_i \) is the tree \( T_{i+1} \), where \( T_{i+1} \) is the path-compression of \( T_i+1 \), and \( T_i+1 \) is the result of a leaf-compression on \( T_i \). In this notation let \( T = T_0 \).

Fig. 1 shows an example of repeated compression of \( T \). Both leaf and path-compressions start at the frontier of each tree. Each time a leaf-compression is applied, all leaves are merged with their parents. For example, in Fig. 1 leaf-compression removes nodes 10–15 (Fig. 1.1) by merging them with their parents (Fig. 1.2). A path-compression merges all paths ending in a leaf into their heads. For example, in Fig. 1.3 the path composed by nodes 4, 6, 9 has been collapsed to the single node 4 (node 4 is the direct representative of 4, 6 and 9). The tree is compressed starting from the leaves and moving towards the root. In Fig. 1 we have marked the representatives of each compression with darker nodes.

4.1. The H-tree

In order to compute NCA queries in optimal time, it is useful to collect the information about representatives in a separate tree, called Horizontal Tree \( (H\text{-tree}) \). The \( H\text{-tree} \), \( H \), can be constructed from the sequence of trees obtained during the compression process (e.g., the trees shown in Fig. 1). If a leaf-compression is applied to node \( v \) in tree \( T_i \) and \( \ell \) is the direct representative of \( v \) in such compression, then node \( v \) is connected to the last occurrence of \( \ell \) in a tree \( T_j \) \( (i < j) \), where \( \ell \) appears in \( T_j \) as a direct representative of a leaf-compression. If all the children of a node \( a \) in \( T_i \) are leaf-compressed at the same time, then the representative of such children is node \( a \) in \( T_{i+1}^L \) (as for leaves 10, 11 in Fig. 1). If the children of \( a \) are leaf-compressed at different points in time (e.g., the children of 1 in Fig. 1), then the representative of such leaf is the last occurrence of its direct representative in a tree as representative in a leaf-compression. If a path-compression is applied, then all nodes in the path are connected to the head of the list in the next tree (see Fig. 1). Such node is the representative of all nodes in the path. \( H \) is obtained using the single node in the last compressed tree (e.g., the node in \( T_2 \) in Fig. 1) as the root and using the links between nodes and representatives as edges (e.g., the dark edges in Fig. 1).
Observe that the leaves of the original tree are leaves in $H$ although $H$ might have additional leaves. Also, each internal node in $T$ is present in $H$, as each internal node is either a representative in a leaf compression or is involved in a path compression. Observe that $H$ has at most $2n$ nodes, since each node can appear in $H$ because of (possibly many) leaf-compressions at most once and can be involved in a path-compression at most once. Moreover, if a node $v \in T$ appears twice in $H$, then it must be the case that one occurrence of $v$ in $H$ is due to the fact that $v$ was a head in a path compression and is a direct representative in leaf compressions which precede the aforementioned path compression. Note that one occurrence of $v$ in $H$ must be a child of the other occurrence of $v$ in $H$. The next lemma provides a result critical to the efficiency of the compression scheme. Let $\text{subtree}_T(v)$ be the subtree of $T$ rooted at node $v$.

**Lemma 1.** If a node $v$ of $T$ still exists in $T_k$ then the $\text{subtree}_T(v)$ has at least $2^k$ nodes.

**Proof.** Let us prove this result by induction on $k$.

**Base:** For $k = 0$ the result is trivial, since the subtree rooted at $v$ contains at least one (i.e., $2^0$) node. Let us consider the case $k = 1$. For each node $v$ in $T_1$ let us call $w_1$ a leaf of $\text{subtree}_{T_1}(v)$. The node $w_1$ is the result of the previous path-compression on $T^L_1$. Let us call $w_2$ the leaf in $T^L_1$ compressed in $w_1$. The node $w_2$ cannot be a leaf in $T_0$, otherwise it would have been compressed with its parent. It follows that $w_2$ has at least one child, let us call it $w_3$. This implies that $v$ is different from $w_3$. This proves that $\text{subtree}_T(v)$ contains at least $2^1$ nodes.

**Inductive step:** Let us consider the case $k = i$, and let us assume by inductive hypothesis that the results hold for $i - 1$.

For each node $v$ in $T_i$ let us indicate with $w_1$ a leaf of $\text{subtree}_T(v)$ (see also Fig. 2). The node $w_1$ is the result of the previous path-compression applied to $T^L_i$. Let us call $w_2$ the leaf in $T^L_i$ that has been compressed with $w_1$. The node $w_2$ cannot have been a leaf of $T_{i-1}$, else it would have been compressed with its parent. It follows that $w_2$ has at least one child, let us call it $w_3$. The node $w_3$ in $T_{i-1}$ is the result of a path-compression applied to $T^L_{i-1}$. If $w_3$ was the only child of $w_2$, then the path-compression would not have ended in $w_3$. Thus, $w_3$ has a sibling in $T_{i-1}$—let us
call it \( w_4 \). Using the inductive hypothesis applied to nodes \( w_3 \) and \( w_4 \) in \( T_{i-1} \), we can conclude that \( \text{subtree}_T(v) \) has at least \( 1 + 2^{i-1} + 2^{i-1} \) nodes. □

**Corollary 1.** Let \( n \) be the number of nodes in \( T \) and let \( k \) be the minimum integer such that \( T_k \) has a single node. Then \( k \leq \lg n \). In other words, \( T \) gets compressed to a single node within \( \lg n \) compressions.

### 4.2. Answering NCAs using H-trees

Given the query \( \text{nca}(x, y) \), where \( x, y \in T \), it is possible to answer the query using \( H \). In particular, the NCA of two nodes in \( T \) can be computed by first computing an NCA of the “entry-points” for \( x \) and \( y \) in \( H \). The entry-point in \( H \) for \( x \) is simply the lower (or the only) occurrence of \( x \) in \( H \). We provide an intuitive description of this method—the algorithm called \( \text{nca}_H \) is presented in Section 5. We show now that the \( H \)-tree preserves enough NCA information from \( T \). Let \( z \) be the \( \text{nca}_H(x, y) \). If \( z \) is a representative of a leaf-compression, then \( z \) is also the \( \text{nca} \) of \( x, y \in T \). Otherwise let \( z_0, z_1, \ldots, z_k \) be the nodes belonging to the path that has been compressed to \( z \). There are two distinct nodes \( z_i, z_j \) in this path such that the subtree rooted at \( z_i \) (\( z_j \)) contains \( x \) (\( y \)). Thus, the \( \text{nca} \) of \( x \) and \( y \) is the highest node between \( z_i \) and \( z_j \), and answering an NCA query in \( T \) boils down to computing an NCA query in \( H \). From Lemma 1, we can infer that the height of \( H \) is \( O(\lg n) \). In the next subsection we illustrate an algorithm that allows the computation of the \( nca \) of any two nodes of a dynamic tree in worst case time complexity \( O(\lg h) \) per query, where \( h \) is the height of the tree. Using this result, we can compute the \( \text{nca} \) of \( x, y \in H \) in worst-case time \( O(\lg \lg n) \). This allows the computation of the NCA in \( T \) with worst-case time complexity \( O(\lg \lg n) \).

#### 4.3. \( O(\lg h) \) time computation of NCA

We provide a worst-case \( O(\lg h) \) solution for the \( \mathcal{NCA} \) Problem on PPMs, where \( h \) is the height of the dynamic tree. It is worth noting that a worst-case constant-time solution is not possible for the \( \mathcal{NCA} \) Problem on pure pointer machines, even in the static case. A lower bound of \( \Omega(\lg \lg n) \) (\( n \) being the number of nodes in the tree) is provided at the end of this section (see also [15]).

The basic idea behind our solution is to maintain the depth of the tree nodes. For any vertex \( x \) in the tree, let us denote with \( \text{anc}(x, d) \) the ancestor of vertex \( x \) lying at depth \( d \) in the tree. Thus, if we have two nodes \( x \) and \( y \) and \( \text{anc}(x, d) = \text{anc}(y, d) \), then we can infer that \( \text{nca}(x, y) \) is at a depth at least \( d \) in the tree. Otherwise: \( \text{nca}(x, y) = \text{nca}(\text{anc}(x, d), \text{anc}(y, d)) \).

In our solution, with each node in the dynamic tree we store the **depth** of the node. The depths of the tree are represented by nodes in another binary tree (the tree to the right in Fig. 3). Each time a new node is inserted in the main tree, we calculate its depth. When the node \( x \) is added to the main tree, we can determine its depth by looking at
the depth pointer stored in the parent node of \( x \). We can assume that each node in the depths tree contains a pointer to the logical successor node in such tree (see the dark links in Fig. 3). If the depth node associated to the parent of \( x \) in the main tree does not have a successor, then this means that the new node \( x \) lies in a new level of the main tree, which will lead to the creation of a new node also in the depth tree. The pointers \( p1 \) and \( p2 \) can be used to determine when a new level has to be created in the depths tree.

Observe that the creation of the depth structure can be performed in \( O(1) \) time each time an \( \text{expand} \) operation is performed.

In addition, we maintain for each node in the tree a list of pointers to selected ancestor nodes (the \textit{predecessors list} described in the next subsection). These pointers are used to perform a binary search leading to the identification of the nearest common ancestor. The resulting data structure, discovered independently, resembles the one used in [25], but does not assume constant time arithmetic capabilities.

