UNIVERSITA' DEGLI STUDI DI VERONA

FACOLTA' DI SCIENZE MM FF E NN

Corso di Laurea in Informatica

TESI DI LAUREA

Alessandro Dal Palú

Matricola IN000438

New Optimal Algorithms on Pointer Machines

Relatore R. Giacobazzi

Co-Relatori A. Dovier

E. Pontelli

D. Ranjan

ANNO ACCADEMICO 2001-2002

Verona, 10 Luglio 2002

Contents

Chapter 1

Introduction

In this thesis we study the lower bound complexity of two data structural problems arising in Computer Science and new methods matching that lower bounds as well. This thesis presents a collection of new optimal Pointer Machine algorithms for the resolution of the *Or-Parallelism Problem* (OP-Problem) and the *Nearest Common Ancestor Problem* (nca Problem). Those problems have been intensively studied in the past with computational models like the Random Access Machines and already optimal algorithms have been discovered. Even though RAM is the most commonly used reference model in studies of complexity of sequential algorithms, the *Pointer Machine* models have received increasing attention as viable model to study complexity of algorithms. The Pointer Machine model is particularly advantageous for complexity studies involving finer level of details [56]. Many lower bound results have been developed using the Pointer Machine model [6, 35, 56]. Moreover, the naïve translation of the existing RAM algorithms or more refined versions so far proposed does not match the optimal running time. The most obvious question would be about the reason that lead us to research for new algorithms with this new model. To answer this question we need to briefly describe the Pointer Machine models.

The Pointer Machine model is a computational model that provides a good base for modelling implementation of linked data structures, like trees and lists. It is simpler than other commonly used models, thus making it more suitable for analysis of lower bounds of time complexity [6, 35, 47, 56]. A simple formalization can be outlined describing the memory as a collection of records. Each record can point to other records or store some kind of alphabet. The only way to reach a specific record is following the path (performing repeated memory accesses) between a given entry point and the desired record. The Pointer Machine model is a restricted version of a RAM: the relevant difference is that there is no direct access to the memory. This limitation suggests that a simple operation will cost more in terms of memory accesses.

Later we will also refer to the *Pure Pointer Machine* (PPM) model, which applies another restriction to the Pointer Machine model: there is no support for the representation or management of numbers. Every number or arithmetic operation has to be coded and, thus,

to be accounted for while analyzing the computational costs. RAM commonly hides the actual cost of arithmetic operations, by allowing operations on numbers of size up to $\lg n$ (n being the size of the input to the problem) to be treated as constant time operations (*uniform cost assumption*). Pure Pointer Machines instead make these (relevant) costs explicit. Indeed, one of the major characteristics of PPMs is the lack of arithmetic. Numbers have to be explicitly represented, e.g., as linked lists of bits. This is realistic in the sense that, for arbitrarily large problems requiring arithmetic operations, the time to perform each arithmetic operation will be a function of the size of the problem. Note that the PPM model is close 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.

In this framework we decided to study new optimal algorithms for the OP-Problem and nca Problem. The first problem arises in the development of a parallel implementation of non-deterministic programming languages and systems (e.g., logic programming, constraint programming, search-based AI systems). The most important issue is the dynamic management of the binding environments — i.e., the ability to associate with each parallel computation the correct set of bindings/values representing the solution generated by that particular branch of the non-deterministic computation. The problem has been abstracted and formally studied previously [25, 42], but to date only relatively inefficient data structures [26, 41, 42] have been developed to solve it. Intuitively this problem can be imagined as a tree problem where some nodes are marked and a query can be described as finding the lowest marked ancestor of a given node.

In [41, 42] it was proved that the problem has a lower bound time complexity of $O(\lg n)$ per operation on Pointer Machines where n is the number of nodes in the dynamic tree. A new data structure that allows the problem to be solved with a worst-case time complexity $O(\sqrt[3]{n})$ per operation was also presented there. This was a significant improvement over the solutions used in the implementations of or-parallel logic and constraint programming systems, as well as binding management in object-oriented systems, which typically incur a worst-case cost of $\Omega(n)$ per operation.

We provide an efficient and simple solution to the OP-Problem. This solution relies on the use of simple data structures (AVL trees and generalized linked lists) and has a worst-case time complexity $O(\lg n)$ per operation on both RAM and Pointer Machines (with unit-cost arithmetic), where n is the number of nodes in the tree. In particular, in the case of Pointer Machines this data structure is *optimal*. We also show how the data structure can be used to provide a $O(\lg^2 n)$ solution on Pure Pointer Machines. By reducing the worst-case time complexity from polynomial to polylog, these solutions provide an *exponential improvement* over the previous best-known solutions (e.g., the $\tilde{\Theta}(\sqrt[3]{n})$ solution presented in [42]) to the problem. In addition, the solution provides further indication that the existing data structures adopted to solve this problem in practice (e.g., concurrent logic and constraint programming systems) are not optimal and can be improved. We also propose a generalization of the problem to handle multiple occurrences of attributes along

each branch of the tree. This generalization is necessary when the OP-Problem is used to model the type of search arising in the context of Object Oriented Systems supporting late binding [41]. In this thesis we show that even this generalized problem can be solved with the same time complexity ($O(\lg n)$ per operation).

We briefly describe the second problem studied in this thesis. The nca is a structural problem, where given a tree and two nodes in it, one may be asked to find the lowest node which is parent of both given nodes. For this problem we consider the static case where we suppose the original tree is known in advance as well as queries. We analyze then the dynamic case, where the tree is built during the running time and queries can be asked at any time. Finally we move on the parallel static case (which can be easily extended to the parallel dynamic case), where the tree is known and many processors can elaborate simultaneously the information. To achieve our results we map the problem on a generic tree in a $\log n$ depth tree problem, where n is the number of nodes in the original tree. We show that the mapping preserves the needed information: it is possible to answer a query using the result of an nca query over the second tree. This second task has already been shown in [44]. We recall how to compute efficiently an answer on Pointer Machines if a $\log n$ depth tree is provided. Hence, our aim is to find the appropriate mapping obtained with a tree compression scheme and how to compute the answer on the original tree.

We present a simple, arithmetic-free, efficient scheme to compress trees maintaining the nca information. We use this compression scheme to provide an $O(n + q \lg \lg n)$ solution for solving the nca problem on *Pure* Pointer Machines 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. Using the nca result, we also provide improved and/or optimal solutions to several other problems including the generalized linked list order maintenance problem on PPMs. Moreover, our nca algorithm has the same complexity as an optimal solution for the problem on Pointer Machines that *do* allow constant-time arithmetic operations [2]. Hence, our result shows that use of arithmetic is not essential for solving the nca problem optimally. This is pleasing because intuitively nca is a structural and not an arithmetic problem.

We also present an efficient parallel compression scheme for trees. This 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 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 [47]. We provide an efficient Parallel Pointer Machine algorithm to solve this problem as well.

Interestingly, we show for the nca problem that any optimal Pointer Machine algorithm can be translated in a PPM algorithm without incurring any penalty in the running time.

This thesis is organized as follows: Chapter 2 presents an excursus over the Pointer Machine model. Section 2.1 briefly describes the Pointer Machine model and how it com-

pares to other models of computation. Section 2.2 recalls how to analyze the complexity of a Pointer Algorithm. In Sections 2.3 and 2.4 the models of Pointer Machine (Pure and Parallel, respectively) here assumed are defined.

Chapter 3 presents the OP-Problem: Sections 3.1 and 3.2 introduce the problem. In Section 3.3 it is given a formal definition of this problem. Section 3.4 presents the already proved lower bounds. In Section 3.5 we revise the known solutions to this problem. In Section 3.6 is presented our first solution to the problem. Sections 3.7 and 3.8 show the complexity of the solution presented above assuming the Pointer Machine and PPM computational models. Section 3.9 introduces the multiple label occurrences extension to the OP-Problem. Finally, in Section 3.10 we provide our optimal solution for the extended problem.

Chapter 4 presents the nca Problem: Section 4.1 introduces this problem and describes the state of the art. Section 4.2 describes our solution and the results we achieved. Section 4.3 presents the novel compression scheme for trees and how it can be used for nca calculations. Section 4.4 describes the optimal nca algorithm for the static case. Section 4.5 describes some applications of an optimal nca algorithm. Section 4.6 describes the optimal nca algorithm for the dynamic case. The dynamic algorithm presented assumes that the tree is modified using only *add-leaf* and *delete* operations although the extension to allow *link* operations is straightforward. In Section 4.7 we introduce the parallel version of the problem and discussing why straightforward parallelizations of the compression scheme fail. In Section 4.8 we present our parallel compression scheme and in Section 4.9 we finally describe our parallel algorithm. We prove that our algorithm works correctly and that it requires $O(\lg n)$ parallel time. In Section 4.10 we compare and contrast our algorithm with other parallel nca algorithms. We conclude this Chapter with an interesting result presented in Section 4.11 : we prove that any optimal Pointer Machine algorithm for the nca problem using arithmetic can be translated to a PPM algorithm without incurring any penalty in the running time.

In Chapter 5 we provide some implementation details: in Section 5.1 and followings we explain how to efficiently solve the nca Problem on a PPM. Our nca algorithm requires an efficient parallel solution of the temporal precedence problem [47]. In Section 5.4 we present a parallel version of the Temporal Precedence Problem. To our knowledge, this is the first proposed solution to this problem.

In Chapter 6 we conclude this thesis and summarize the work presented.

Chapter 2

Pointer Machines

2.1 Introduction to Pointer Machines

Abstract machines were introduced in Computer Science to formalize the intuitive notion of *algorithm*. The most popular examples of such models are the Turing Machine and the RAM [59, 61]. Usually these machines make use of media for input and output, conventionally described as tapes on which symbols from a finite alphabet Σ are written. Thus, algorithms defined over these models compute functions from Σ^* to Σ^*. It is possible to compare the power of various models by comparing the complexity of the same problem implemented on different machines or comparing the complexity of simulating one model by another.

The differences between the models are usually the *storage structure* used and the set of operations available to form a *program*. It is often desired that the internal operations of a machine have a *discrete* nature as argued by Kolmogorov and Uspenskii [34]:

> The mathematical notion of Algorithm has to preserve two properties...
>
> 1. The computational operations are carried out in discrete steps, where every step uses a bounded part of the result of all preceding operations.
> 2. The unboundedness of memory is only quantitative: i.e. we allow an unbounded number of elements to be accumulated, but they are drawn from a finite set of types, and the relations that connect them have limited complexity.

Schönhage referred in [52] to abstract machines that have these properties as *atomistic*.

The form of storage suggested by Kolmogorov and Uspenskii, as well as by Schönhage, is a collection of *nodes* that are interconnected by *pointers*. All machine models that have this type of storage structure can be described as *atomistic Pointer Machines*. All models of Pointer Machines share the common characteristic of disallowing indexing into an array (i.e., pointer arithmetic), as opposed to RAM models.

2.1.1 Kolmogorov-Uspenskii Machines

The Kolmogorov-Uspenskii model (KUM) [34] represents storage as an undirected finite graph (see Figure 2.1), in which every edge has a *label* chosen from a finite set and edges incident to the same node must have different labels. This implies a finite bound on the degree of the graph. At any moment during the computation, one node is designated as the *active node*. The neighborhood of the active node, defined by some fixed radius, is called the *active zone*. In the original formulation, each step of the machine consists of applying a fixed transformation that maps every possible form of the active zone into a subgraph that has the same boundary. For example, the active zone can be contracted, with the effect that nodes that were previously too far are pulled inside for inspection. A more conventional programming formulation, which serves to bring this model under the common framework described above, defines a program to be a sequence of instructions that include the following types: input, output, unconditional and conditional branch, and storage modification. The conditional branch instruction specifies two strings of labels, not longer than the radius of the active zone; the destination of the branch is determined by whether the two paths starting at the active node, and specified by the given labels, lead to the same node. A storage modification instruction can add a node, remove one, add or remove an edge.

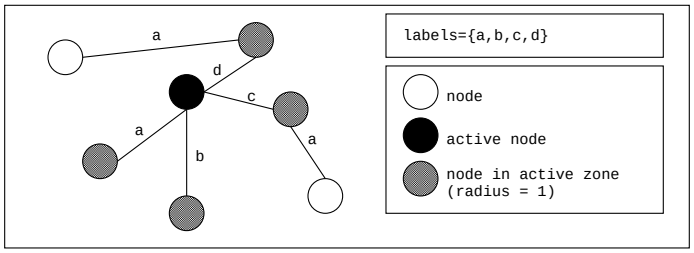

Figure 2.1: An Example of Storage on KUMs

2.1.2 Storage Modification Machines

The Storage Modification Machines (SMM) model, introduced by Schönhage, differs from the KUM by representing storage as a directed finite graph (see Figure 2.2); apart from that the description in the last paragraph applies. Schönhage did not include the requirement of a fixed radius for the active zone, but mentioned that this change may give a "more precise and realistic" measure of running time. In contrast with KUMs, here the finite size of the set of labels only restricts the *out-degree* of nodes; an unbounded number of pointers may lead to a single node. Observe that using a set of two distinct labels, i.e. nodes of out-degree two, any algorithm using a larger out-degree can be simulated with at most a constant factor loss in running time as well as in the number of nodes in storage. This is

true for both the SMM and the KUM.