The rest of this section describes how to realize these ideas. We start by describing the data structures needed to efficiently solve the \( \text{NCA} \) problem.

4.3.1. \textit{Predecessors list}

4.3.1.1. General definitions. To support the execution of the \( \text{nca} \) operation we need an additional data structure superimposed on the dynamic tree: a \textit{predecessors list} (p-list) attached to each node of the tree. Each element of the p-list is a record with four fields, called respectively \textit{data link}, \textit{right-up link}, \textit{left-up link}, and \textit{middle-up link} (in addition to the standard fields required to maintain the linked list).

In the rest of this paper we will make use of the following notation: given a node \( x \) in the tree, \( \text{last}(x) \) (\( \text{first}(x) \)) denotes the last (first) record in the p-list associated to the node \( x \). In addition, \( \text{pred}(x, i) \) indicates the \( i \)th element of the p-list of \( x \). If \( s \) is an element of a p-list, then \( \text{data}(s) \), \( \text{right}_\text{-link}(s) \), \( \text{left}_\text{-link}(s) \), and \( \text{middle}_\text{-link}(s) \) represent the content of the data link, right-up link, left-up link, and middle-up link stored in the element \( s \). The successor (predecessor) of an element \( s \) of a p-list is indicated with \( \text{next}(s) \) (\( \text{prev}(s) \))—i.e.,

\[
\text{first}(x) = \text{pred}(x, 0), \\
\text{next}(\text{pred}(x, i)) = \text{pred}(x, i + 1), \\
\text{prev}(\text{pred}(x, i + 1)) = \text{pred}(x, i).
\]

The p-list of node \( x \) contains \( \lceil \lg h \rceil + 1 \) elements, where \( h \) is the depth of node \( x \) in the tree. The p-list of \( x \) is designed to contain pointers to ancestor nodes of \( x \). In particular the p-list of \( x \) points to the ancestors of \( x \) which have distance 1, 2, 4, 8, \ldots, \( 2^{\lceil \lg h \rceil} \) from node \( x \). Let \( \text{jump}(x, k) \) denote the ancestor of node \( x \) which is at a distance \( k \) from \( x \) in the tree. The data link field of the \( i \)th element of the p-list of \( x \) contains a pointer to the node \( \text{jump}(x, 2^i) \), for \( i = 0, 1, \ldots, \lceil \lg \text{depth}(x) \rceil \). More precisely, if \( x \) is a node in the tree, then

\[
\text{data}(\text{pred}(x, i)) = \text{jump}(x, 2^i).
\]

The other three pointers maintained in each element of the p-list, the \textit{right-up link}, the \textit{left-up link}, and the \textit{middle-up link}, point to elements in the p-lists of other nodes in the tree. As we will show in the rest of this section, the \textit{right-up links} are employed to help in the creation of the p-list, while the \textit{left-up} and the \textit{middle-up} links are employed to allow the efficient computation of the \( \text{nca} \).
Fig. 4. Right-up links between nodes of the p-lists.

The meaning of these links can be described as follows. If \( x \) is a node in the tree, then the right-up link of \( \text{pred}(x, i) \) is a pointer to the \((i + 1)\)th element of the p-list of the node \( \text{jump}(x, 2^i) \), if it exists (it is \text{nil} otherwise). More precisely,

\[
\text{right\_link}(\text{pred}(x, i)) = \text{pred}(\text{jump}(x, 2^i), i + 1).
\]

This is illustrated in Fig. 4. Similarly, the middle-up and the left-up links of the \( i \)th element of the p-list of \( x \) contain pointers to the \( i \)th and the \((i - 1)\)th elements of the p-list of \( \text{jump}(x, 2^i) \)—i.e.,

\[
\text{middle\_link}(\text{pred}(x, i)) = \text{pred}(\text{jump}(x, 2^i), i),
\]

\[
\text{left\_link}(\text{pred}(x, i + 1)) = \text{pred}(\text{jump}(x, 2^{i+1}), i).
\]

Observe that only one of the three link pointers is actually required—e.g., the right-up link can be directly computed from the middle-up link. We maintain the three links for the sake of clarity of presentation.

4.3.1.2. Management of the p-list. The management of the p-lists is somewhat involved. Nothing special needs to be done whenever \text{create\_tree} and \text{remove} are performed—observe that by definition the \text{remove} operation is applied only to leaves of the tree, and it will not affect the p-lists of other nodes. Let us show how we can create the p-list for each new node created when an \text{expand} operation is performed.

Let us consider the execution of \text{expand}(x, y, z). Let us focus on the creation of \( y \) as a child of node \( x \) (the creation of \( z \) is basically identical). The construction can be accomplished by taking advantage of the properties of the right-up links. The creation of a new p-list can be obtained by simply following the appropriate right-up links and copying the p-list elements encountered. In particular, the creation of the p-list for \( y \) can be performed as follows:

1. The first element of the p-list for \( y \) contains
   
   (a) A pointer to the node \( x \) in the data field, i.e.,
   
   \[
   \text{data}(\text{first}(y)) = x.
   \]

   (b) A pointer to the second node in the p-list of \( x \) in the right-up link field, i.e.,
   
   \[
   \text{right\_link}(\text{first}(y)) = \text{pred}(x, 1).
   \]

   (c) A pointer to the first element of the p-list of \( x \) in the middle-up link field, i.e.,
   
   \[
   \text{middle\_link}(\text{first}(y)) = \text{first}(x).
   \]

   (d) A \text{nil} pointer in the left-up link field.

2. All the other elements of the p-list for \( y \) can be obtained essentially by following the right-up link list of p-list elements, starting with the first element of the p-list of \( x \) and copying each one of them. The appropriate right-up, middle-up, and left-up links for this new list can be calculated simultaneously, as illustrated in the algorithm in Fig. 5.
Observe that this construction can be improved; if the newly created node \( y \) already has a sibling \( z \), then the p-lists of \( y \) and \( z \) are identical and can be shared. Similarly, if two nodes \( y \) and \( z \) are at the same level and \( \text{jump}(y, 2^i) = \text{jump}(z, 2^i) \), then
\[
\text{pred}(y, j) = \text{pred}(z, j)
\]
for \( j \geq i \). Nevertheless, it is unclear whether such optimizations may lead to an improved asymptotic space complexity.

**Example 1.** Consider the tree in Fig. 6\(^1\) and let us assume that a new node (node 10) is inserted as right child of 9. The figure shows the p-lists of the existing nodes as well as the p-list of the new node. This has been created applying the algorithm described earlier. The first element of the p-list contains a pointer to parent 9; the rest of the p-list is a copy of the list made by right-up links and starting from the first element of the p-list of 9 (such list is shown using solid lines in the figure). The right-up links of the new elements are shown as dashed lines.

Note that the creation of the new p-list requires \( \Theta(\lg h) \) time, where \( h \) is the depth of the new node. Hence, the \texttt{expand} operation takes time \( \Theta(\lg h) \). The correctness of the algorithm follows by observing that the data field of \( \text{pred}(y, i + 1) \) is copied from the \( i \)th element of the p-list of the node pointed to by the data field of the \( i \)th element of the p-list of \( y \), i.e.,
\[
\text{data}(\text{pred}(y, i + 1)) = \text{data}(\text{pred}(\text{data}(\text{pred}(y, i)), i)).
\]
Noting that \( \text{jump}(\text{jump}(y, 2^i), 2^i) = \text{jump}(y, 2^{i+1}) \), it is easy to establish inductively that the data fields of the p-list of \( y \) are correctly computed. The fact that the right-up, middle-up, and left-up links are correctly computed is also straightforward to establish.

### 4.3.2. Computing the nca

We subdivide the process of computing \( \text{nca}(x, y) \) into two subproblems: (i) determine the nearest common ancestor under the assumption that \( x \) and \( y \) are at the same depth; (ii) given \( x \) and \( y \) such that \( \text{depth}(x) < \text{depth}(y) \) determine the ancestor \( y' \) of \( y \) such that \( \text{depth}(y') = \text{depth}(x) \). It is clear that the ability to solve these two subproblems will provide a general solution to the task of determining \( \text{nca}(x, y) \) for arbitrary nodes \( x, y \). Below we provide a \( O(\lg h) \) solution for both these subproblems. This gives us a \( O(\lg h) \) solution for the NCA problem.

#### 4.3.2.1. Determining nca for same depth nodes

We will make use of the elements of the p-list. Let us assume \( x \) and \( y \) to be two nodes that are at the same depth \( (h) \) in the tree. Then, the p-lists of \( x \) and \( y \) each contain \( \lfloor \lg h \rfloor + 1 \) elements. The computation of \( \text{nca}(x, y) \) is performed through the algorithm in Fig. 7.

\begin{verbatim}
1:  scan := first(x) ; // first element of p-list of x
2:  p-list(y) := new () ; // create first element p-list of y
3:  data(first(y)) := x;
4:  right_link(first(y)) := next(first(x));
5:  left_link(first(y)) := nil
6:  middle_link(first(y)) := first(x);
7:  prev := first(y);
8:  while (scan != nil) do
9:    next(prev) := copy(scan); // create copy of element scan
10:   right_link(prev) := next(scan);
11:   middle_link(prev) := scan;
12:   left_link(prev) := prev(scan);
13:   scan := right_link(scan);
14:   prev := next(prev);
15: endwhile;

Fig. 5. Computation of right-up links.
\end{verbatim}

\(^1\) The figure shows only the nodes on the path from the root to the new node.
The idea behind the algorithm is to locate the nearest common ancestor of two nodes by performing successive “jumps” in the tree, making use of the pointers stored in the p-list of the two nodes. The first loop (lines 1–8) is used to deal with the special case where the nca lies in the highest part of the tree (above the highest node pointed by the p-list). The loop in lines 10–13 compares ancestors of the two nodes, starting from the ancestor in the p-list which is farther away from the nodes x and y. During the successive iteration of this first loop we compare the nodes $\text{jump}(x, 2^i)$
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig8.png}
\caption{Searching for nearest common ancestor.}
\end{figure}

4.3.2.2. Determining equal height ancestor. Fig. 9 provides a procedure that, given nodes $x, y$ such that $\text{depth}(x) < \text{depth}(y)$, returns a node $y'$ such that $y' \leq y$ and $\text{depth}(y') = \text{depth}(x)$. The method uses the depth information stored in the nodes to determine the jump necessary to find the ancestor $y'$, and uses the p-lists to jump to the correct ancestor. The subtraction in line 1 creates a list of records representing the binary number obtained from the subtraction of the depths of $x$ and $y$. This operation can be done in time $O(\lg h)$ ($h = \text{depth}(y)$) by simply scanning the two lists of records representing $\text{depth}(y)$ and $\text{depth}(x)$ from right to left (recall that the length of such lists is bounded by $h$). In Fig. 9, a jump of size $2^i$ is performed from the current ancestor of $y$ if digit $j_i$ is one. All jumps together add to a total jump of $j$ from $y$. The overall complexity is $O(\lg h)$.

4.3.2.3. Comment. It is possible to extend the solution to trees with unbounded arity. For trees with unbounded degree, when an operation $\text{expand}(x, a_1, a_2, \ldots, a_r)$ is performed only one p-list is created. Each of $a_1, a_2, \ldots, a_r$...
Theorem 2. The overall idea is to insert elements in a tree which is composed only by a right spine; each element inserted is as well (where $n$ is the number of nodes in the tree).

Proof. Let us implement the two operations of the $TP$ problem and let us show how we can use it to build a solution for the $TP$ problem. The $TP$ problem is the problem of performing an on-line sequence of operations, and the only two allowed operations are $\text{insert}(x)$—that adds the element $x$ to a collection of elements—and $\text{precedes}(x, y)$—that verifies whether the element $x$ has been inserted before the element $y$. In [21] we have proved that the $TP$ problem has a lower bound time complexity of $\Omega(\lg \lg n)$ per operation, where $n$ is the number of elements inserted. This allows us to conclude the following result:

Theorem 2. The $NC\! A$ problem has a lower bound time complexity of $\Omega(\lg \lg n)$, where $n$ is the number of nodes in the tree.

Proof. Let us implement the two operations of the $TP$ problem using the operations available in the $NC\! A$ problem. The overall idea is to insert elements in a tree which is composed only by a right spine; each element inserted is a new leaf created down the right spine. Each $\text{precedes}$ operation can be realized as follows: $(\text{precedes}(x, y) \iff nca(x, y) = x)$. This allows us to conclude that the $NC\! A$ problem has a lower bound time complexity of $\Omega(\lg \lg n)$ as well (where $n$ is the number of nodes in the tree). 

5. A sequential algorithm for NCA queries in the static case

In the next sections, as in [1], we use a more general definition of $nca$: $nca(x, y) = (z, z_x, z_y)$, where $z$ is the $nca$ of $x$ and $y$, and if $x = z$ ($y = z$) then $z_x = z$ ($z_y = z$) else $z_x$ ($z_y$) is the ancestor of $x$ ($y$) such that $z_x$ ($z_y$) is a child of $z$.

In the static case $T$ is pre-processed before any query is executed. Conceptually, the pre-processing creates $2k + 1$ trees, named $T_0, T^L_1, T_1, T^L_2, \ldots, T^L_k, T_k$. The $T_0$ tree is equal to $T$ and each other tree is the result of the corresponding compression. To improve the time and space required, $T_i$ and $T^L_i$ trees are not explicitly created. Each time only the nodes being encountered for the first time are created anew, except that during a path compression a new node is created for the head of the path. $H$ is composed of the union of all nodes created during the various compression phases.

5.1. Data structures

For each $w$ in $T$, let $\text{entry}(w)$ be the entry point of $w$ in $H$. Note that each node in a tree $T_i$ or $T^L_i$ is a copy of a node existing in $T$; if the node $v \in H$ is a copy of node $u \in T$, then $\text{node}(v)$ is a pointer to $u$. Let $\text{children}(v)$ be a pointer to a list of nodes containing the children of node $v$ in $H$, and let $\text{parent}(v)$ denote the parent of node $v$ in $H$. During the pre-processing phase, we will also make use of two flags associated to each node of $H$:

1. A flag $\text{leaf-compr}(v)$ that indicates whether the node $v$ is the result of a leaf compression.
2. A flag $\text{is-leaf}(v)$ that indicates if the representative produced by a leaf compression is a leaf in the new tree.

5.2. Construction of the $H$-tree

To answer the NCA queries, we require the ability to efficiently compare the depth of nodes that appear on the same branch. This can be accomplished by making use of the data structures developed to solve the $TP$ problem [21,17].
Using the optimal scheme from Pontelli and Ranjan [17], we can build a data structure in $O(n)$ time that allows us to compare depths in $O(\lg \lg n)$ time.

The pre-processing algorithm used to construct $H$ is described in Fig. 10. With one visit of $T$, one can create the leaves of $T_0$ by simply copying each leaf $v$ of $T$ to a new node $u$ and updating both $\text{entry}(v) = u$ and $\text{node}(u) = v$ (lines 1–4 of Fig. 10). After this, the process proceeds by repeatedly applying leaf-compression (lines 7–19) and path-compression (lines 21–42). The process stops as soon as we are left with a tree containing a single node. The last $T_k$, $k \leq \lg (n)$, has only one node.

**Lemma 2.** The time required to construct $H$ is $O(n)$, where $n$ is the number of nodes in $T$.

**Proof.** Let us start by observing that each node $v$ in $T$ appears at most 2 times in $H$ (once as representative of a leaf-compression and once as a representative of a path-compression). Let us call $c(v)$ the number of children of $v$ in $T$. Each node $v$ can be involved in:

- at most 1 leaf compression where $v$ is a leaf;
- at most 1 path compression.

As we will show in the successive two subsections, each individual leaf- and path-compression can be performed in worst-case time complexity $O(1)$ per node.

Observe that during a leaf-compression, each node $v$ can either

- be compressed to its parent;
- be the direct representative of any of its $(c(v))$ children.

By distributing the cost of leaf-compressions to the leaf nodes, the amortized cost of leaf compressions is $O(1)$ per node.

Observe now what happens during a path compression; let us assume that node $v$ gets path-compressed to a representative $w$:

- if $v \neq w$, then $v$ will not be present in the $T_i$ tree obtained from the path compression;
- if $v = w$, then $v$ will appear as a leaf in the $T_i$ tree resulting from the path-compression; this also means that $v$ will be removed at the next step (when $v$ is leaf-compressed).

As before, by distributing the cost of path compression to the nodes involved, the amortized cost is $O(1)$ per node.

Hence, the total cost over all compressions is at most $\sum_{v \in T}(1 + 1) = 2n$. □

### 5.3. Leaf compression

For each leaf $v$ in $T_i$, a leaf-compression is applied. A new representative $w$ for the parent of $v$ in $T$ ($\text{entry}(\text{parent}(v)) = w$) is added to the tree $T^L_{i+1}$, if this is the first time a node is compressed to $w$ (lines 8–12). After this check, the node $v$ is compressed to its parent $w$ (lines 13, 14). Once all the leaves of $T_i$ are processed, for each node $w$ in $T^L_{i+1}$ we set the flag $\text{is-leaf}(w) = \text{true}$ if $w$ is a leaf in $T^L_{i+1}$ (lines 15–19). The $\text{is-leaf}$ flag indicates the nodes where the next path-compression will start.

Observe that determining the leaves of the tree can be efficiently performed, e.g., by keeping a copy of $T$ and removing nodes from it as they get compressed. In the copy of the tree we can maintain the frontier as a linked list—in time $O(1)$ on a PPM [20].

### 5.4. Path compression

Path-compression is initiated from each node $v$ in $T^L_i$, such that $\text{is-leaf}(v) = \text{true}$. We have already mentioned in the previous subsection how this test can be performed in $O(1)$ time. Execution of path-compression starting from $v$ leads to the addition of a node $w$ to $T_i+1$. $w$ will be the representative of the compressed path starting from $v$ (lines 20–24). When the path compression stops, the node $w$ will be assigned to the correct node in $T$—i.e., the representative of the path.
Fig. 10. Construction of the $H$-tree.