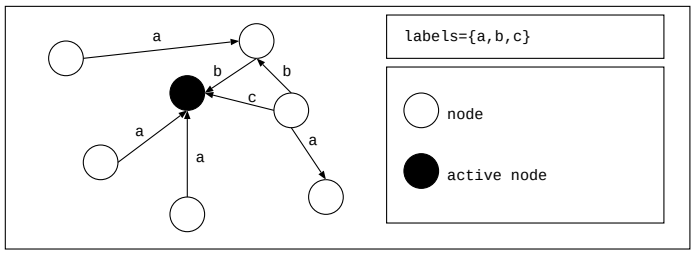

Figure 2.2: An Example of Storage on SMMs

2.1.3 Knuth's Linking Automaton

The Linking Automaton (KLA) was defined by Knuth [33] as a model that may help to understand better the capabilities of algorithms that operate on linked structures. It is defined the same as the SMM except that every node has, in addition to the fixed number of out-going pointers, also a fixed number of *value fields*. Each value field stores one symbol out of a given alphabet (see Figure 2.3 for an example). The program has the capabilities of moving symbols around and of comparing value fields for equality of contents.

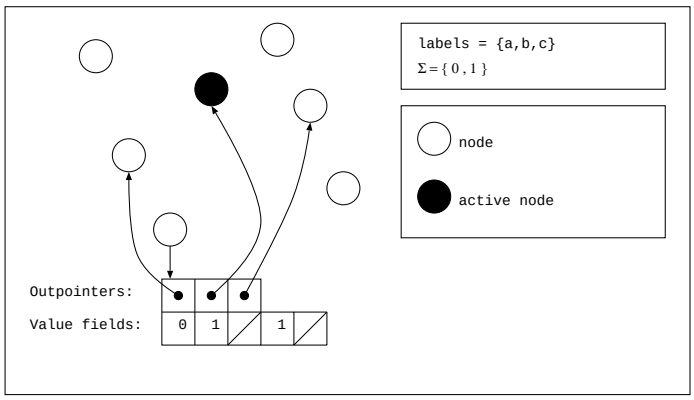

Figure 2.3: An Example of Data Structure for a Node in KLAs

2.1.4 Generalized Atomistic Pointer Machines

Shvachko [53] suggested to study machines which operate on *graphs*, under different specifications of the class of allowable graphs. He mentions the KUM and SMM as particular

examples, where the former is obtained by selecting undirected graphs of bounded degree and the latter by selecting directed graphs of bounded out-degree. In addition to these models, he considers "tree pointer machines". These are naturally obtained by restricting the graphs to be bounded-degree trees. We remark that Shvachko applies the original programming style of Kolmogorov and Uspenskii for all the models he considers.

2.1.5 Jones' I Language

An interesting observation on the "abstract machine" way of formalizing algorithms, is that once a programming language is defined, with proper semantics and cost functions (to evaluate complexity), a computational model has been fully specified. In a model thus obtained, the notion of *algorithm* is identified with that of a *program* and not of *machine*.

Jones [32] took this approach, and presented two programming languages, I and I^{su}, that can be used as general computational models. He used denotational semantics; however, the natural choice of a model for operational semantics is, in the case of I^{su}, an SMM of out-degree two. This makes the I^{su} model a programming-oriented alternative to the SMM. It only differs by the restriction of the out-degree (which could be easily relaxed), and the fact that the I^{su} language is structured, while the SMM is programmed using `goto`s.

The I language is a restricted version of I^{su}: in this language, the destination of a pointer may only be set when the node it emanates from is created. In programming terms, it has a `cons` instruction but not a `setcar` or `rplaca` instruction (according to LISP dialects). This results in the memory graph being always *acyclic*. It is widely believed that this makes the model weaker in an asymptotic sense, a conjecture that has not yet been proved.

2.2 Complexity of Pointer Algorithms

In the study of Pointer Algorithms, problems we are interested in may be generally described as *data-structure problems*. A common representation for such structures is a directed graph, implemented as a network of records in memory with edges represented by pointers from one record to another. We assume that our data structure is the one of the above kind, without entering in implementation details. The *complexity* of performing data-structure operations is measured by the number of modifications or accesses to the graph. Thus we define the class of *pointer algorithms* as algorithms for performing some given data structure operations on a graph representation of the structure. Clearly, the concrete relationship of the graph to the represented structure has to be defined specifically for each problem. In general this relationship should be of such nature that the data-structure operations accept pointers to nodes in the graph as input and deliver their output in form of such pointers. To measure the *time complexity* of a pointer algorithm we consider the number of edges followed by the algorithm, i.e., the length of the shortest path leading from the set of input nodes up to every member of the set of output nodes. We also count the number of edges added or removed during the operation. Note that, while a particular algorithm

may traverse the same edge more than once, we only count the number of edges used. This may underestimate the cost of an algorithm, but this is inherent in the use of this kind of model, and it turns out that tight lower bounds are nonetheless obtained for various problems. Thus the Pointer Machine model is a good tool for the study of lower bounds. *Space complexity* is sometime measured by the number of nodes in the graph.

A more precise way of defining this model is as an implementation of a given *abstract data structure* by means of the abstract data structure *directed graph*. This structure provides a well-defined set of operation: add node, add pointer, get pointer and delete pointer. The time complexity of the implementation is defined by the number of these operations and the space by the number of *add node* operations.

We now compare the pointer-algorithm model with the abstract machine approach to computational complexity of problems. An essential characteristic of the abstract machine approach is the machine-independent representation of the problem, as a function over strings, integers etc. This machine-independence allows us to ask for the complexity of the same problem on different models and, in fact, to compare different models for their power. To contrast, in the pointer-algorithm model the problem definition is all but machine-independent: it makes use of pointers, an entity which has no external representation. The fact that input and output are given as pointers is essential to the model [3].

2.3 Pure Pointer Machines

A *Pure Pointer Machine* (PPM) is a restricted version of a Pointer Machine, that does *n*ot allow constant time arithmetic operations. The Pure Pointer Machine model here assumed is essentially the Linking Automaton model and is a representative of the *atomistic Pointer Machine model*.

We now provide a more detailed definition of this machine. 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, i.e., they all contains the same number of fields. Each field may contain either a *data* or an address. The PPM is also supplied with two finite collections of registers, $d_1, d_2, \ldots$ (data registers) and $r_1, r_2, \ldots$ (pointer registers). Each register d_i can contain a data element, while each register r_i can contain 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.

2.4 Parallel Pointer Machines

As with sequential Pointer Machine model, 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. 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 *CRCW Parallel Pointer Machine* [23], 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 [11]. 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 thought as taps on other processors' outputs. The usual Parallel PPM model allows for unbounded fan-out but only constant fan-in. Each finite state machine has the ability to change its links in a restricted way: in particular, 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 constitute a Parallel PPM can be found in [11]. 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 CROW (Concurrent-Read Owner-Write model), and the SIMDAG model with its variants [22]. Several interesting results regarding their computational power have been established. In particular, an $n-$processor CROW PRAM running in time $O(\lg n)$ can be simulated by a Parallel PPM in time $O(\lg n \lg\lg n)$ using polynomially many processors. In addition any step-by-step simulation of an n processors CROW PRAM by a Parallel PPM requires time $\Omega(\lg\lg n)$ per step [17].

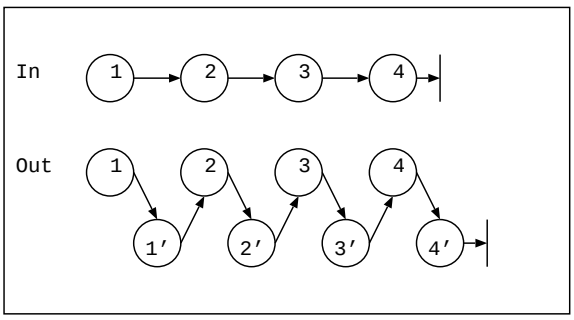

Figure 2.4: An Example of PPM Program

2.5 An Example of Pointer Algorithm

We provide an example of a simple pointer algorithm (see Figure 2.4). Let In be a list of nodes. The pointer l to the head of the list is given as input to the program. The algorithm returns a pointer r to a list Out such that $|Out| = 2|In|$. We assume that a node n can store a pointer p to another node m ($m = n.p$ in a more compact form). We also assume that the tail of the list is pointing to nil. A possible implementation could be:

```
1: head=l;
2: while (l ≠ nil)
3:     temp:=new_node();
4:     temp.p=l.p;
5:     l.p=temp;
6:     l=temp.p;
7: loop
8: return head;
```

Note that the variables can contain only nodes and/or nil. The function $new\_node()$ allocates a new node in the memory.

Chapter 3

The OP-Problem

3.1 Introduction to the OP-Problem

Non-determinism arises in many areas of computer science. Artificial intelligence and constraint-based optimizations are two such areas where non-determinism is commonly found. Non-determinism naturally arises when there are multiple (potential) solutions to a problem. For instance search problems, generate-and-test problems and constrained optimization problems. Non-determinism is incorporated in many programming languages: logic programming languages (e.g., Prolog), constraint programming languages (e.g., Chip [60]), concurrent constraint languages (e.g., AKL [48]), and rule-based languages (e.g., OPS5 [7]) being some of the salient examples.

Non-determinism present in a problem offers a rich source of parallelism. A problem is usually expressed as a *goal* to be achieved/proved/solved together with *rules* (or *clauses*) that specify how a given goal can be reduced to other subgoals. Given a (sub-)goal, there may be multiple ways of reducing it (non-determinism). Translating non-determinism into concurrent execution leads to what is commonly referred to as *Or-parallelism* [26]. Or-parallelism has been shown as a practical and effective methodology to provide transparent parallelization from the execution of non-deterministic systems, leading to speedups on large classes of applications (e.g., [26, 39]).

The OP-Problem represents an abstraction of a key problem that occurs in the development of or-parallel implementations of non-deterministic programming languages, such as logic and constraint programming languages. The problem is essentially dynamic management of binding environments during a parallel execution, where each active processor needs to maintain a correct view of the variable bindings produced along its branch of the computation tree [26]. This problem—commonly referred to as the *binding environment problem* in the logic programming literature—has been shown to be the major source of complexity in the development of an or-parallel implementation. The analogous problem can be observed in parallel implementations of search-based systems (e.g., [18, 39]).

In [42] it was demonstrated that the binding environment problem that arises in logic, constraint, and search-based computations can be abstractly described as a dynamic tree data structure problem, where the computation is naturally abstracted as a dynamic tree and bindings are abstracted as attributes attached to the tree nodes. In [41] it was also demonstrated that a similar abstraction originates from the management of binding between attributes and their definition in object-oriented systems, leading to a very similar abstract problem. This single abstract data structure problem has been named the OP-Problem.

3.2 Applications to Parallel Logic Programming

The OP-Problem was originally proposed in [40, 42]. In that context, the OP-Problem represents a correct abstraction of the problem of dynamically handling binding environments during an or-parallel execution of *logic programming* languages (e.g., Prolog).

An or-parallel execution can be described as the dynamic development of a tree, called the *or-tree*. Each node contains a goal obtained from the computation. The root node is labelled with the initial goal. During a Prolog execution, the leftmost subgoal in the goal of each node is selected and the matching program clauses are found. For each matching clause, a child node is created. If $B_1, \ldots, B_n$ is the goal at the node, for each clause $H_j : -D_j^1, \ldots, D_j^m$ such that θ_j is the most general unifier of H_j and B_1, a child node labelled with the goal $(D_j^1, \ldots, D_j^m, B_2, \ldots, B_n)\theta_j$ is created [36]. Note also that the child node v_i corresponding to a matching clause C_i that textually precedes another matching clause C_j is placed to the left of v_j, where v_j is the child node corresponding to C_j. Sequential execution corresponds to building the or-tree one node at the time, in depth-first order. Or-parallel executions correspond to concurrently growing the or-trees at multiple sites (nodes).

In the or-tree, each branch maintains a local view of the substitution computed. This is essential because, during a reduction like the one mentioned above, the substitutions θ_j produced by unification may potentially conflict and must be kept separate in different branches in the or-tree. For example, if a node containing a goal `p(X)` is expanded into two branches due to the two matching clauses,

```
p(1) :- ....          p(2) :- ....
```

then the variable `X` receives a different binding from each of the two clauses. These bindings need to be represented and maintained separately, as they will each lead to a different solution (`X=1` and `X=2`) for the initial goal. `X=1` and `X=2` represent different instances of `X` and they need to be separately maintained and properly associated to the correct thread of execution (i.e., the correct branch in the or-tree).

One of the main issues, thus, in designing an implementation scheme capable of efficiently supporting or-parallelism is the development of an efficient way of associating the correct set of bindings to each branch of the or-tree. The naïve approach of keeping θ_j for each branch is highly inefficient, since it requires the creation of complete copies of the substitution

(which can be arbitrarily large) every time a branching takes place [26, 42].

The abstraction of an or-parallel execution should account for the various issues present in OP-Problem, (e.g., management of variables and of their bindings, creation of tasks etc.). Previous research, as well as development of practical implementations, [1, 26, 37] have demonstrated that variable management is indeed one of the key issue in an or-parallel system. Actual or-parallel systems have experimentally demonstrated the impact of variable management [54]; furthermore, concurrent exploitation of and/or-parallelism exacerbates the problem [24, 27, 38, 63], making variable management one of the basic problems, if not the fundamental one.

The development of the or-parallel execution (creation of tasks, execution of resolution steps) can be easily abstracted as the construction and management of a tree using the create_tree, expand, and remove operations. Variables that arise during execution, whose multiple bindings have to be correctly maintained, can be modelled as attributes of nodes. Binding a variable X in the goal represented by node u can be directly abstracted using the operation $\mathsf{assign}(u) = X$. Similarly, retrieving the value of the variable X when solving the goal in node u can be abstracted using the operation $\mathsf{dereference}(X, u)$. In [42] it is shown that, to complete the abstraction of or-parallelism, we need an additional operation:

- alias two attributes X_1 and X_2 at node u; this means that for every node v such that $v \succeq u$ every reference to X_1 in v will produce the same result as X_2 and vice-versa.

This operation is needed to abstract Prolog's ability to alias unbound variables. Nevertheless, as discussed in [42], the addition of alias to the OP-Problem does not modify the problem's complexity—the alias operation can be efficiently implemented using union-find techniques [35].

In the previous abstraction we have assumed the presence of one variable binding per node. This restriction can be made without loss of generality (we can always assume that the number of bindings in the node is bound by a program dependent constant). It is also important to observe that in this framework we are assuming that each attribute is assigned at most once along each branch of the tree—this models the single-assignment nature of bindings in logic and constraint programming. This abstraction has been argued to be correct in [40].

Observe that the similar situation would also occur in any parallel execution where the management of the binding environment is dynamic. For example, a parallel execution of a functional language (e.g., Lisp) making use of dynamic scoping would encounter analogous problems and require analogous solutions. The same problem occurs in the management of search-parallelism in the context of constraint programming [50] and in a variety of AI applications [18, 39].

It is important to underline that the ability to handle the dynamic binding environment is the single most important aspect in the efficient development of a parallel search computation.

3.3 Formal Definition of the Problem

Given a direct graph $G = \langle N, E \rangle$, where N is the set of all the nodes in the tree and E is the set of the edges between the nodes, G is *rooted* if there is exactly one node (root) in N which has no incoming edges. G is *acyclic* if for each node a in N there are no paths in the form $(a = a_1, a_2)(a_2, a_3)...(a_{k-1}, a_k = a)$, for all $k \leq |N|$ and $(a_i, a_{i+1}) \in E$. A tree $T = \langle N, E \rangle$ is an acyclic, rooted, and directed graph. Without loss of generality we restrict our attention to the case of *binary trees*. In fact it is possible to map a generic tree with bounded degree into a binary one, preserving the information we are interested in. For example, each set (with at least three nodes) of children of a given node, can be rearranged as complete binary tree rooted at that node. The structure of the trees is manipulated using three instructions:

1. create_tree() which creates a tree containing only the root;
2. expand(n, b_1, b_2) which, given a node n and labels b_1, b_2, creates two new nodes and adds them as children of n (b_1, b_2 can be thought of as the "names" of the new nodes);
3. remove(n) which, given a leaf n of the tree, removes it from the tree. This instruction cannot be applied directly to an internal node. To remove a subtree R of T, one should repeatedly remove all nodes in R starting from leaves.

These three operations are assumed to be the only ones available for modifying the "physical structure" of the tree.

The tree implements a partial ordering between the nodes in the tree. Given two nodes n and n', we write $n \preceq n'$ if n is an ancestor of n'; $n \prec n'$ additionally says that $n \neq n'$. The root is the bottom of the ordering.

We also consider a set of *attributes* Γ, with $|\Gamma| = O(|N|)$. These attributes are used in association with the nodes of the tree—we assume that attributes from the set Γ are attached to given nodes of the tree. At each node u of the tree, two operations are possible:

- assign$(u) = X$: $N \rightarrow \Gamma \cup \{?\}$, this labelling function attaches the attribute X to the node u. If assign(u) has not been executed, then assign$(u) =?$.
- dereference(X, u): this operation identifies the closest ancestor v of node u such that assign(X, v) has been executed. More precisely, the operation is used to retrieve the node $v(=$ dereference$(X, u))$ such that

$$\text{assign}(v) = X \ \wedge \ v \preceq u \ \wedge \ \forall z.(v \prec z \Rightarrow \text{assign}(z) =?)$$

The OP-Problem is the problem of handling any arbitrary, correct, on-line sequence of operations create_tree, expand, remove, assign, and dereference.

We impose the additional requirement that each attribute can be used at most once along each branch of the tree; formally, for each pair of distinct nodes u_1, u_2 in the tree belonging

to the same branch (i.e., $u_1 \preceq u_2$ or $u_2 \preceq u_1$) and for each attribute X, we have that $\mathsf{assign}(u_1) = X$ implies $\mathsf{assign}(u_2) \neq X$. This restriction is motivated by the dynamic binding environments that arise during Prolog and constraint programming computations: here attributes are used to represent logical variables, and each logical variables can be assigned at most once along each non-deterministic branch of computation. Later, in Section 3.9, we will extend this problem to the case handling multiple occurrences of an attribute along a branch.

In the formalization above we have assumed the presence of only one attribute per node. This restriction can be made without loss of generality (we can always assume that the number of bindings in the node is bound by a program dependent constant).

Figure 3.1 illustrates an example of a tree with various operations performed on it.

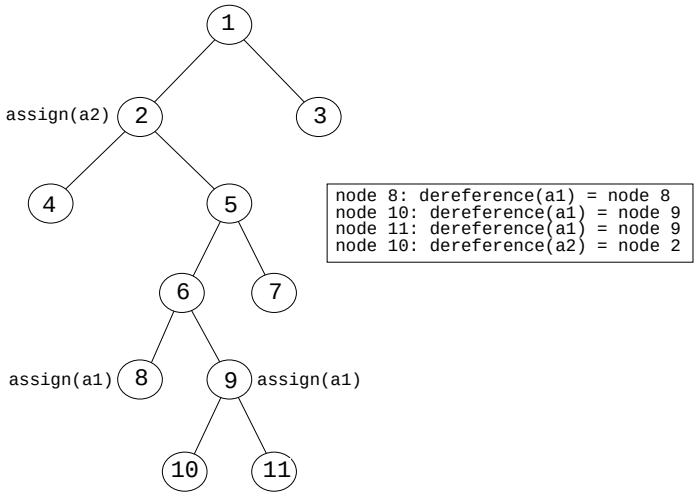

Figure 3.1: OP Operations on a Tree

3.4 Lower Bound Time complexity

The complexity of the OP-Problem on Pointer Machines has been studied previously [42].

In [41, 42] the following result regarding the worst-case time complexity of the OP-Problem was established.

Theorem 3.4.1 *On both Pointer Machines and Pure Pointer Machines, the worst case time complexity of* OP-*Problem is* $\Omega(\lg n)$ *per operation, where* n *is the number of instructions executed.*

The basic idea of the proof is that since there is no direct addressing and no pointer arithmetic in the Pointer Machines, starting from a particular node only a "small" number of nodes can be accessed in a small number of steps.

3.5 Existing Solutions to OP-Problem

As mentioned earlier, the OP-Problem originates from the abstraction of problems arising in the context of parallel execution of Prolog programs and from the management of late-binding in object-oriented languages. The existing methodologies used to solve such problems in the respective domain, provide sub-optimal solutions in our abstractions. For example, most solutions proposed in the context of parallel Prolog boil down to a worst-case time complexity $O(n)$ per operation, where n is the size of the tree.

In [40, 42] a solution is presented for the OP-Problem on (Pure) Pointer Machines that makes use of tables to summarize the assignment of attributes along each branch. It is proved that this solution has a worst-case time complexity of $O(\sqrt[3]{n})$ per operation. To our knowledge, this is the best solution ever proposed for this problem.

3.6 Our Solution to OP-Problem

We present a solution to the OP-Problem that has a time complexity of $O(\lg n)$ per operation. For now, we assume the RAM model for complexity analysis. The next two sections discuss the complexity of the solution on Pointer Machine models.

We start by looking at a simplified version of the problem. We focus on a growing only tree. Thus, the only operation allowed on the tree is $\mathsf{expand}(u, X_1, X_2)$ which expands the leaf u by appending two successors n_1 and n_2, respectively labelled with attributes X_1 and X_2. Here we are assuming, as mentioned above, that each node in the tree has bounded degree; thus, without loss of generality, we can as well restrict our attention to binary trees. Also, for simplicity we assume that X_1 and X_2 are the actual attributes associated to such nodes—i.e., we avoid explicit use of assign. Only leaves can be expanded. The tree is coupled with the labelling function:

$$\mathsf{assign}\colon N \to \Gamma \cup \{?\}$$

where N is the set of all the nodes in the tree and Γ is the finite set of attributes. Given this, we can define the function

$$\mathsf{dereference}\colon \Gamma \times N \to N \cup \{\bot\}$$

which given a label X and a node u returns the node v (if any) on the path (from the root) ending in u which is labelled X. Our problem now reduces to the efficient implementation of the two operations expand and dereference.

Recall that for each pair of distinct nodes u_1, u_2 belonging to the same branch (i.e., $u_1 \prec u_2$ or $u_2 \prec u_1$) and for each attribute X, we have that $\mathsf{assign}(u_1) = X$ implies $\mathsf{assign}(u_2) \neq X$. This means that each attribute X can be assigned at most once in each branch of the tree. Note that this condition is always satisfied when modelling computations that originate from execution of declarative languages (e.g., logic and constraint programming), where variables are mathematical entities that can be instantiated at most once along each (non-deterministic) computation.

3.6.1 A Gist of the Solution

Our solution to the problem can be described briefly as follows: For each attribute X, order the nodes according to pre-order traversal of the dynamic tree. While doing expansion, maintain these nodes in a structure where binary search can be performed efficiently. The operation dereference(X, u) can then be performed by doing a binary search on the structure for X and making repeated use of the ancestor test. The details of the solution are provided in the following sections.

3.6.2 Auxiliary Data Structures

The solution makes use of an efficient solution to the ancestor problem. The ancestor problem for dynamic trees has been studied extensively [4, 10, 29, 44, 58]. The ancestor problem can be solved in worst-case time $O(1)$ per operation on Random Access Machines [58].

We also use AVL trees in our solution [12]. Nevertheless, any equivalent method that allows to maintain insertion and deletion of nodes in a binary search tree maintaining a balanced structure with $O(\lg n)$ length branches can be substituted for AVL trees in our solution.

Another data structure that we use is a data structure for the management of *generalized linked lists*. A generalized linked list allows one to perform three operations:

- insert(x, y): inserts x immediately after y in the list;
- delete(x): removes x from the list;
- compare(x, y): returns true if x occurs before y in the list.

An efficient solution for managing generalized linked lists has been proposed, for example, in [57]. The solution is based on the use *indexed* $BB[\alpha]$ trees and allows to perform n insertion/deletions in time $O(n)$ and each comparison in time $O(1)$ on RAMs and (non-pure) Pointer Machines. For the sake of simplicity we will also consider the following two additional operations in the generalized linked list, previous(x) and next(x) that respectively return the predecessor and successor of element x in the linked list. Both operations can be easily supported in time $O(1)$.

3.6.3 The expand Operation

We explain how the operation expand(u, X_1, X_2) is accomplished. Each node created is inserted in a regular tree structure implemented using a record per node; we assume that each node u in the tree contains pointers to its parent (*parent*(u)) and to its left and right child (*left*(u), *right*(u)). In addition, following the idea used in [57, 58], we maintain two generalized linked lists along with the main tree. The first linked list (*pre-list*) maintains the nodes of the trees according to a *pre-order* traversal of the main tree, while the second

linked list (*post-list*) maintains the nodes of the tree according to a *post-order* traversal.[1] This situation is illustrated in a simple example in Figure 3.2.

Each node u in the main tree contains two pointers, one indicating its representative in the pre-list ($pre(u)$) and one pointing to the node's representative in the post-list ($post(u)$). Since our only operation to introduce nodes in the tree is expand, which adds new nodes as leaves of the tree, then managing the pre-list and post-list is fairly simple. For expand(v, X_1, X_2), if u_1 and u_2 are the new nodes created (respectively left and right child of v), then the following steps are required: insert$(pre(u), pre(u_1))$ and insert$(pre(u_1), pre(u_2))$ in the pre-list, and insert(previous$(post(u)), post(u_1))$ and insert$(post(u_1), post(u_2))$ in the post-list. Thus, considering that inserting the node in the main tree can be done in $O(1)$ time, the management of the pre-list and post-list requires $O(\gamma)$ per expand, where γ is the complexity of insert in the generalized linked list. Using the results from [57], we can ensure that an arbitrary collection of n expand can be accomplished in time $O(n)$ (w.r.t. the management of pre-list and post-list).