The following iteration (lines 25–27) assigns to each $v$ in the current path the direct representative $w$, and tries to extend the path leading to the root with $\text{parent}(v)$, if $\text{parent}(v)$ is still part of the path (lines 28–39). When the iteration stops, the current node $v$ is the head of the path, and its representative $w$ in $T_{i+1}$ is a copy of the node $v$ (lines 40–42). The check that verifies whether a node is part of a path can be implemented in constant time without the use of arithmetic (as described earlier).
5.5. Final pre-processing step

Once $H$ has been constructed, it is further processed to enrich it with the data structures required to support the computation of NCAs in $O(\lg h)$ time ($h$ being the height of $H$). The details of these data structures have been described in [19] and in Section 4.3. The process requires the creation of the p-list data structure for each node in $H$. Since the height of $H$ is at most $O(\lg n)$, the p-list of each node $v$ contains at most $\lg \text{depth}(v)$ elements, and each element of a p-list can be built in $O(1)$ time, the process of creating these additional data structures requires $O(n \lg \lg n)$ time. Observe also that the space requirement for this solution becomes $O(n \lg \lg n)$. It is possible to improve this pre-processing time, reducing it to $O(n)$, through the use of the MicroMacroUniverse approach described in [1] and in Section 6: in this case $H$ is partitioned in $\mu$Trees (Micro-Trees) with depth at most $\lg \lg n$.

5.6. Answering NCA queries

To compute the $\text{nca}$ of $x, y \in T$, the algorithm $\text{nca}_H(x, y)$ works as follows:

1. Compute the $\text{nca}(\text{entry}(x), \text{entry}(y)) = (z, z_x, z_y), z \in H$. The computation can be performed using the algorithm in Section 4.3.
2. If $\text{leaf-compr}(z) = \text{true}$ then return $(\text{node}(z), \text{node}(z_x), \text{node}(z_y))$.
3. Otherwise $z$ is the result of a path compression. In this case if $\text{node}(z_x)$ is higher than $\text{node}(z_y)$ in $T$, the $\text{nca}$ is $(\text{node}(z_x), w_1, w_2)$, where $w_1$ is the node corresponding to the child of $z_x$ that is ancestor of $\text{entry}(x)$ in $H$ and $w_2$ is the node in $T$ corresponding to the left sibling of $z_x$ in $H$. $w_1$ can be obtained by using p-lists in time $O(\lg \lg n)$ and $w_2$ can be found in constant time. The case where $\text{node}(z_y)$ is higher than $\text{node}(z_x)$ is symmetric. The test to check if $\text{node}(z_x)$ is higher than $\text{node}(z_y)$ can be performed in time $O(\lg \lg n)$ using the previously mentioned depth information.

6. A sequential algorithm for NCA queries in the dynamic case

In this section, we present a sequential algorithm to address the NCA problem in the case where modifications of the tree structures (as addition and deletion of leaves). The method will guarantee the following result:

**Theorem 3.** The method offers a $O(n + q \lg \lg n)$ solution to the NCA problem, where $n$ is the number of add-leaf/delete operations and $q$ is the number of NCA queries.

The algorithm follows an approach similar to the MicroMacroUniverse described in [1,11] in conjunction with repeated use of the static algorithm described in Section 5. Let $n$ be the number of nodes in $T$. We consider two types of trees (Fig. 11):

- The $\mu$Trees (Micro-Trees) are trees of a forest $S$ of disjoint subtrees partitioning $T$.
- The $M$Tree (Macro-Tree) is a tree collecting the roots of the $\mu$Trees.

The MTree essentially compresses the nodes of each $\mu$Tree into its root node and preserves the structure of $T$ on these resulting root nodes. The height of each $\mu$Tree is restricted to be at most $c_T(n)$. When a node $v$ is added to $T$, if the $\mu$Tree containing $\text{parent}(v)$ has a depth greater than $c_T(n)$, then a new $\mu$Tree rooted in $v$ is created else the node is simply added to the appropriate $\mu$Tree. To obtain the optimal result, the MicroMacroUniverse approach is applied again to the $\mu$Trees. For the MTree the scheme used is based again on partitioning the tree into disjoint subtrees. However this partitioning is more dynamic in nature, since the subtree to which a node belongs can change.

In order to answer an NCA query, our algorithm first solves the problem in the MTree and then refines the solution by working in the appropriate $\mu$Tree. We denote with $\text{nca}_\mu$ the NCA algorithm used for the $\mu$Trees in $S$ and $\text{nca}_M$ the NCA algorithm used for the MTree. Each time a node $v$ is inserted in $T$, $v$ is also inserted in a data structure that collects the relative height information of the nodes, using a Temporal Precedence list. As described in Section 5.6, this information will be required to perform an NCA query using p-lists.
6.1. The $nca_M$ algorithm

In this section we present an algorithm to compute the $nca$ with a cost of $O(N + Q \log \log N)$, with $N$ add-leaf/delete operations and $Q$ NCA queries in the MTree. The problem is solved by using another MicroMacroUniverse approach applied to the MTree. The intuitive idea is to dynamically maintain a set of trees, pre-processed with the static algorithm presented in Section 5. As in the standard micro–macro schemes the MTree is partitioned into micro-trees whose roots are maintained in a macro-tree. However, the partitioning is dynamic in nature. The sizes of the micro-trees can be very different as well as the micro-tree to which a node belongs changes dynamically. Let us call each of these trees $d\mu$Tree (dynamic-micro-Tree). The pre-process of a $d\mu$Tree allows us to efficiently solve each NCA query on that tree, using the $nca_H$ algorithm presented earlier. The root of each $d\mu$Tree is represented by a node in another tree, called $dMTree$ (dynamic-Macro-Tree). We will show that the $dMTree$ has a “small” depth—thus, simpler NCA algorithms (e.g., the one based on p-lists from Section 4.3) can be used here to provide efficient NCA computation.
Let the *pre-process* of a $dμ$Tree $T_m$ be the static pre-process described in Section 5 applied to the tree $\text{subtree}_{MTree}(\text{root}(T_m))$. Thus, when $T_m$ is pre-processed, all other $dμ$Trees hanging on $T_m$ are merged in a single new $dμ$Tree. The basic idea is to wait to re-pre-process a $dμ$Tree $T_m$ until the number of nodes in $\text{subtree}_{MTree}(\text{root}(T_m))$ has doubled since the last pre-process of $T_m$. To answer an NCA query we first solve the problem with the p-list NCA algorithm from Section 4.3 on the $dMT$ree and then we “refine” that solution using the ncaH on the $dμ$Tree associated to the result obtained from the $dMT$ree.

**Dynamic insertions:** Let $T_m$ be a $dμ$Tree and $v_m$ the node representative of $T_m$ in the $dMT$ree. Let us define $\text{root}(T_m)$ to be the root of $T_m$ in the MTree, $\text{micro}(v_m)$ a pointer to $\text{root}(T_m)$, $\text{macro}(T_m)$ a pointer to $v_m$. We also maintain the size of $T_m$ by keeping a pointer $\text{size}(T_m)$ that points to a list that has length $|T_m|$. $\text{size}(T_m)$ is created and inserted with a number of nodes equal to the number of nodes in the $\text{subtree}_{MTree}(\text{root}(T_m))$, when $T_m$ is pre-processed. The $\text{size}(T_m)$ list is used as a decrementing counter to decide when to do another pre-process.

Each time a node $v$ is inserted in the MTree as child of $v$, a new $dμ$Tree $T_m$ is created, and the representative of $\text{root}(T_m) = v$ is added in the $dMT$ree as child of $\text{macro}(T_m)$, where $T_m$ is the $dμ$Tree containing $v$. For each node on the path $P$ from $\text{macro}(T_m)$ to the root of the $dMT$ree (these correspond to some $dμ$Tree $T'_m$), we update the number of new nodes added in the $\text{subtree}_{MTree}(\text{root}(T'_m))$ by 1. This can be done decrementing the “counter” $\text{size}(T'_m)$ by one, that is, shifting the pointer $\text{size}(T'_m)$ to the next node in that list. If a $dμ$Tree $T_m$ has consumed all nodes in $\text{size}(T_m)$, then $T_m$ has to be pre-processed again. Let us call $v_h$ the highest node in the path $P$ considered, such that $\text{micro}(v_h)$ has to be pre-processed. The pre-process is applied to $\text{subtree}_{MTree}(v_h)$, which becomes the new $dμ$Tree. All nodes in $\text{subtree}_{MTree}(v_h)$ are deleted and replaced by the node $v_h$. The $\text{size}(\text{micro}(v_h))$ list is initialized with the insertion of a number of nodes equal to the number of nodes contained in $\text{subtree}_{MTree}(v_h)$.

Notice that each time a node $v$ is involved in a pre-process resulting in a tree $T'$, the size of $T'$ is at least twice the size of the tree $T''$ which contained $v$ before the pre-process. The following result can be derived:

**Proposition 4.** A node $v$ in the MTree of size $t$ is involved in at most $\lg t$ distinct pre-process operations.

As we will show in the next lemma, the $dMT$ree has depth less or equal to $O(\lg N)$, where $N$ is the number of nodes in MTree. Thus the update of counters may be performed $O(\lg N)$ time for each insertion of a node in the MTree.

**Lemma 3.** Immediately after a pre-process, if a path $P$ starting from a node in $dMT$ree and ending on a leaf has $k$ nodes, then the total number of nodes in $dμ$Trees represented by nodes in $P$ is at least $2^k - 1$.

**Proof.** Let us prove this result by induction on $k$.

*Base:* For $k = 1$ the result is trivial, since the $dμ$Tree represented contains at least one (i.e., $2^0$) node.

*Inductive step:* Let us consider the case $k = i > 1$, and let us assume by inductive hypothesis that the results hold for $i - 1$.