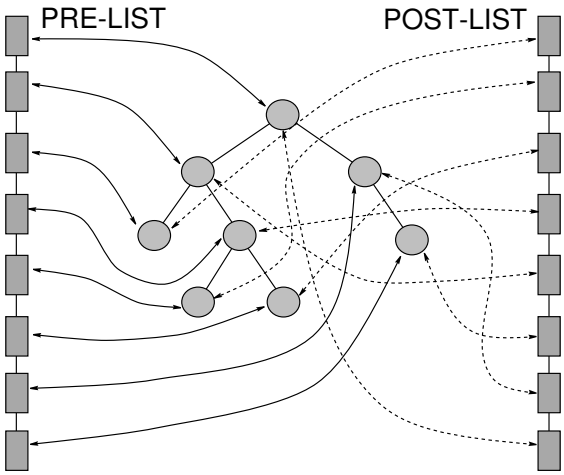

Figure 3.2: OP: Main Tree, pre-list, and post-list

Handling Attributes

The steps described above allows to handle only one part of the expand operation, i.e., the addition of new nodes in the trees. To complete the execution of this operation we need to also handle the attributes associated to the new nodes. We extend our picture by adding a collection of AVL binary trees used to keep track of occurrences of attributes in the main tree. We make use of a separate AVL tree for each attribute used in the tree. Whenever a new occurrence of a certain attribute is introduced in the tree (in node u), then a new node ($alink(u)$) is added to the AVL tree for that attribute. In this context we assume, without loss of generality, that attributes are represented simply as pointers to the root of the AVL

[1] In [57] it is also shown how to combine both information in a single linked list.

tree for that particular attribute.

Let u be a node in the main tree, and let X be the attribute stored in u ($attr(u) = X$). We will assume that u stores a pointer ($alink(u)$) to the node in the AVL tree of X representing that particular occurrence of the attribute. We also assume that each node in each AVL tree has a pointer back to the corresponding node in the main tree.

Insertion of new nodes in the AVL tree has to be made in a position that reflects the pre-order of nodes in the main tree. This can be accomplished using a simple binary search in the AVL tree, using the pre-list to determine at each step if we need to move left or right in the AVL tree. This is sketched in the following code fragment where u is the new node in the main tree and X is its attribute:

```
1: r := root of AVL tree of X;
2: last := nil;
3: while (r ≠ nil) do
4:     last := r;
5:     r_main := node in the main tree corresponding to r;
6:     if compare(pre(r_main), pre(u)) then
7:         r := right(r);
8:     else
9:         r := left(r);
10: endwhile
11: r_last := node in the main tree corresponding to last;
12: if compare(pre(r_last), pre(u)) then
13:   add alink(u) as right child of last in AVL tree of X
14: else
15:   add alink(u) as left child of last in AVL tree of X
```

Note that the tests in lines 6 and 12 can be performed in $O(1)$ using the generalized linked list used for the pre-list.

This additional component of the expand operation requires $O(\lg k_X)$, where k_X is the number of occurrences of X in the main tree.

Combining these components we can conclude that the expand operation can be performed in $O(\lg n)$.

3.6.4 The dereference Operation

The operation dereference(X, u) can be accomplished by performing a binary search using the AVL tree for attribute X. The idea is use the pre-list to find a node in the AVL tree for X that is an ancestor of u. This process is illustrated in the following code fragment:

```
1: r := root of AVL tree of X;
2: while (r ≠ nil) do
3:     r_main := node in the main tree corresponding to r;
4:     if compare(pre(r_main), pre(u)) then
5:         if (r_main is an ancestor of u) then
6:             return r_main;
7:         else
8:             r := right(r);
9:     else
10:        r := left(r);
11: endwhile
12: return nil;
```

Observe that:

- the test in line `4` can be done in $O(1)$ using the pre-list;
- the test in line `5` can be also performed in $O(1)$: r_{main} is an ancestor of u iff `r` appears before u in the pre-list and after u in the post-list. Thus, two compare tests (in two generalized linked lists) are sufficient to answer the question.

Thus, the overall cost of this operation is simply $O(\lg k_X)$ where k_X is the number of occurrences of the attribute X in the main tree. Overall, this operation is $O(\lg n)$.

To summarize, we can conclude:

Proposition 3.6.1 *The* OP*-Problem can be solved on RAM with worst-case time complexity* $O(\lg n)$ *per operation, where* n *is the number of nodes in the tree.*

3.6.5 Relaxing the Restrictions

Most of the restrictions imposed on the problem description studied above, such as using binary trees and combining expand and assign, can be made without any loss of generality.

The problem we tackled earlier deals with growing tree only. Nevertheless, the management of the remove operation does not introduce any extra complication. Removing a node for the main tree requires adjusting the auxiliary data structures—i.e., removing the node from the AVL tree and from the generalized linked lists. All these steps can be accomplished in $O(1)$ time.

3.7 OP-Problem on Pointer Machines

The construction described above does not make use of any of the peculiar features of the RAM model. This is because in no place we have used the ability to perform random

access—that represents the key difference between Pointer Machines and RAM. Additionally, the management of the generalized linked lists provided in [57] works also on Pointer Machines (with arithmetic capabilities). This allows us to conclude:

Proposition 3.7.1 *The* OP-*Problem can be solved on Pointer Machines with worst-case time complexity* $O(\lg n)$ *per operation, where* n *is the number of nodes in the tree.*

This result is particularly important since this shows that our solution is *optimal*, as the complexity matches the lower bound on Pointer Machines proved in [42].

3.8 OP-Problem on Pure Pointer Machines

The $O(\lg n)$ solution to the problem on RAM and Pointer Machines does not immediately provide an $O(\lg n)$ solution on Pure Pointer Machines, i.e., Pointer Machines without arithmetic capabilities. While the construction of the main tree and the AVL trees can be realized also on Pure Pointer Machines without added complexity, the problem arises from the management of generalized linked lists. The solution proposed in [57], as well as the other solutions proposed in the literature, make use of arithmetic, which cannot be reproduced on Pure Pointer Machines (without adding a significant cost penalty). For example, a simplified version of the generalized linked list, where insertion can be performed only at the end of the list and no removals are allowed, has been shown to have a worst case time complexity $\Omega(\lg \lg n)$ per operation [47].

A simple solution for the OP-Problem on Pure Pointer Machines can be derived by replacing the generalized linked lists used for the pre-list and post-list with AVL trees. In this case, the cost of performing insert and compare becomes $O(\lg n)$. This leads to the simple proposition:

Proposition 3.8.1 *The* OP-*Problem can be solved on Pure Pointer Machines with worst-case time complexity* $O(\lg^2 n)$ *per operation, where* n *is the number of nodes in the tree.*

Although this solution is not optimal (the best known lower bound is still $\Omega(\lg n)$), it is still exponentially better than the previously best known solution, which is $\tilde{O}(\sqrt[3]{n})$ [42].

3.9 Handling Multiple Occurrences

The original definition of the problem relies on the essential assumption that each attribute can be used at most once along each branch of the tree—i.e., for each pair of distinct tree nodes u_1, u_2 such that $u_1 \preceq u_2$ or $u_2 \preceq u_1$, and for each attribute X we have that

$$\mathsf{assign}(u_1) = X \Rightarrow \mathsf{assign}(u_2) \neq X$$

In this section we provide two solution to the OP-Problem supporting also multiple occurrences of the same attribute along the same branch.

3.9.1 Motivations

Allowing multiple occurrences of the same attribute along each branch of the tree is motivated by the needs of modelling alternative search problems. In [41] we argued that the same construction provided by the OP-Problem represents a correct abstraction for the problem of managing late-binding in the context of object-oriented languages.

The abstraction of an object-oriented execution should account for the various issues present in object-oriented programming, (e.g., definition of new classes/objects, inheritance of attributes etc.). Previous research, as well as experience gained from the development of practical implementations [21, 49, 55, 62] shows that management of objects' attributes is indeed one of the key issue in the implementation of an object-oriented system. The fact that the efficiency of attribute management has a significant impact on the performance of an object-oriented system has been experimentally demonstrated [15, 16, 30, 64]; furthermore, exploitation of parallelism and adoption of more complex approaches to object orientation make the problem of attribute management even more complex and difficult to implement.

As described earlier, one of the main issues in dealing with inheritance in object-oriented programming is the search of a given attribute. This search may be needed when a class/object is created (in order to detect which variables should be included in each object) or during execution (in order to detect what is the correct method to execute in response to a given message). In the rest of the discussion we will adopt the following assumptions: the hierarchy is represented by a single-rooted tree which can dynamically grow and shrink; each node either introduces one attribute or redefines one. In [41] we have shown that it is not restrictive to focus on binary trees. A hierarchy—here assumed to be represented by a dynamic tree of either classes (class-based system), or objects (prototype-based system)—is dynamically created. Each new leaf created in the tree refines the object(s) defined at its parent node, by adding attributes and/or shadowing existing ones. The structure (set of attributes) of an object at node n (denoted by $S(n)$) is thus determined by a combination $d(n, d(n_1, d(n_2, \ldots)))$ of the definitions present in the nodes n_i, where n_i is the ancestor of n at distance i from node n. d is a function with signature

$$d : \mathit{Nodes} \times 2^{\mathit{Nodes} \times \mathit{Attr}} \rightarrow 2^{\mathit{Nodes} \times \mathit{Attr}}$$

where *Attr* is the set of possible attributes and *Nodes* is the set of nodes in the hierarchy. Intuitively, each object is described by a set of pairs $\langle n, \psi \rangle$, where each pair identifies one attribute composing the object; ψ determines the attribute and n determines the node (class/prototype) from where we are going to inherit the definition of ψ. If $\gamma(n)$ is the attribute introduced in node n, then $d(n, \mathcal{S})$ is defined as

$$d(n, \mathcal{S}) = \{\langle n, \gamma(n) \rangle\} \cup \{\langle m, \psi \rangle \mid \langle m, \psi \rangle \in \mathcal{S} \ \wedge \ \psi \neq \gamma(n)\}$$

We can assume that a total of M attributes are available—i.e., there are M variables and methods. If the computation tree has size N, then we can assume that $M = O(N)$ (see also Section 3.6.5).

If $\langle m, \psi \rangle \in S(m)$ and there are no further definitions of ψ in the sequence of the path from m to a descendent node n, then the path from m to n is said to be ψ-*free*.

At each node n two operations on attributes are possible:

1. assign an attribute ψ to a node n; as a consequence, every reference to ψ in a node $m \succeq n$ (such that the path from n to m is ψ-free) will produce n as result—i.e., identifies that $\langle n, \psi \rangle$ is present in $S(m)$.

2. dereference an attribute ψ at a node n—that is identify the node

$$max_{\preceq}\{M \mid m \text{ defines } \psi \ \wedge \ m \preceq n\}$$

   equivalently, we want to determine the node m such that $\langle m, \psi \rangle \in S(n)$.

Also in this case, the formalization of the problem can be completed by adding an alias operation—used to subsume constructions like the ability to access redefined properties by using superclass names as qualifiers (as in C++), and the *super* (Smalltalk) and *resend* (Self), as well as the "inverted" behavior of languages like Beta (with the *inner* construct). The fact that the OP-Problem correctly abstracts the problem of handing attribute binding in object-oriented systems of the described type has been shown to be correct in [41].

3.9.2 A Simple Solution

We start by providing a simple solution to this generalized version of the problem. The solution will provide the ability to detect the attribute with a time complexity $O(\lg^2 n)$ where n is the number of nodes in the tree. The idea is to reduce the situation to the simpler problem described in the previous sections—i.e., to the problem of detecting occurrences of attributes along a branch under the assumption that each attribute is used at most once in each branch. To accomplish this, we organize the various occurrences of attribute X in the tree in separate lists, that we call $\ell_1^X, \ell_2^X, \ldots$ The list ℓ_i^X contains the nodes of the tree that are labelled with attribute X and that represent the ith occurrence of X on the path to the root of the tree (see also Figure 3.3). Intuitively, we consider the elements of ℓ_i^X as if they were labelled with a different attribute (X_i)—that clearly will occur at most once in each branch of the tree. This allows us to reuse the algorithms from the previous section to perform dereference and verify whether a given leaf v has an ancestor labelled X_i. If such ancestor is not found, then this means that on the path from the root to v there are less than i occurrences of X; on the other hand, if an ancestor of v is found in ℓ_i^X, then we know that there are at least i occurrences of X on the path from the root to v. This property can be used to impose a binary search on the lists ℓ_i^X, using the result of the local searches on each list to decide how to proceed. I.e., if we have k lists ℓ_i^X (e.g., $\ell_1^X, \ldots, \ell_k^X$) then we can start doing the first search on the list $\ell_{k/2}^X$; if this tells us that there are fewer than $k/2$ occurrences of X on the path, then we can repeat the process on $\ell_{k/4}^X$, otherwise we should continue the search by processing $\ell_{3k/4}^X$, and so on.

For each attribute X we need to use a set of AVL trees, let us call them X_1, X_2 , ... , X_n. Each tree X_i contains nodes v_j assigned to X such that v_j is the i^{th} node assigned to X in the branch starting from the root of the tree and ending in v_j. Nodes in X_i are ordered following the pre-list order. For each attribute X we define an AVL tree called X-tree. Each X-tree contains ordered nodes numbered from 1 to n, such that a node j points to the root of the corresponding X_j AVL tree. We also use the previously described pre and post lists. Figure 3.3 shows an example of these data structures.

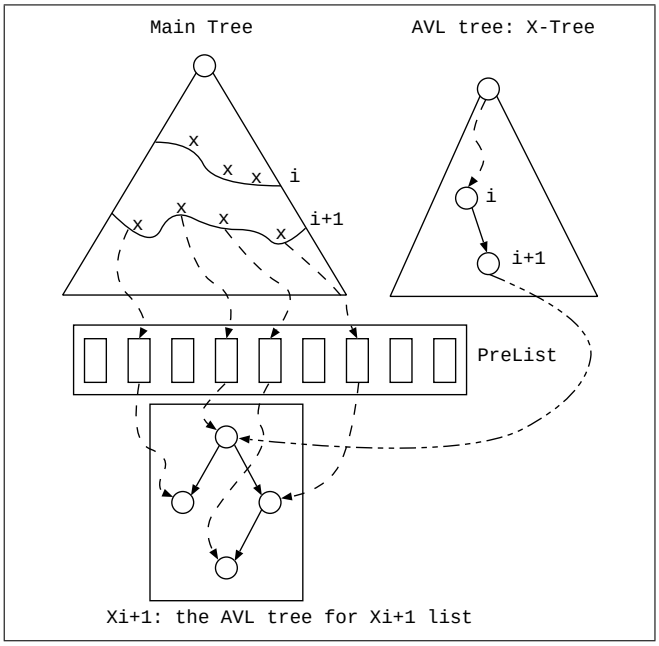

Figure 3.3: Data structures for a simple solution for the OP-Problem

The expand operation

This operation updates the pre-list and post-list, as already described in the previous sections. Let us assume that a new node u is added and assigned to the X attribute. Let v=dereference(X, u)—i.e., v represents the closest ancestor of u that has also been assigned to the same attribute X. Assuming that v is in the X_i tree, then we need to add u to the X_{i+1} tree, according to pre-list order. This last operation can be executed in $O(\lg n)$. Thus the total cost is $O(\lg n$ + cost of dereference).

The dereference operation

Let assume dereference(X, u) is requested and the main tree contains n nodes. The search starts from the root r of the X-tree. Let assume that r points to the X_r tree root. As described before we perform a binary search on the X_r tree, searching for an ancestor of u only looking at X_r nodes (in $O(\lg n)$ time). If this search finds an ancestor of u, we iterate

this process on the right child of r, else the ancestor can only be found in a previous level and we proceed with a left child of r. This double bisection takes $(O(\lg^2 n))$ time as the expand operation.

3.10 Our $O(\lg n)$ Solution

Let us now see how we can *efficiently* solve this more general version of the problem. To handle this problem we make use of an extension of the data structure presented in [44] called *ancestor-Tree*. This structure supports the following operations:

- add_node(x, p): adds the node x as child of node p
- nca (u, v): returns the nearest common ancestor of u and v

The data structure presented in [44] is based on the use of the so-called *p-lists* associated to each node of the tree; the p-lists maintain a collection of pointers to selected ancestors of the given node, providing the ability to perform a binary search on each branch of the tree when looking for nearest common ancestors (see Section 5.1 for further details). Thanks to such data structure, it is possible to perform both the above mentioned operations (add_node and (nca) in worst-case time $O(\lg n)$ on a Pure Pointer Machine (in the complete dynamic case—i.e., without the need of any pre-processing of the tree).

The intuition behind our solution is to maintain with the main tree two different traversals: a pre-order traversal (as done in the previous solution) as well as *reversed pre-order*, where the right subtree is traversed before the left subtree. The use of these pre-orders allows us to facilitate detecting the ancestor of interest, by detecting two nodes (one from each traversal), lying respectively to the left and to the right of the ancestor of interest.

3.10.1 The expand operation

Whenever an expand operation is performed, the new node is inserted in the regular tree. The two linked lists, PreLR-list and PreRL-list, maintain the nodes of the regular tree according to a *pre-order* visit. The PreLR-list (PreRL-list) lists the *pre-order* visit executed choosing *first* the left (right) child of a node and *after* the right (left) one. For each node u in the main tree, we will call $preLR(u)$ ($preRL(u)$) the node in the PreLR-list (PreRL-list) associated to u. For expand(u, X_1, X_2) it is possible to maintain the two orders in time $O(1)$, as already discussed earlier.

3.10.2 Attributes

We use two distinct AVL trees (LR AVL tree and RL AVL tree) for each attribute. One refers to the PreLR-list and the other to the PreRL-list, and they are managed as described earlier.

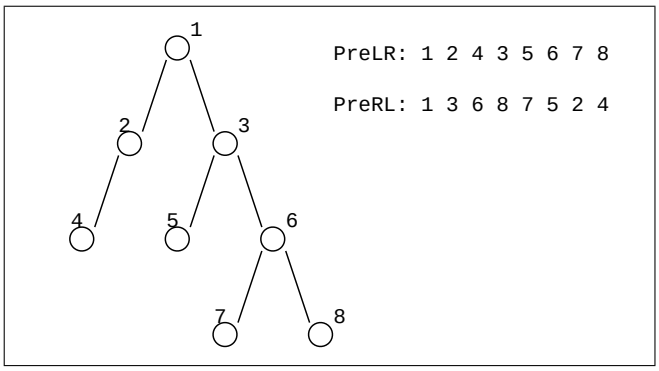

Figure 3.4: Example of preLR and preRL visits

When an attribute X is assigned to a node u, we assume that X stores two pointers ($LRlink(X)$ and $RLlink(X)$) to the roots of the two AVL trees (LR AVL tree and RL AVL tree respectively) associated to the attribute X. We assume that each node in each AVL tree has a pointer back to the corresponding node in the main tree and has a pointer to the corresponding node in the ancestor-Tree structure. The intuition behind the ancestor-Tree for an attribute X is the following: if u and v are two nodes in the main tree that have been assigned X, and u is the nearest ancestor of v having attribute X, then u will be the immediate parent of v in the ancestor-Tree of X.

When an attribute X is assigned for the first time, we create the root node in the ancestor-Tree for the X attribute. We assume that each node n in each ancestor-Tree has a pointer ($back(n)$) back to the corresponding node in the main tree. We also assume that for all root node r_X, $back(r_X) = nil$. Let X be assigned to the node n in the main tree and $\mathsf{dereference}(X, u) = v$. Thus, the node v is the nearest ancestor of u that is also assigned the attribute X. This operation can be performed in $O(\lg n)$ time, as we will show later. Thus, we insert a new node in the ancestor-Tree structure using the operation $\mathsf{add\_node}(u, v)$, with a constant time cost. We assume that the new node u has a pointer back to the corresponding node in the main tree.

According to this, the expand operation incurs a cost of $O(\lg n)$.

3.10.3 The dereference operation

The operation $\mathsf{dereference}(X, u)$ can be realized with the following steps:

1. first of all, we can execute the following code fragment on PreLR-list (PreRL-list) using the corresponding AVL tree to find the closest node X_{LR} (X_{RL}) to node u

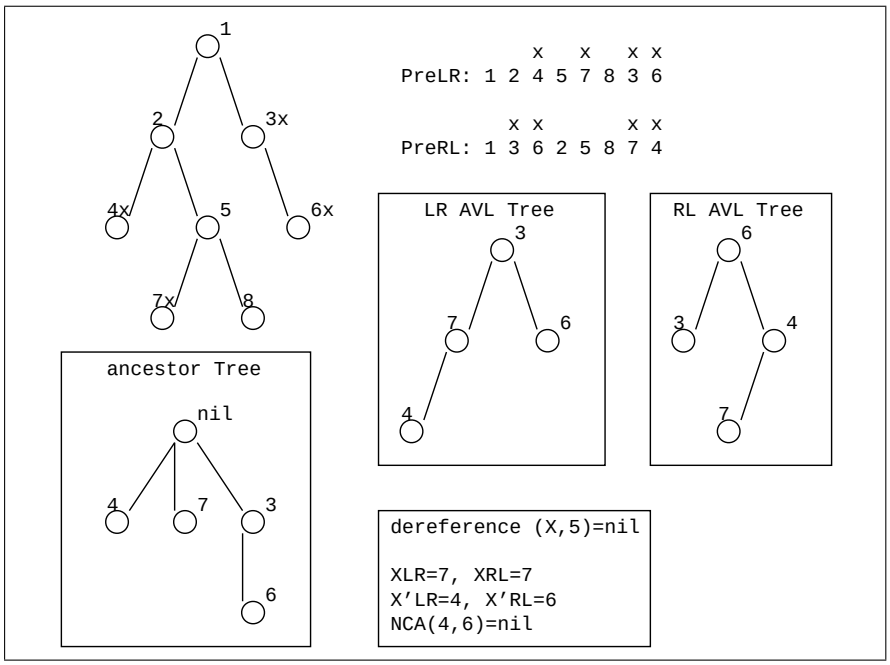

Figure 3.5: OP-Problem: Example 1 - A nil dereference

```
1: r := root of AVL tree of X;
2: last := nil;
3: while (r ≠ nil) do
4:     last := r;
5:     r_main := node in the main tree corresponding to r;
6:     if compare(preLR(r_main), preLR(u)) then
7:         r := right(r);
8:     else
9:         r := left(r);
10: endwhile
11: return last;
```

This algorithm will spend $O(\lg n)$ time to find the closest node X_{LR} (X_{RL}). For example, referring to Figure 3.5, we want to calculate dereference$(X, 5)$. Running the fragment code above on PreLR-list X_{LR} is 7, and on PreRL-list X_{RL} is 7. Another example of a dereference operation is presented in Figure 3.6.

2. if compare$(preLR(X_{LR}), preLR(u))$ then X'_{LR}=X_{LR} else X'_{LR}=the previous node starting from X_{LR} using the PreLR-list order. (This can be done in $O(\lg n)$ time). In the example, the node 7 is after the node 5, and X'_{LR} is 4.

3. if compare$(preRL(X_{RL}), preRL(u))$ then X'_{RL}=X_{RL} else X'_{RL}=the previous node starting from X_{RL} using the PreRL-list order.

4. return nca (X'_{LR}, X'_{RL}), i.e., compute the nearest common ancestor of the two nodes X'_{LR} and X'_{RL} in the ancestor-Tree. This operation can be performed in worst-case time $O(\lg n)$, using, for example, the techniques studied in [44].

Combining these steps together, one can observe that the operation of dereference has a worst-case time $O(\lg n)$.

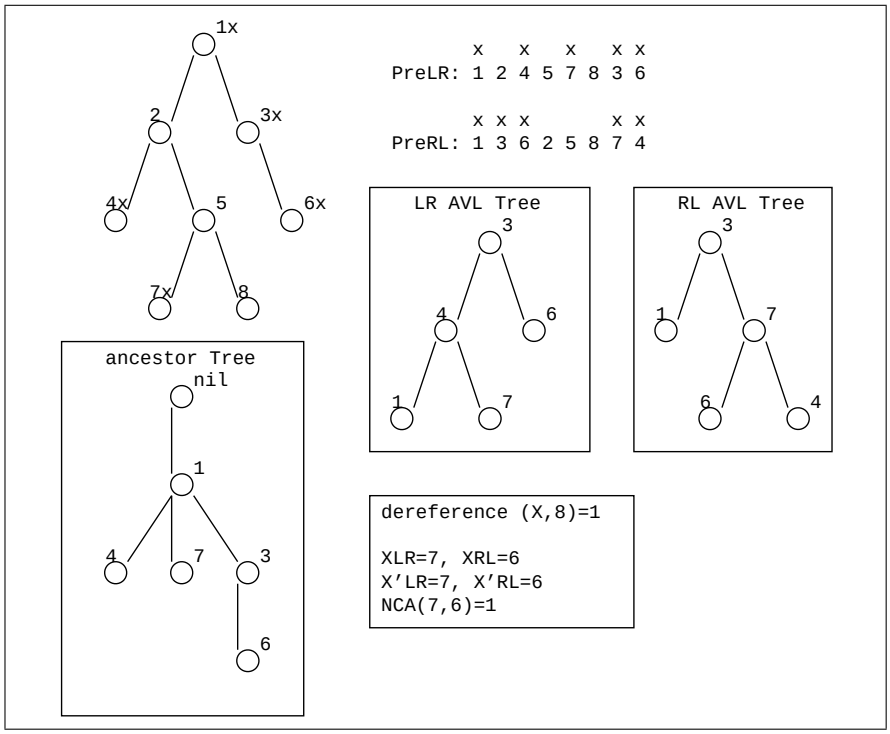

Figure 3.6: OP-Problem: Example 2

3.10.4 Correctness

In this section we want to prove that nca (X'_{LR}, X'_{RL}) = dereference(X, u).