Let $v_l$ be the first node in the path $P$ and $R$ the rest of the path from $v_l$ to the appropriate leaf. Let $T_m$ be the $dμ$Tree represented by $v_l$ in the $dMT$ree. Using the inductive hypothesis applied to $R$ we can infer that the nodes in $R$ represent at least $2^{i-2}$ nodes in the MTree. The tree $T_m$ has at least $1 + 2^{i-2}$ nodes, as otherwise, according to the algorithm, it (or one of its ancestors) should be pre-processed again (and will therefore be a leaf in the resulting $dMT$ree) and by hypothesis the MTree was just pre-processed. Thus the total number of nodes represented by the path $P$ is at least $1 + 2^{i-2} + 2^{i-2} > 2^{i-1}$. □

It follows that the $dMT$ree has depth at most $\lg N$, where $N$ is the number of nodes in the MTree. The MTree has at most $N \leq n/c_T(n)$ nodes. Choosing $c_T(n) = \lg n \lg \lg n$, the MTree has at most $O(n/(\lg n \lg \lg n))$ nodes. Applying Lemma 3, we conclude that the $dMT$ree has depth at most $\lg n$. Thus, the NCA in the $dMT$ree can be computed in time $O(\lg \lg n)$ using p-lists.

We now show that $n$ insertions in $T$ will cost $O(n)$ to maintain the MTree structure. We showed that a pre-process of a tree with $t$ nodes in the static case costs $O(t \lg t)$. Let $N$ be the number of nodes in the MTree. From Proposition 4, we know that a node $v$ in the MTree is involved in at most $\lg n$ pre-processes. Each of them will cost $\ell \lg \ell$, where $\ell$ is the size of the tree pre-processed. Thus, for each $v$ the amortized cost per process is $\lg \ell \leq \ell \leq \lg N$ and the cost per node is $\lg N \lg \lg N$. Recalling that $N = n/c_T(n)$ and $c_T(n) = \lg n \lg \lg n$, the total amortized cost is $O(1)$ per insertion in $T$. 

NCA queries: Let us now show how to compute the \( nca_M(x, y) \) with \( x \) and \( y \) in MTree. It is possible to find \( x' \) and \( y' \) \( d\mu \)Trees containing \( x \) and \( y \), respectively in constant time—e.g., once a node is pre-processed, we can directly set in the node a pointer to the corresponding \( d\mu \)Tree.

If \( x \) and \( y \) are in the same \( d\mu \)Tree \( \mathcal{T}_m \) return \( nca_H(x, y) \) using the previously pre-processed \( H \) tree for \( \mathcal{T}_m \). Otherwise, we can compute the \( nca(root(x'), root(y')) = (z_x, z_y, z) \) on the \( d\mathcal{M} \)Tree, using \( p \)-lists. If \( z_x = z \) then the result is given by \( nca_H(x, parent(micro(z_y))) \). Otherwise, the result is

\[
nca_H(parent(micro(z_x)), parent(micro(z_y))).
\]

The algorithm \( nca_H \) requires \( O(\lg \lg n) \) time and the \( p \)-list algorithm used for the \( d\mathcal{M} \)Tree requires \( O(\lg t) \), where \( t \leq \lg(n/c_T(n)) \) and \( c_T(n) = \lg n \lg \lg n \). This allows us to conclude that the total time is \( O(\lg \lg n) \).

6.2. The \( nca_{\mu} \) algorithm

In this section we provide an algorithm with an amortized time complexity of \( O(1) \) per insertion and worst-case complexity of \( O(\lg \lg n) \) per query for the \( \mu \)Trees.

The scheme uses the standard MicroMacroUniverse approach on the \( \mu \)Trees. The optimal solution is computed combining an optimal solution on \( M\mu \)Tree and an optimal solution on \( \mu \mu \)Tree. Choosing \( c_S(n) = \lg n \lg \lg n \) and recalling that \( c_T(n) = \lg n \lg \lg n \), all the \( \mu \)Trees can be processed in \( O(n) \) time. To find the \( nca_{\mu} \) of two nodes \( x \) and \( y \) in a \( \mu \)Tree, we combine a \( p \)-list search on \( M\mu \)Tree and a brute force search applied on the resulting \( \mu\mu \)Tree. Clearly this requires \( O(\lg \lg n) \) time.

6.3. Handling deletions

Observe that the deletions are not performed explicitly, instead the deleted nodes are just marked as such. The marked nodes are deleted at the time when they are involved in the next pre-processing. We do not update the counters when nodes are deleted. This does not affect our analysis, because the number of operations is greater than the number of nodes in \( T \).

6.4. Discussion

6.4.1. Some implementation details

The previously described algorithms require the comparison between depths of nodes and the values of the functions \( c_T(n) \) and \( c_S(n) \).

Let us start by assuming \( n \) to be known (see later about this assumption). In this case, before starting the main computation, we can construct lists of length \( c_T(n) \) and \( c_S(n) \)—this can be achieved in time \( O(n) \), by building first a list of length \( n \), \( \lg n \), and \( \lg \lg n \) (using a complete binary tree to determine \( \lg n \) from the list \( n \), and then using the longest branch in such tree to build another binary tree whose height is \( \lg \lg n \)). Once these data structures are available, we can readily construct a list representing \( c_S(n) \)—i.e., the list of length \( \lg \lg n \)—and a list representing \( c_T(n) \)—by scanning the list of length \( \lg \lg n \) as many times as the number of elements in the list of length \( \lg n \).

The mentioned comparisons can be realized by storing in each node of the tree points to the elements of the lists \( c_S(n) \) and \( c_T(n) \) indicating the depth of the nodes—this can be realized in time \( O(1) \) (as the depths of a node can be determined from the depths of the parent of the node in the tree). The number comparison is simply obtained looking at the pointer from a depth tree node to the list, and testing whether the node pointed is the last element of the list.

Observe that the knowledge of the value of \( n \) at the beginning is not a severe restriction. In [1] it is observed that “If the total number of nodes is not known in advance, then \( n \) is guessed to be a constant and each time the number of nodes extends the guess we double the guess and reconstruct the structure”, and the observation easily applies to our

\[ \text{The case } z = z_y \text{ is symmetrical.} \]
case as well. Alternatively, one can observe the following:

- We can use as value for $n$ the number of nodes existing at that particular moment in the tree; this number will change during the computation, thus growing the structures for the representation of $c_T(n)$ and $c_s(n)$ as nodes are inserted in the tree.
- The structures need to be reconstructed each time the dynamically changing value of $n$ reaches a power of 2. If $n$ is the final size of the tree, then the total cost deriving from the additional reorganizations of the data structures required is
  \[ \sum_{i=1}^{\lfloor \log_2 n \rfloor} (2^i \log i) = \Theta(n \log n \log \log n). \]

These additional re-processing do not modify the overall complexity of the solution.

Observe that the overall space requirement for this solution is $O(n \log \log n)$.

### 6.4.2. APMs vs PPMs

The commonly used APM model allows constant time arithmetic on $\Theta(\log n)$ sized integers. The PPM does not allow such arithmetic, and one has to account for simulating any arithmetic needed, when analyzing the running time. The arithmetic can be simulated in PPMs by explicitly representing the integers via $\Theta(\log n)$ sized lists. This entails that a generic translation (that just simulates the arithmetic) of APM algorithms to PPMs will incur a polylog penalty. More precisely an algorithm $A$ that runs in time $t(n)$ on an APM and uses any arithmetic at all, will take time $t(n) \log^k n$ for some $k > 0$ on a PPM. We present an interesting result about the $NCA$ problem. We show that any optimal APM algorithm for the $NCA$ problem can be converted into a PPM algorithm without incurring any penalty.

**Proposition 5.** An APM algorithm $A$ solving the $NCA$ problem with amortized cost of $O(\log^k n)$ per insertion and worst-case cost $O(\log \log n)$ per query, can be translated into a PPM algorithm with an amortized cost of $O(1)$ per insertion and worst-case cost $O(\log \log n)$ per query.

**Proof.** The tree $T$ can be partitioned in $\mu$Trees with maximal depth of $\log^k n$. The partition is accomplished with the sequential dynamic algorithm presented above, inserting nodes of $T$ with an order of insertion defined by a breadth first visit and setting $c_T(n) = \log^k n$. Then the MTree collecting the $\mu$Trees has $n/(\log^k n)$ nodes and the algorithm $A$ requires a total of $O(n)$ time for insertions and $O(\log \log n)$ for the NCA queries. The MicroMacroUniverse approach is applied again to the $\mu$Trees. Each $\mu$Tree $T_m$ is partitioned in $\mu$Trees of depth at most $\log \log n$. The corresponding $\mu\mu$Tree contains $\log^k n/(\log \log n)$ nodes. The pre-process for a $\mu\mu$Tree consists of attaching an $O(\log \log n)$ sized p-list to each node and can be accomplished in time $O(\log^k n/(\log \log n) \log \log n) = O(\log^k n)$. The total cost of pre-processing all the $\mu\mu$Trees, therefore, is $O(n/\log^k n) \cdot O(\log^k n) = O(n)$. The cost of an NCA query on a $\mu\mu$Tree is $O(\log \log n)$ (brute force), on a $M\mu$Tree is $O(\log \log n)$, because the depth of the tree is $O(\log^k n)$, and on the MTree is $O(\log \log n)$ by hypothesis. In Section 6 we showed how to collect the optimal NCA queries for each tree with a constant overhead, to achieve a total cost of an NCA query of $O(\log \log n)$.

**Corollary 2.** The theorem remains true even if $A$ was a Pointer Machine algorithm that made use of constant-time arithmetic for $\log n$ size integers.

**Proof.** A Pointer Machine algorithm with running time $O(t(n))$ can be naively converted to a PPM algorithm with running time $O(t(n) \log^k n)$ for some $k$.

### 7. A parallel compression scheme for trees

The compression algorithm is a sequential iteration of parallel phases. Each parallel phase is composed of two parallel steps. The first step is compression of leaves (leaf compression) in the current tree and the second step contributes to the compression of paths (path compression) in the current tree using a step of pointer doubling [10].
Additionally, our efficient parallel solution for the \(NCA\) problem requires the ability to efficiently solve the \(TP\) problem [21] in parallel. A parallel version of the problem and an efficient parallel solution are presented in Section 7.6.

### 7.1. From sequential to parallel

The direct simulation of the sequential algorithm requires \(O(\log^2 n)\) parallel time. Unfortunately, this direct simulation may also require \(\Omega(\log^2 n)\) time. Consider, for example, the situation in Fig. 12. The tree is composed of a main path, with a number of complete trees (of depth \(k, k-1, \ldots, 1\)) hanging from it. In this situation, at every leaf compression, a path of length \(l\) is created in the main branch, allowing for the next path compression to take place; this path compression will require \(\log l\) time. The process is repeated \(k\) times, hence the total parallel time is \(k \log l\). If \(l\) is chosen equal to \(2^k\), then the total number of nodes \(n = \Theta(2^{2k})\), thus \(k = \Theta(\log n)\) and the parallel time is \(k^2 = \Theta(\log^2 n)\).

We could attempt to improve this running time by allowing path compressions to occur also in the internal paths (i.e., paths that do not end in a leaf); similarly we could allow leaf compression to be performed at all the leaves and heads of paths detected at each parallel step. Unfortunately this will not help our case either. As illustrated by the example in Fig. 13, the \(H\)-tree resulting from these compressions can have linear depth, thus preventing us from using the \(H\)-tree to perform fast computation of NCA queries.

However, these considerations do suggest a possible way to improve parallel running time without losing the efficient computation of NCA queries. The idea is that the scheme should compress all paths present in the tree (even the internal ones), but leaf compressions should not be performed on nodes that are currently not leaves. This idea is translated into a concrete parallel algorithm in the next section.

### 7.2. The algorithm

We start by introducing some notation. For a node \(v\) in \(T\), \(T_v\) denotes the subtree of \(T\) rooted at node \(v\). A parallel phase \(i\) of the algorithm is the sequence of two parallel steps called \(a\) and \(b\), which are executed at parallel time \(i(a)\) and \(i(b)\), respectively. For an integer \(i\), \(T_i\) denotes the tree after the \(i\)th parallel phase. Given a tree \(T_i\), the result of step \(a\) applied to \(T_i\) is the tree \(T^a_{i+1}\) and the result of step \(b\) applied to \(T^a_{i+1}\) is \(T^b_{i}\).

During the processing, nodes in the tree may get marked with the symbol \(L\); if node \(v\) in \(T\) is marked \(L\) at parallel time \(i(a)\) (\(i(b)\)), then we denote this with \(m^a_i(v) = L\) (\(m^b_i(v) = L\)). We will often refer to this marking as \(m(v)\) when the time is clear from the context. If \(v\) is not marked then \(m(v) = ?\). Every node \(v\) in \(T\) has a pointer \(\pi\) to an ancestor of \(v\) at parallel time \(i(a)\) (\(i(b)\)) and we denote it with \(\pi^a_i(v)\) (\(\pi^b_i(v)\)).

A leaf compression of a tree \(T_i\) is executed in step \(i(a)\) and returns a tree \(T^a_{i+1}\) such that for each node \(v\) in \(T_i\) (see Fig. 14):
(i) if \( (m_i(v) = L \text{ and } v \text{ currently has no sibling}) \) then
\[
p_i^{a+1}(v) \leftarrow p_i(\text{parent}(v)) \quad \text{and} \quad m_i^{a+1}(\pi_i^{a+1}(v)) \leftarrow L;
\]
(ii) if \( (m_i(v) = L \text{ and } v \text{ has a sibling } z \text{ and } m_i(z) = ?) \) then
\[v \text{ is merged with its parent } \pi_i^{a+1}(v) \leftarrow \text{NULL};\]
(iii) if \( (m_i(v) = ? \text{ and } (v \text{ has a sibling } z \text{ and } m_i(z) = L) \text{ or } (v \text{ currently has no sibling})) \) then
\[\pi_i^{a+1}(v) \leftarrow \pi_i(\text{parent}(v));\]
(iv) if \( (m_i(v) = L \text{ and } v \text{ has a left sibling } z \text{ and } m_i(z) = L) \) then
\[v \text{ is merged with its parent and } \pi_i^{a+1}(v) \leftarrow \text{NULL};\]
(v) if \( (m_i(v) = L, v \text{ has a right sibling } z \text{ and } m_i(z) = L) \) then
\[\pi_i^{a+1}(v) \leftarrow \pi_i(\text{parent}(v)) \quad \text{and} \quad m_i^{a+1}(\pi_i^{a+1}(v)) \leftarrow L.\]

A path compression of a tree \( T_i^a \) is executed in step \( i(b) \) and returns a tree \( T_i \), such that for each \( v \) in \( T_i^a \), \( \pi_i(v) \leftarrow \pi_i(\pi_i(v)) \) and if \( m_i(v) = L \) then \( m_i(\pi_i(v)) = L \) (see Fig. 15).

If \( T \) is the initial tree, then the tree \( T_0 \) is a copy of \( T \), such that for each \( v \) in \( T_0 \), \( p_0(v) = v \) if \( v \) has a sibling, else \( p_0(v) = \text{parent}(v) \); in addition, for each leaf \( l \) of \( T_0 \), \( m_0(l) = L \). The root is the only exception: \( p_0(\text{root}) = \text{root} \).

Fig. 16 provides an example of a compression. The nodes marked represent the nodes labeled \( L \) and the dashed pointers are the \( \pi \) pointers. The pointers \( \pi \) pointing to \text{NULL} are not shown. The figure also shows the parallel \( H \)-tree (discussed in Section 7.3).

**Definition 6.** A node \( x \) is finished after step \( k \) if one of the following holds:

(1) \( x \) is root and \( m_k(x) = L \);
(2) \( \exists y \text{ } y \text{ is a proper ancestor of } x \text{ and } m_k(y) = L \);
(3) \( \pi_k(x) = \text{NULL} \).

The theorem below provides a result that is critical for establishing the efficiency of the compression scheme.

**Theorem 7.** For each parallel time step \( k \) and for each node \( x \) in \( T \) one of the following holds:

(1) \( x \) is finished before or at the end of parallel step \( k \);
(2) \( x \) is marked \( L \) during parallel step \( k \), it is unfinished after parallel step \( k \), \( |T_x| \geq 2^k - 1 \) and \( |T_{\text{parent}(x)}| \geq 2^k + 1 \);
(3) \( x \) is unmarked and unfinished after parallel step \( k \), and either \( \pi_k(x) = \text{root} \) or \( (|T_{\text{parent}(\pi_k(x))}| - |T_x| \geq 2^k - 1 \) and \( |T_x| \geq 2^k + 1) \).

---

**Fig. 14.** Example of leaf-compression for node \( v \).

**Fig. 15.** Example of path-compression for node \( v \).
Proof.

Base case: \( k = 0 \). If \( x \) is finished after step 0, then \( x \) can only be the root and \( T = x \), since only leaves are marked with \( L \) and for each \( v \) in \( T \), \( \pi(v) = v \). If \( m_0(x) = L \) then \( x \) is a leaf and trivially \( |T_x| \geq 2^{0-1} = 2^{-1} \) and \( |T_{\text{parent}(x)}| \geq 2^0 + 1 = 2 \). If \( x \) is unmarked and unfinished, \( x \) is an internal node in \( T \).

In this case if \( x \) is the root then \( m_0(x) = \text{root} \) else \( |T_{\text{parent}(x)}| - |T_x| \geq 2^{0-1} = 2^{-1} \) and \( |T_x| \geq 2^{0+1} = 2 \).

Inductive step: Let us consider the case \( k \), and let us assume by inductive hypothesis that the results hold for all integers less than \( k \). For every node \( x \) in \( T \), one of the following holds:

1. \( x \) is finished and unmarked after phase \( k \). In this case, the first claim of the theorem is trivially satisfied.

2. \( x \) is marked \( L \) after phase \( k' \leq k - 1 \). In this case, one of the following holds:
   (i) if \( x \) is the root, then \( x \) is also marked \( L \) after phase \( k \);
   (ii) if \( x \) is not the root, then during step \( k' + 1(a) \) \( x \) can be leaf compressed to its parent and \( \pi_{k'}(x) = \text{NULL} \);
   (iii) otherwise, if \( x \) is not leaf compressed this implies that \( x \) had a right sibling \( y \) marked \( L \) after phase \( k' - 1 \).
   
   Thus, \( \pi_{k'}(x) \) is a proper ancestor of \( x \) and it is marked \( L \) after phase \( k' \).

Hence \( x \) is finished after phase \( k' + 1 \) and it is also finished after phase \( k \).

3. \( x \) is marked \( L \) during phase \( k \).