Let X-node be a node n such that $assign(n) = X$. The basic idea is that, given the node v = dereference(X, u), then the path between u and v does not contain any other X-node. X-nodes can be found in the partitioning subtrees of v called A, B, C in Figure 3.7. An X-node in the A (B) subtree will appear *between* v and u in the PreLR-list (PreRL-list), but *after* u in the PreRL-list (PreLR-list). X-nodes in the C subtree will always be after the node u in both pre-lists. X-nodes X'_{LR} and X'_{RL} are *before* u in both pre-lists. Hence X'_{LR} is in A (or B) and X'_{RL} in B (or A) and nca (X'_{LR}, X'_{RL}) will be the root v of the two subtrees A and B.

We now provide a formal proof of correctness. Let v=dereference(X, u). Let us assume that node u is in the subtree rooted in the right son of v. Let L_1 be the sublist of PreLR list containing all left children of v, R_1 the sublist of PreLR list starting from the right son

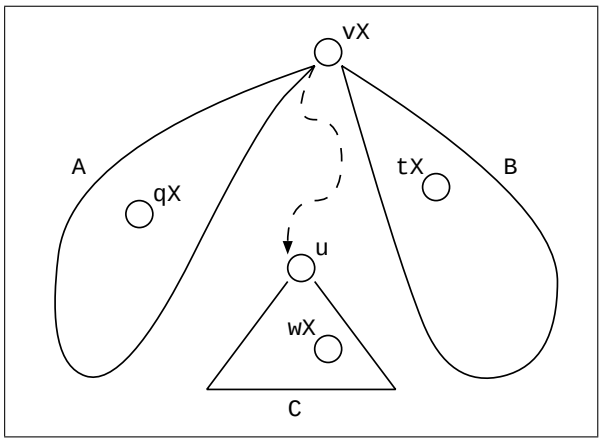

Figure 3.7: OP-Problem: A, B, C partitioning of v subtree

of v until u and R_2 the sublist of PreRL list starting from the right son of v until u. See figure 3.8.

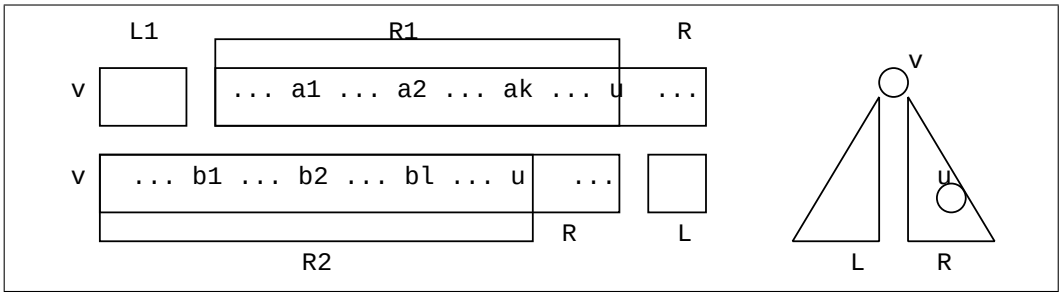

Figure 3.8: OP-Problem: Correctness

Thus, u is either in R_1 or R_2. We can distinguish two cases:

- R_2 does not contain any X-node. This implies that $X'_{RL} = v$ and nca $(X'_{LR}, v) = v$, since X'_{LR} is a child of v.
- R_2 contains l X-nodes. Let $a_1 ... a_k$ be the X-nodes in R_1 and $b_1 ... b_l$ the X-nodes in R_2. By construction we have that $a_k = X'_{LR}$ and $b_l = X'_{RL}$. For all i in $\{1 \dots k\}$ and for all j in $\{1 \dots k\}$, we have that $a_i \neq b_j$. For the sake of contradiction, let us assume that there are i and j such that $a_i = b_j$; as a consequence, the node a_i has to be an X-node parent of u. This goes against the assumption that v is the result of a the dereference operation.

  Given this construction, it follows that nca (a_k, b_l) is in the subtree rooted in v. Let us assume that t=nca $(a_k, b_l) \neq v$, where t is the corresponding in the regular tree of an X-node stored in the ancestor-Tree structure. Let $<_{LR}$ and $<_{RL}$ be the two partial orders induced by the PreLR and PreRL traversals. Thus, $v <_{LR} t <_{LR} u$ and $v <_{RL} t <_{RL} u$. From the second disequality, it follows that t is in R_2. Let us assume

that m is such that $t = b_m$. Since all nodes in R_2 are not in R_1, then it follows that a can only be in L_1. This leads to a contradiction, because any node in L_1 cannot be an ancestor of a node in R_1.

These facts allow us to conclude that nca (X'_{LR}, X'_{RL})=dereference(X, u).

Chapter 4

The nca Problem

4.1 Introduction to the nca Problem

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. In the *offline* version, T as well all the nca queries are known in advance.

The nca problem has been studied extensively. For the static case, the original work by Harel and Tarjan [29] provides a constant-time algorithm for performing the $\mathsf{nca}(x, y)$ operation after a linear-time preprocessing of the input tree. This result was later simplified and parallelized by Schieber and Vishkin [51]. A nice exposition of this result can be found in [28]. Later, Bender and Farach-Colton [4] provided an easily understandable and effectively implementable algorithm which provides constant time execution of $\mathsf{nca}(x, y)$ operation with linear-time preprocessing of the tree. In all the three algorithms above, complexity analysis is done assuming the RAM model. For the dynamic case, a work by Tsakalidis [58] provides algorithms with $O(\lg h)$ worst-case time for the nca operation and almost amortized constant time for add-leaf and delete operations in a dynamic tree, where h is the height of the tree. The algorithm is developed for a Pointer Machine model under the uniform cost measure assumption (constant time arithmetic for $\Theta(\lg n)$-size integers). This result has been recently improved in [2], where it is shown that the nca problem can be solved in worst-case $O(\lg \lg n)$ time per operation, and that it can be solved in $O(n+q \lg \lg n)$ time, where n is the number of link operations and q is the number of nca queries. Once again, the Pointer Machine model with uniform cost measure for arithmetic is assumed. The algorithms of Cole and Hariharan [10] provide the ability to insert (leaves and internal nodes) and delete nodes (leaves and internal nodes with one child) in a tree and execute the 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 of the problem, a linear-time algorithm on Pointer Machines (with arithmetic) was given by Buchsbaum et

al. [8].

The best known parallel nca algorithms on PRAMs (e.g., the algorithm by Schieber and Vishkin [51]) require $O(\lg n)$ parallel time for preprocessing using $O(n/\lg n)$ 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 preprocessing using an optimal number of processors, and $O(m)$ query time, where $I_m(n) = \min\{i \mid A(m,i) \geq n\}$ and A is the Ackermann's function. If this extra information is not known in advance, then the parallel preprocessing time is increased to $O(\lg n)$ 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 of factor of $\lg n$ in this translation (recall that PRAMs allow constant time arithmetic on numbers of size up to $\lg n$ and indexing into arrays using integer addresses, unlike Pointer Machines).

4.2 Our Solution to the nca Problem

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 (based on heavy/light edges [29] or centroid paths [10]). It has several nice features. In particular, it *does not* make any use of arithmetic. We use this compression scheme to provide an $O(n + q \lg\lg n)$ solution for solving the nca problem on PPMs in both the static case and the dynamic case where n is the number of add-leaf/delete operations and q is the number of nca queries. This solution is optimal, because of a previously known matching lower bound [29]. Moreover, it has the same complexity as that of an optimal solution for the problem on Pointer Machines that do allow constant-time arithmetic operations. 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 structure 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 polylog penalty that one has to incur in a generic translation of an algorithm designed for pointer machines with arithmetic to PPMs. We use this optimal solution to the nca problem to improve the solutions for many previously defined problems. In particular, we obtain an optimal solution to the generalized linked list maintenance problem on PPMs and we improve the solution of the OP-Problem to worst case $O(\lg n \lg\lg n)$ from the previously described above $O(\lg^2 n)$ [45].

Moreover, the scheme proposed is very local in nature and hence seems eminently parallelizable. We present (see Section 4.9) 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. Our algorithm requires $O(\lg n)$ 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 $O(\lg \lg n)$ 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 [47]. We provide an efficient Parallel Pointer Machine algorithm to solve this problem as well.

4.3 A compression scheme for trees

The compression algorithm we propose starts from the initial tree $T = T_0$ and repeatedly performs two types of compressions, thus 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 (called H-Tree), that summarizes the nearest common ancestor information of T. The key property of the H-Tree is that its depth is at most logarithmic in the number of nodes T. This allows a fast nca calculation.

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

Figure 4.1 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 Figure 4.1 leaf-compression removes nodes 10−15 (Figure 4.1.1) by merging them with their parents (Figure 4.1.2). A path-compression merges all paths ending in a leaf into their heads. For example, in Figure 4.1.3 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 always compressed starting from the leaves and moving towards the root, when possible. In Figure 4.1 we have marked the representatives of each compression with darker nodes.

4.3.1 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-Tree)*. The H-Tree, H, can be constructed from the sequence of trees obtained during the compression process (e.g., the sequence of trees shown in Figure 4.2).

If a leaf-compression is applied to node v in tree T_i and ℓ is the direct representative of

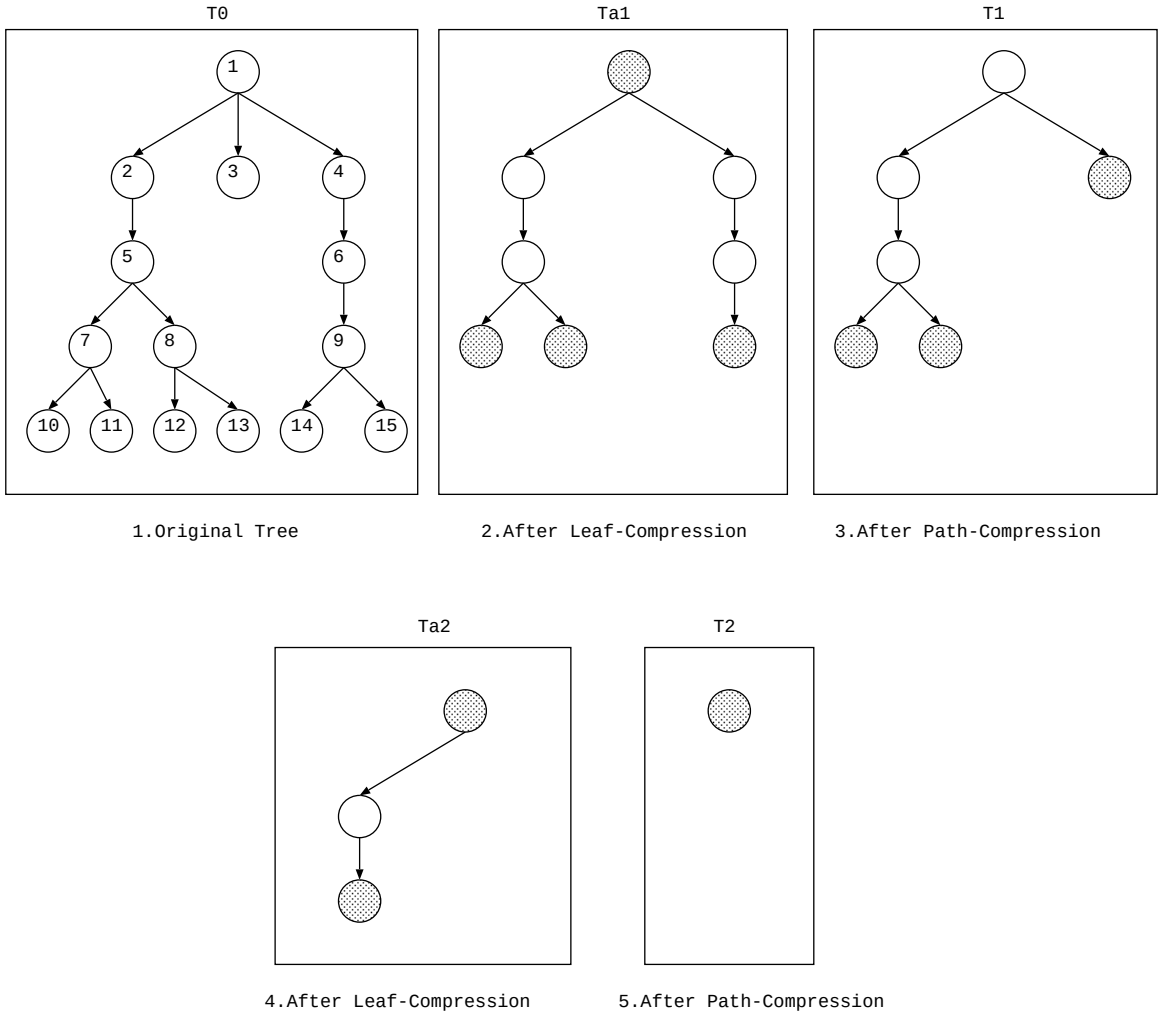

Figure 4.1: nca : An Example of Compression

v in such compression, then node v is connected to the last occurrence of ℓ in a tree T_j $(i < j)$, where ℓ 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^a_{i+1} (as for leaves $10, 11$ in Figure 4.2). If the children of a are leaf-compressed at different points in time (e.g., the children of 1 in Figure 4.2), 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, as shown in Figure 4.2. Such node is said to be 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 Figure 4.2) as the root and using the links between nodes and representatives as edges (e.g., the dark edges in Figure 4.2).