There are two possibilities:

Case 1: \( x \) is marked \( L \) during step \( k(a) \). If \( x \) is the root, then \( x \) is finished after phase \( k \) and the claim holds. Otherwise let \( x' = \text{parent}(x) \). Suppose \( x \) does not have any siblings after step \( k(a) \), then an ancestor of \( x' \) is marked \( L \) after step \( k(b) \) (because of pointer doubling) and \( x \) is finished after phase \( k \). If \( x \) has a sibling \( z \) after step \( k(a) \), then \( z \) could not have been marked \( L \) after phase \( k - 1 \), otherwise it would have been leaf compressed. Therefore by inductive hypothesis (case (3)) \( |T_z| \geq 2^{k-1} \). Also, since \( x \) is unmarked after step \( k - 1 \), \( |T_x| \geq 2^{k-1} \). Moreover, \( |T_{\text{parent}(x)}| = |T_x| = |T_z| + |T_x| + 1 \geq 2^k + 1 \).

Case 2: \( x \) is marked \( L \) during step \( k(b) \). If \( x \) is the root, then it is finished after phase \( k \) and the claim holds. Otherwise let \( x' = \text{parent}(x) \).
Let us consider a node $x$. The root is unfinished after phase $k$.

Proof. Let $n$ be the number of nodes in $T$ and let $k$ be the smallest integer such that the root is finished after phase $k$. From Theorem 7 it follows that $|T| \geq |T_{\text{parent}(x)}| \geq 2^{k-1} + 1$. In other words, the algorithm requires at most $\log(n) + 1$ phases.

Fig. 17. $x$ marked during step $(b)$.

Fig. 18. $x$ unmarked and unfinished after phase $k$.

Suppose that $x$ does not have any sibling after step $(a)$. It follows that $\pi_k(x)$ is an ancestor of $x'$ because $\pi_{k-1}(x)$ was pointing at least to $x'$. Since $x$ is unmarked and unfinished after step $k-1$, there exists a descendent $u$ of $x$ such that $m_{k-1}(u) = L$ and $\pi_k(\pi_k(u)) = x$. Let us denote with $u'$ the node $\pi_k(u)$. Clearly $u' \neq x$, otherwise $x$ would be marked in step $(a)$. Note that $u'$ will be marked $L$ after phase $(a)$. It follows that $\pi_k(u') = \pi_k(\pi_k(u')) = \pi_k(x)$ which is a proper ancestor of $x$—since $x$ does not have siblings after step $(a)$, $\pi_k(u')$ is marked $L$ during step $(b)$. Therefore $x$ is finished after phase $k$.

Suppose instead that $x$ has a sibling $z$ after step $(a)$ (see Fig. 17). Then $m_{k-1}(z) \neq L$ otherwise $z$ would have been leaf compressed at step $(a)$. Therefore by inductive hypothesis (case (3)) we have $|T_z| \geq 2^{k-1}$. Moreover $m_{k-1}(x) \neq L$ and, using the same inductive hypothesis, $|T_x| \geq 2^{k-1}$. Finally $|T_{\text{parent}(x)}| = |T_x'| = |T_x| + |T_z| + 1 \geq 2^{k+1}$.

(4) $x$ unmarked and unfinished after phase $k$.

If $x$ is the root, then $\pi_k(x) = \text{root}$ and the claim holds. If a proper ancestor of $x$ is marked $L$ after phase $k$, then it follows that $x$ is finished after phase $k$ and the claim holds. Otherwise, we have that $x$ is unfinished and $x$ has no ancestors marked $L$ after step $k$. Let $u$ be equal to $\pi_{k-1}(x)$ (see Fig. 18). If $u$ is the root, the $\pi_k(x) = \text{root}$ and the claim holds. Otherwise, let $u'$ be the parent of $u$, and $v = \pi_{k-1}(u')$. If $v$ is the root, then $\pi_k(x) = v = \text{root}$ and the claim holds. Otherwise let $v' = \text{parent}(v)$. By inductive hypothesis (case (3)) $|T_{\text{parent}(\pi_k(u))} - |T_k| \geq 2^{k-2}$ and $|T_{\text{parent}(\pi_k(u'))} - |T_k'| \geq 2^{k-2}$. If $u$ has no siblings or has a sibling marked $L$ after phase $k-1$, then $\pi_k(u) = \pi_{k-1}(u') = v$ (because of the leaf compression) and $\pi_k(x)$ is at least $v$ or above. Then $|T_{\text{parent}(\pi_k(x))} - |T_k| \geq 2^{k-1}$. If $u$ has an unmarked sibling $z$ after phase $k-1$, then by inductive hypothesis (case (3)) $|T_z| \geq 2^{k-1}$. Also, $\pi_k(x) = \pi_{k-1}(x) = u$, then $|T_{\text{parent}(\pi_k(x))} - |T_k| = |T_k'| - |T_k| \geq |T_z| \geq 2^{k-1}$. In $T_x$ there exists a node $y$ such that $m_k(y) = L$ and it is unfinished after phase $k$, $y \neq x$ because $x$ is not marked $L$ after phase $k$. Thus, $x$ is a proper ancestor of $y$ and from the above proof of Case 2 we have that $|T_x| \geq |T_{\text{parent}(y)}| \geq 2^{k+1}$.

Corollary 3. Let $n$ be the number of nodes in $T$ and let $k$ be the smallest integer such that the root is finished after phase $k$. Then $n \geq 2^{k-1} + 1$. In other words, the algorithm requires at most $\log(n-1) + 1$ phases.

If we have $n$ processors that have been assigned to the $n$ different nodes of $T$, then both leaf and path compressions will be performed in constant parallel time. From Corollary 3, the total number of compressions is at most $\log n$, thus the total parallel time required by the algorithm is $O(\log n)$. 
Even though the algorithm has been presented for binary trees, the above results remain true even if we consider trees with arbitrary arity. In this case the leaf compression will change as follows: the last node marked \( L \) among a set of siblings is the only one that is not leaf compressed. If all unfinished siblings are marked \( L \) at the same time, then the node not leaf compressed will be the leftmost one. The proofs in this case remain identical.

7.3. The \( H \)-tree

The \( H \)-Tree built in the parallel scheme is similar to the one described in Section 4. The difference is that, here, each path-compressed branch is extended to the leftmost leaf, instead of being split into more levels by alternation of sequential leaf and path compressions. The topological relation for NCA queries are nevertheless preserved, and thus the same NCA query algorithm can be employed.

It is possible to reuse the \( \pi \) pointers to build \( H \) in constant parallel time. Once a node \( v \) is leaf compressed into its parent, \( \pi(v) \) is set to \( \text{NULL} \). At that time for every node \( w \) in the path having \( v \) as head, we have \( \pi(w) = v \). The goal is to maintain this information in the successive phases, avoiding pointer doubling if a node is finished (line 26 in Fig. 19). Once the compression is completed, every head of a path \( x \) has \( \pi(x) = \text{NULL} \) and the \( \pi \) pointer for every other node \( y \) points to the head of the path containing \( y \).

Let us introduce another pointer \( p_H \), that will be used as the parent pointer in \( H \). During a leaf compression the pointer \( p_H(v) \) is set to point to \( \text{parent}(v) \). Since the root is not leaf compressed, \( p_H(\text{root}) \) is set to point to \( \text{NULL} \). For each head \( x \) of a path \( l \) in \( T \), a new node \( x' \) is created in \( H \), with \( \pi(x') = x' \), \( p_H(x') = p_H(x) \) and \( \pi(x) = x' \). After each step of pointer doubling (applied to the \( \pi \) pointers), every node in a path points to a newly created copy of the head. For every node \( x \) in a path \( p_H(x) = \pi(x) \), and this completes the building of \( H \).

Finally, for each path the auxiliary data structure for the \( TP \) Problem is set up. It is possible to identify the tails of path in \( O(1) \) time as a node \( v \) is a tail iff for all children \( w \) of \( v \) \( \pi(v) \neq \pi(w) \). Given a tail \( t \), the corresponding list is processed as described in Section 7.6. The complete algorithm is presented in Fig. 19. The lines marked with * are the ones necessary to set up the \( H \) tree.

7.4. Answering NCA queries using \( H \)-trees

The \( H \)-tree can be used to answer NCA queries in the same way as in the sequential case. In Section 4.3 we showed that there is a PPM algorithm, given a tree with height \( h \), pre-processes the tree in time \( O(n \lg h) \) and then can compute the NCA of any two given nodes in the tree in worst case time complexity \( O(\lg h) \) per query. The sequential scheme presented in Section 4.3 can be easily translated into a parallel scheme that uses \( n \) processors and \( O(\lg h) \) parallel time for pre-processing. Using this result, we can pre-process the \( H \)-tree in parallel time \( O(\lg \lg n) \) using \( n \) processors. Then, the NCA in \( H \), and hence in \( T \), can be computed in time \( O(\lg \lg n) \) using a single processor.

7.5. Discussion

The algorithm described above clearly can be directly implemented on a CRCW Parallel PPM. A problem arises if we were not allowed concurrent writes, because too many processors may attempt to update the \( L \) mark of the same node in the tree at the same time (e.g., line 10 in Fig. 19). This will not be allowed in the CREW/EREW/CROW parallel pointer machines. This is also not allowed in the Parallel PPM (as described in Section 2) because it would correspond to an unbounded fan-in. However, it is possible to modify the algorithm to overcome this problem. This is essentially obtained by concurrently performing a pointer doubling in the reverse direction along the branches of the tree. More precisely, each node \( u \) of the tree maintains a pointer \( \pi_{\text{down}}(u) \) which is updated to point to \( \pi_{\text{down}}(v) \) whenever the node has only one child \( v \). In addition, if \( \pi_{\text{down}}(v) \) is marked \( L \), then \( u \) will mark itself \( L \) as well. With this addition, the fan-in of each unit is restricted to be finite. Moreover, the algorithm still requires only \( O(\lg n) \) parallel phases. Hence, the algorithm can be modified to correctly work on Parallel PPMs.