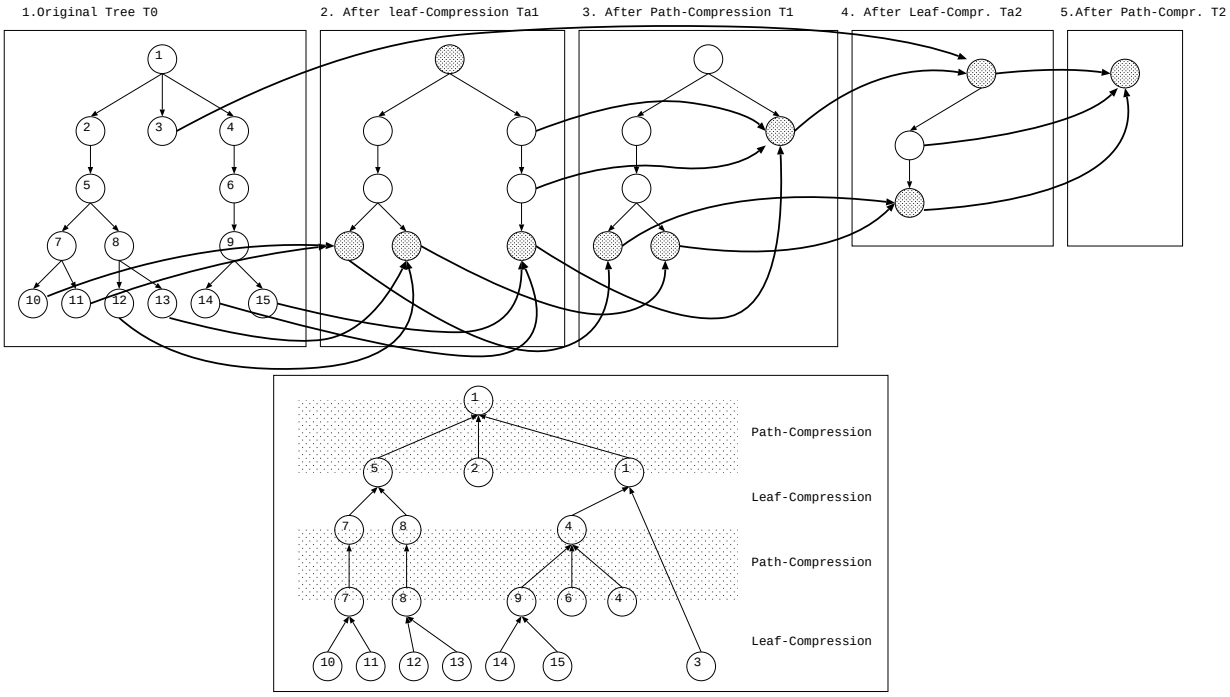

Figure 4.2: nca : Building the H-Tree

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. More importantly, 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. Moreover, note that one occurrence of v in H must be a child of the other occurrence of v in H.

Next lemma provides a result critical to the efficiency of the compression scheme. Let $subtree_T(v)$ be the subtree of T rooted at node v.

Lemma 1 *If a node v of T still exists in T_k then the $subtree_T(v)$ 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 $subtree_{T_1}(v)$. The node w_1 is the result of the previous path-compression on T_1^a. Let us call w_2 the leaf in T_1^a 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 $subtree_T(v)$ 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 $subtree_{T_i}(v)$ (see also Figure 4.3). The node w_1 is the result of the previous path-compression applied to T_i^a. Let us call w_2 the leaf in T_i^a 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_{i-1}^a. 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 $subtree_T(v)$ has at least $1 + 2^{i-1} + 2^{i-1}$ nodes.

□

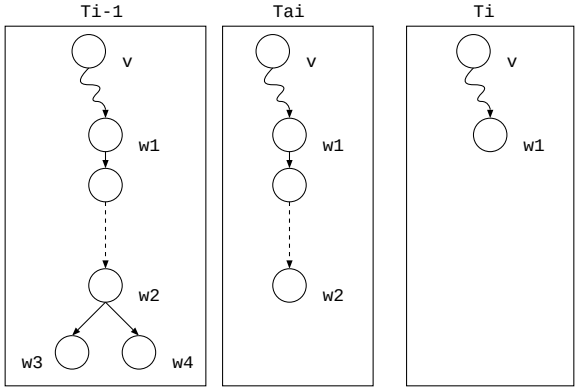

Figure 4.3: nca : Generic compression

Corollary 4.3.1 *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.*

4.3.2 Answering ncas Using H-Trees

Given the query $\mathsf{nca}(x, y)$, where $x, y \in T$, it is possible to answer the query using H. In particular, computation of the nca of two nodes in T can be computed by first considering 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 formal algorithm called nca_H is presented in Section 4.4.

We show now that the H-Tree preserves enough nca information from T. Let z be the $\mathsf{nca}_H(x, y)$. If z is a representative of a leaf-compression, then z is also the 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) In this case, the nca of x and y is the highest node between z_i and z_j. Thus, answering an nca query in T boils down to computing an nca query in H.

From Corollary 4.3.1, we can infer that the height of H is $O(\lg n)$. In [44] it was shown that there is a (dynamic) PPM algorithm given a tree with height h allows the computation of the nca of any two nodes in the tree in worst case time complexity $O(\lg h)$ per query. A summary of the method used in the algorithm presented in [44] is presented in Section 5.1. Using this result, we can compute the 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 $O(\lg \lg n)$.

4.4 An Algorithm for Static nca Queries

In the following sections, as in [2], we use a more general definition of nca: $\mathsf{nca}(x, y) = (z, z_x, z_y)$, 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 preprocessing creates $2k + 1$ trees, named $T_0, T_1^a, T_1, T_2^a, \ldots, T_k^a, T_k$, for some $k \leq \lg |n|$ where n is the number of nodes in T. 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_i^a 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.

Data structures (see Figures 4.4 and 4.5): For each w in T, let $entry(w)$ be the entry point of w in H. Note that each node in a tree T_i or T_i^a is a copy of a node existing in T; if the node $v \in H$ is a copy of node $u \in T$, then $node(v)$ is a pointer to u. Let $children(v)$ be a pointer to a list of nodes containing the children of node v in H, and let $parent(v)$ 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:

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

Construction of the H-Tree: To answer the nca queries, we require the ability to compare the depth of nodes that appear on the same branch. To efficiently accomplish this, we attach to each node information representing the depth of the node in T. This can be accomplished by making use of the data structures developed to solve the *Temporal Precedence ($\mathcal{TP}$)* problem [43, 47]. These data structures allow one to efficiently maintain a list, where new elements can be added at the end of the list (insert) and elements can be compared (precedes)

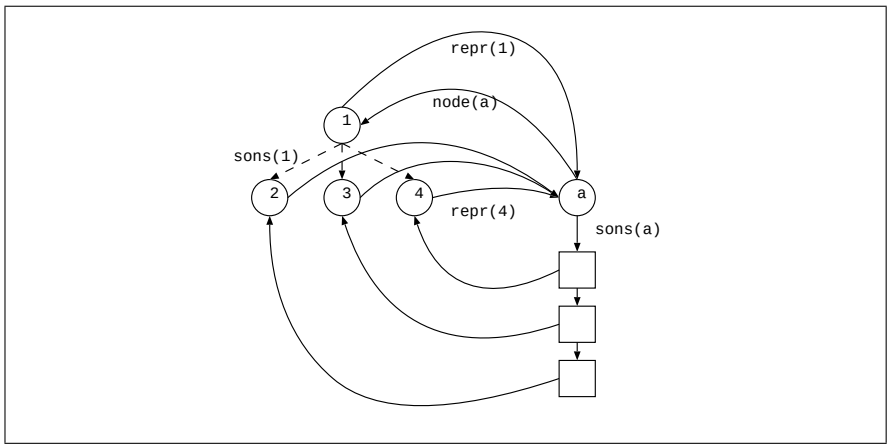

Figure 4.4: nca : Data Structure Involved During Leaf-compression

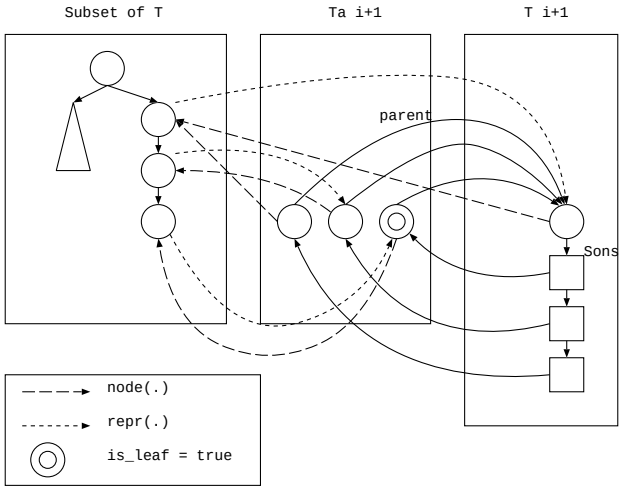

Figure 4.5: nca : Data Structure Involved During Path-compression

to determine which one was inserted first. From [43] we know that this two operations can be performed on-line in a PPM with worst-case time complexity $O(\lg \lg n)$ per precedes and amortized $O(1)$ per insertion. Thus, the labelling of nodes of T with their depth information can be performed in $O(n)$ time.

The preprocessing algorithm used to construct H is described in Figure 4.6. 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 $entry(v) = u$ and $node(u) = v$. (lines `1-4` of Figure 4.6). After this, the process proceeds by repeatedly applying leaf-compression (lines `7-19` in Figure 4.6) and path-compression (lines `21-41`). 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.

```
0:  Build TP instance for the nodes of T ordered with a depth first visit.
1:  For each leaf v of T do
2:      create a node u in T_0;
3:      entry(u)=v;
4:      node(v)=u;
5:  while last T_i has more than one node
6:      //begin of leaf compression
7:      for each v in T_i
8:           if (entry(parent(node(v))) not defined)
9:                {create w in T^a_{i+1};
10:               entry(parent(node(v)))=w;
11:               node(w)=parent(node(v));
12:               leaf-compr(w)=true};
13:          add v to children(w);
14:          parent(v)=w;
15:     for each w in T^a_{i+1}
16:          if (w is a leaf in T^a_{i+1})
17:               is-leaf(w)=true;
18:          else
19:               is-leaf(w)=false;
20:     //begin of path compression
21:     for each v in T^a_{i+1} such that is-leaf(v)=true
22:          create w in T_{i+1};
23:          leaf-compr(w)=false;
24:
25:          iteration
26:               parent(v)=w;
27:               add v in children(w);
28:               if (node(v)==root) break iteration;
29:               if (parent(v) has at least 2 sibling in T^a_{i+1})
30:                    break iteration; //the path is ending on v
31:               else
32:                    if (entry(parent(node(v))) not defined)
33:                         {create x in T_{i+1};
34:                         entry(parent(node(v)))=x;
35:                         node(x)=parent(node(v));}
36:                    v'=entry(parent(node(v)));
37:                    entry(parent(node(v)))=w;
38:                    v=v';
39:          iterate;
40:          node(w)=node(v);
41:     //end of path compression
42:  end while
43:  preprocess the horizontal tree adding p-lists
```

Figure 4.6: nca : Static Preprocessing Algorithm

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 $\sum_{v \in T} 1 + 1 = 2n$.

□

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 ($entry(parent(v)) = w$) is added to the tree T^a_{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^a_{i+1} we set the flag $is\text{-}leaf(w) = true$ if w is a leaf in T^a_{i+1} (lines `15-19`). The $is\text{-}leaf$ flag indicates the nodes from where the next path-compression will start.

Observe that the determination of the leaves of the tree can be performed very efficiently. A simple way to accomplish this is to maintain a copy of T, and remove nodes from it as they get compressed. In the copy of the tree we can keep the frontier as a linked list. This can be accomplished in $O(1)$ time on PPMs [46].

Path compression: Path-compression is initiated from each node v in T_i^a, such that $is\text{-}leaf(v) = true$. Execution of path-compression starting from v leads to the addition of a node w to the tree 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.