The algorithm requires \( n \) processors to perform the \( O(\lg n) \) parallel time pre-processing. After pre-processing, a single processor can answer an NCA query in time \( O(\lg \lg n) \). It is interesting to compare this result with the other parallel algorithms proposed for the NCA problem. The best known PRAM algorithms require \( O(n/\lg n) \) processors and work in \( O(\lg n) \) parallel time for pre-processing. After pre-processing, a single processor can answer an NCA query in time \( O(1) \). Hence, going from a PRAM to a Parallel PPM we incur a penalty of \( O(\lg n) \) in number of processors.
and total time taken, and a penalty of $O(\lg \lg n)$ time to answer a query. Observe that we do not incur any penalty in parallel time for pre-processing.

It is also important to observe that if we have any CROW PRAM NCA algorithm which solves the problem in parallel time $O(\lg n)$ with $f(n)$ processors and answers a query in $O(1)$ time, then for a generic translation of this algorithm to a Parallel PPM algorithm (as illustrated in [9]) one can only claim that it requires parallel time $O(\lg n \lg \lg n)$ with polynomially many processors, and answers an NCA query in time $O(\lg \lg n)$. Hence, the algorithm presented here is substantially better than a generic translation of any PRAM NCA algorithm presented in the literature to date to a Parallel PPM algorithm.

It is interesting to note that if we have simple arithmetic capabilities (actually only constant-time addition is needed), then we can compute the centroid path and the $H$-tree based on it in $O(\lg n)$ parallel time. This is obtained by keeping a count of the number of nodes in the subtree rooted in each node during the algorithm execution. Each time we have a leaf compression phase where both children of a node are marked $L$, instead of leaf compressing the right child, we leaf compress always the child with a smaller count. It is easy to show that this will build the centroid path tree.

![Fig. 19. Pre-processing algorithm.](image-url)

```plaintext
1: pardo
2: if (v=root or v has a sibling) $\pi_0(v) := v$ else $\pi_0(v) := parent(v)$;
3: if (v is leaf) $m_0(v) := L$ else $m_0(v) := ?$;
4: i:=0;
5: repeat
6: // leaf compression
7: pardo
8: if ($m_i(v) = L$ and v has currently no sibling)
9: $\pi_{i+1}(v) := \pi_{i}(parent(v))$;
10: $m_{i+1}(\pi_{i+1}(v)) := L$;
11: else if ($m_i(v) = L$ and v has a sibling z and $m_i(z) =$?)
12: $\pi_{i+1}(v) := NULL$;
13: $p_H(v) := parent(v)$;
14: else if ($m_i(v) =$? and ((v has a sibling z and $m_i(z) = L$) or (v currently has no sibling)))
15: $\pi_{i+1}(v) := \pi_{i}(parent(v))$;
16: else if ($m_i(v) = L$ and v has a left sibling z and $m_i(z) = L$)
17: $\pi_{i+1}(v) := NULL$;
18: $p_H(v) := parent(v)$;
19: else if ($m_i(v) = L$ and v has a right sibling z and $m_i(z) = L$)
20: $\pi_{i+1}(v) := \pi_{i}(parent(v))$;
21: $m_{i+1}(\pi_{i+1}(v)) := L$;
22: else $\pi_{i+1}(v) := \pi_{i}(v)$;
23: // path compression
24: pardo
25: $\pi_{i+1}(v) := \pi_{i+1}(\pi_{i+1}(v))$;
26: if ($\pi_{i+1}(v) = NULL$) $\pi_{i+1}(v) := \pi_{i+1}(v)$;
27: if ($m_{i+1}(v) = L$) $m_{i+1}(\pi_{i+1}(v)) := L$;
28: i:=i+1;
29: until $m_i(root) = L$;
30: * $p_H(root) := NULL$;
31: * pardo x head of a path
32: * create $x'$ and $\pi(x') := x'$; // $x'$ is a copy of $x$
33: * $p_H(x') := p_H(x)$;
34: * $\pi(x') := x'$;
35: * pardo $p_H(x) := \pi(\pi(x))$; // Each node in the path of $x$ points to $x'$
36: * pardo if (v is tail) TP pre-process on the list starting at v and ending at $\pi_i(v)$
```
7.6. A parallel algorithm for the T P problem

The T P Problem, first defined in [21], can be reformulated in the context of parallel computations as follows: given a list L with l nodes representing an ordered sequence of objects, we want to answer the query \texttt{precedes}(x, y), where x, y are pointers to nodes in that list. We present a solution to this problem on Parallel PPMs that requires O(lg l) parallel pre-processing time, and O(lg lg l) time to answer each query using a single processor thereafter.

The basic idea is to create an auxiliary complete binary tree BT, such that each leaf is assigned to an element of L. If BT maintains a left to right ordering in each level, then the \texttt{precedes}(x, y) query can be answered comparing the children of \texttt{nca}(x, y) in BT. We maintain this order in each level of BT using \texttt{sibl} pointers. BT is constructed via a parallel level-by-level construction. During the construction, each node of BT has one processor associated to it. The root of BT is created in the first step. Then, in parallel (for lg l steps), each new processor p associated to a node v in BT executes the following operations: create two new nodes \((v_l,v_r)\) with new processors associated to them, set \texttt{sibl} \((v_l,v_r)\) to \texttt{left child of sibl}(v).

The last level of BT contains a list of nodes S. Since L is the input, from the way inputs are presented in the Parallel PPM model, we can assume that active processors can be assigned to each element of L in time O(lg l). We can also assume that these processors have pointers that point to the \texttt{previous} element in the list (see Fig. 20). The elements of the original list L are mapped to the elements of S in O(lg l) parallel time with O(l) processors using a pointer doubling scheme, which modifies the sibling list of S and \texttt{previous} pointers of L. Each node of L contains a pointer map that is used to point to the corresponding node in S. Initially none of the map pointers is set. In the first step processor assigned to the head of L sets its map pointer to the head of S. At the same time, the second element of L sets its map pointer to the second element of S (see Fig. 21). This is followed by a step of pointer doubling in both S and L. In S pointer doubling is accomplished using the \texttt{sibl} pointers, while in L it is performed using the \texttt{previous} pointers. After a step of pointer doubling, if the previous pointer of a node v in L whose map pointer is not set points to a node u whose map pointer is set, then map(v) is set to sibl(map(u)). Note that the map pointer of a node in L is set only once. The process continues until all the nodes in L have their map pointers set. The whole process requires O(lg l) parallel time.

The last step of the pre-processing constructs in O(lg l) parallel time the auxiliary data structures (called p-lists) using a straightforward parallelization of the algorithm presented in Section 4.3. A \texttt{precedes}(x, y) query is then answered by a single processor in O(lg lg l) time using the algorithm presented in Section 4.3.

Note that if we are allowed only one processor to answer an NCA query, then the time required must be at least \(\Omega(lg lg n)\) [21]. Hence our algorithm is optimal in that regard. Observe also that the parallel time O(lg n) used to perform pre-processing is the best known for any parallel NCA algorithm (including PRAM algorithms). If one were allowed arbitrary (e.g., \(n^2\)) number of processors, then it is possible to devise a Parallel PPM algorithm that requires O(lg n) parallel time for pre-processing and answers NCA queries in time O(1) [9]. This can be simply accomplished by pre-computing all the answers in parallel (in time O(lg lg n)) and making a different processor responsible for each different possible query.
8. Conclusions and future work

We have defined a novel compression scheme and used it for solving the NCA problem optimally on PPMs both in the static and the dynamic case. The compression scheme is interesting due to its simplicity, locality properties, efficiency and arithmetic-free nature. However, it is not essential for obtaining the optimal NCA algorithm for the PPMs due to the following remarkable theorem making use of the MicroMacroUniverse scheme presented in Section 6. We have also shown that for the NCA problem, it is possible to totally avoid the polylog penalty that one has to incur in a generic translation of an algorithm designed for APMs to PPMs. This gives rise to the question: Is there any natural problem for which the optimal solution on PPMs is provably logarithmically worse as compared to the optimal solution on APMs. As of now, we believe that the worst such known penalty incurred is $O(\lg \lg n)$ [21]. It will be especially interesting if there is no problem at all where the logarithmic penalty has to be incurred because that will show that the generic translation is non-optimal.

We also presented an efficient parallel pointer machine algorithm for the NCA problem for trees in the static case. Our algorithm requires $O(\lg n)$ parallel time and $O(n)$ processors for pre-processing where $n$ is the number of nodes in the tree. Thereafter, it can answer any NCA query in $O(\lg \lg n)$ time. Our NCA algorithm required an efficient parallel solution of a parallel version of the Temporal Precedence problem [21]. We provided an efficient parallel pointer machine algorithm to solve this problem.

Acknowledgments

We would like to thank the anonymous referees for their comments. The research has been partially supported by NSF Grants EIA-0220590 and HRD-0420407.

References