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 $parent(v)$, if $parent(v)$ 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 before).

Final Preprocessing 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 this data structure have been described in [44] and summarized in Section 5.1. 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 $depth(v)$ 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. It is possible to improve this pre-processing time, reducing it to $O(n)$, through the use of the MicroMacroUniverse approach described in [2] and in Section 4.6: in this case H is partitioned in μTrees (MicroTrees) with depth at most $\lg \lg n$.

Answering nca Queries:

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

1. Compute the $\mathsf{nca}(entry(x), entry(y)) = (z, z_x, z_y)$, $z \in H$. The computation can be performed using the algorithm in [44], based on the p-lists described in Section 5.1.

2. If $leaf\text{-}compr(z) = true$ then return ($node(z)$,$node(z_x)node(z_y)$).

3. Otherwise z is the result of a path compression. In this case if z_x is higher than z_y in H, the nca is (node(z_x), w_1, w_2), where w_1 is the node corresponding to the child of z_x that is ancestor of $entry(x)$ 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 z_y is higher than z_x is symmetric.

The test to check if z_x is higher than z_y can be performed in time $O(\lg \lg n)$ using the previously maintained depth information.

4.5 Applications of Optimal nca Algorithm

The availability of an optimal solution to the nca problem allows us to improve the solution of other interesting problems on PPMs. In particular it allows us to obtain an optimal solution to the generalized linked list problem [57]. This problem is the maintenance of

a linked list where new elements x can be inserted immediately after any existing element y. The two operations allowed are: insert(x, y) and compare(x, y) which returns true iff x occurs before y in the list. The following algorithm can be used to optimally solve this problem on PPMs.

To preserve the relationships between inserted nodes, we maintain a tree T' processed to answer nca queries and a data structure maintaining a Temporal Precedence order. Every time an insert(x, y) is done, the node x is inserted as rightmost child of y in T'. Since the tree T' cannot support an ordering of children of a node, we also insert the node y in the Temporal Precedence data structure. Note that a leftmost depth first visit of T' reconstructs the list. Thus the nca of two nodes either precedes both the nodes in the order or is equal to one of them. To answer a query compare(x, y), we find the nca(x, y)=(z, z_x, z_y) in T. If z=z_x then return true, because x=z and x is an ancestor of y in T'. If z=z_y then return false, because y is an ancestor of x and y cannot have been inserted before x. Otherwise return precedes(z_x, z_y) in the Temporal Precedence order.

Another problem whose solution can be improved using this optimal nca solution is the OP-Problem described above (see Chapter 3. The $O(\lg^2 n)$ solution proposed can be improved to a $O(\lg n \lg\lg n)$ solution by using the optimal generalized linked list scheme proposed above.

In fact, all the open problems described in [44] can be solved optimally on PPM using the optimal nca solution presented here.

4.6 An Algorithm for Dynamic nca Queries

The algorithm follows an approach similar to the MicroMacroUniverse described in [2, 20] in conjunction with repeated use of the static algorithm described in Section 4.4. Let n be the number of nodes in T. We define two kinds of trees (see Figure 4.7):

- The *μTrees* (Micro-Trees) are trees in a forest S of disjoint subtrees partitioning T.
- The *MTree* (Macro-Tree) is a tree collecting the roots of μTrees.

The MTree essentially compresses the nodes of each μTree into its root node and preserves the structure of T on these resulting root nodes. The height of each μTree is restricted to be at most $c_T(n)$. When a node v is added to T, if the μTree containing $parent(v)$ has a depth greater than $c_T(n)$, then a new μTree rooted in v is created. To obtain the optimal result, the MicroMacroUniverse approach is applied again to the μTrees. For the MTree the scheme used is based again on partitioning the tree into disjoint subtrees. However this partitioning is more dynamic in nature in the sense that 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 μTree. We denote by nca$_S$ the nca algorithm used for the μTrees in S and nca* 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 4.4, this information will be required to perform an nca query using p-lists.

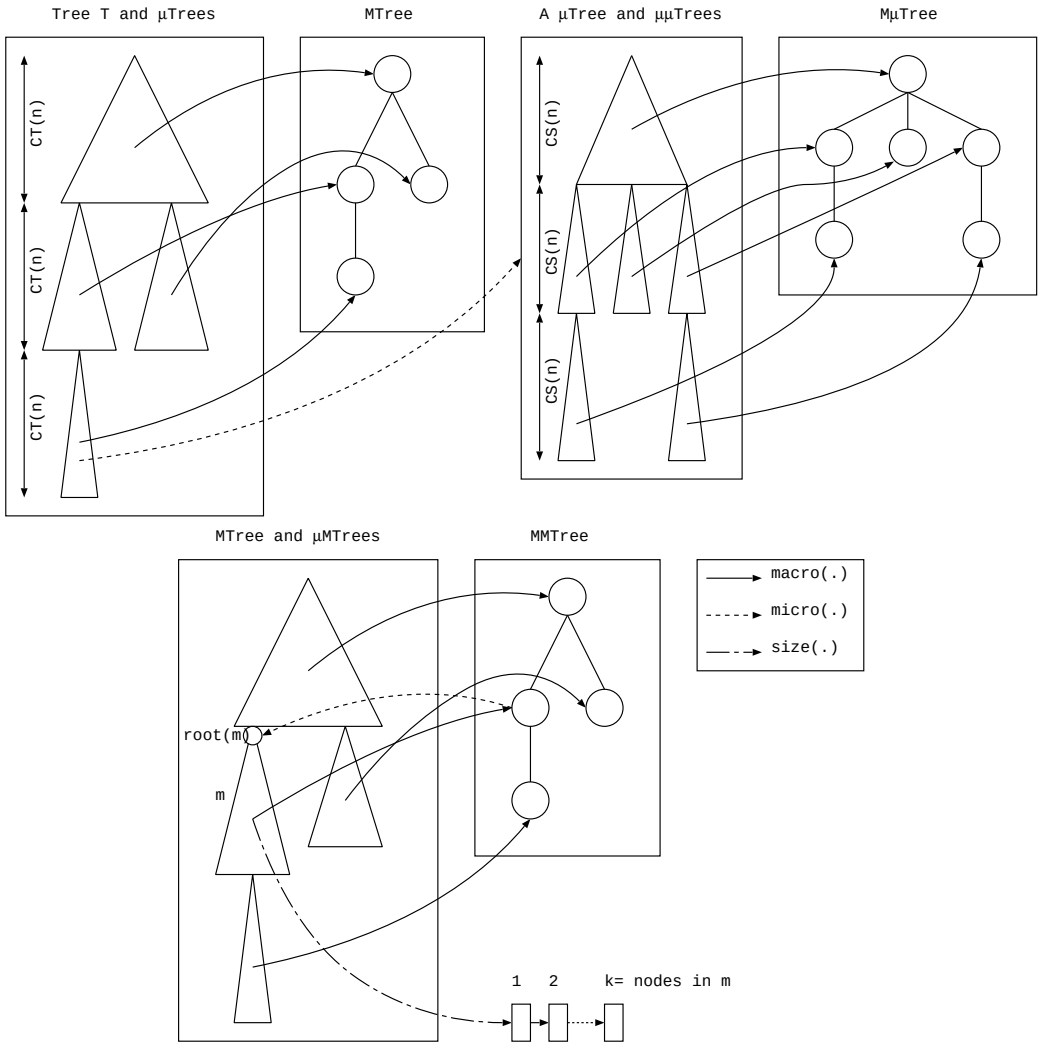

Figure 4.7: nca : Tree Structures Involved In The Dynamic Algorithm

4.6.1 The nca* algorithm

We present an algorithm to compute the nca with a cost of $O(N + Q \lg \lg N)$, 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 4.4. Let us call each of these trees *μMTree* (Micro-Macro-Tree). The preprocess of a

μMTree allows us to efficiently solve each nca query on that tree, using the nca_H algorithm presented earlier. Each root of a μMTree is represented by a node in another tree, called *MMTree* (Macro-Macro-Tree). We will show that the MMTree has a "small" depth—thus, simpler nca algorithms (e.g., the one based on p-lists [44]) can be used here to provide efficient nca computations.

Let the *preprocess* of a μMTree T_m be the static preprocess described in Section 4.4 applied to the tree $subtree_{MTree}(root(T_m))$. Thus, when T_m is pre-processed, all other $\mu\mu$Trees hanging on T_m are merged in a single new μMTree. The basic idea is to wait to re-preprocess a μMTree T_m until the number of nodes in $subtree_{MTree}(root(T_m))$ has doubled since the last preprocess of T_m.

To answer an nca query we first solve the problem with the p-list nca algorithm (Sections 4.4 and 5.1) on the MMTree and then we "refine" that solution using the nca_H on the μMTree associated to the result obtained from the MMTree.

Dynamic insertions: Recall that the MTree is partitioned into a set of μMTrees. Each μMTree is represented by its root in another tree, called MMTree.

Let T_m be a μMTree and v_m the node representative of T_m in the MMTree. Let us also define $root(T_m)$ the root of T_m in MTree, $micro(v_m)$ a pointer to $root(T_m)$, $macro(T_m)$ a pointer to v_m. We also maintain the size of T_m by keeping a pointer $size(T_m)$ that points to a list that has length $|T_m|$. $size(T_m)$ is created and inserted with a number of nodes equal to the number of nodes in the $subtree_{MTree}(root(T_m))$, when T_m is pre-processed. The $size(T_m)$ list is used a decrementing counter to decide when to do another preprocess.

Each time a node v is inserted in the MTree as child of w, a new μMTree T_m is created and the representative of $root(T_m) = v$ is added in the MMTree as child of $macro(T_w)$, where T_w is the μMTree containing w. For each μMTree T'_m corresponding to a node on the path P from $macro(T_m)$ to the root of the MMTree, we update the number of new nodes added in the $subtree_{MTree}(root(T'_m))$ by 1. This can be done decrementing the "counter" $size(T'_m)$ by one, that is, shifting the pointer $size(T'_m)$ to the next node in that list. If a μMTree T_m has consumed all nodes in $size(T_m)$, then T_m has to be pre-processed. Let us call v_h the highest node in the path P considered, such that $micro(v_h)$ has to be processed. The preprocess is applied on $subtree_{MTree}(v_h)$, which become the new μMTree. All nodes in $subtree_{MTree}(v_h)$ are deleted and replaced by the node v_h. The $size(micro(v_h))$ list is initialized with the insertion of a number of nodes equal to the number of nodes contained in $subtree_{MTree}(v_h)$.

As we will show in the next Lemma, the MMTree has depth less or equal to $O(\lg h)$, where h is the depth of the MTree. Thus the update of counters may be performed $O(\lg h)$ times for each insertion of a node in the MTree. Since a node is added in the MTree every $c_T(n)$ insertions in T and $h = n/c_T(n)$, if $c_T(n) \geq \lg n$, then the total time spent in updating the counters in all insertions in T is $O(n)$.

Notice that each time a node v is involved in a preprocess resulting in a tree T', the size

of T' is at least twice the size of the tree T'' which contained v before the preprocess. Then it follows:

Proposition 4.6.1 *A node v in the MTree of size t is involved in at most $\lg t$ distinct pre-processes.*

Lemma 3 *Immediately after a preprocess, if a path P starting from a node in MMTree and ending on a leaf has k nodes, then the total number of nodes in μMTrees 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 μMTree 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 pointed by v_l. Let us call T_m the μMTree represented by v_l in the MMTree. 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 should be preprocessed again and by hypothesis the MTree was just preprocessed. Thus the total number of nodes represented by the path P is at least $1 + 2^{i-2} + 2^{i-2} \geq 2^{i-1}$.

□

It follows that the MMTree has depth at most $\lg N$, where N is the number of nodes of the MTree. The MTree has at most $n/c_T(n)$ nodes. Choosing $c_T(n) = \lg n \lg\lg n$, the MTree has depth at most $O(n/(\lg n \lg\lg n))$. Applying lemma 3 we conclude that the MMTree has depth at most $\lg n$. Thus the nca in the MMTree 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 preprocess of a tree with t nodes in the static case costs $O(t \lg\lg t)$. Let N be the number of nodes in the MTree. From Proposition 4.6.1 we know that a node v in the MTree is involved in at most $\lg N$ preprocesses. Each of them will cost $l \lg\lg l$, where l is the size of the tree preprocessed. Thus for each v the amortized cost per process is $\lg\lg l \leq \lg\lg N$ 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: We show now how to compute the $\mathsf{nca}^*(x, y)$ with x and y in MTree. It is possible to find x' and y' μMTrees 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 μMTree. If x' and y' are in the same μMTree T_m return $\mathsf{nca}_H(x, y)$ using the previously pre-processed H tree for T_m. Otherwise, we can compute the $\mathsf{nca}(root(x'), root(y')) = (z, z_x, z_y)$ on the

MMTree, using p-lists. If $z_x = z$ then the result is given by $\mathsf{nca}_H(x, parent(micro(z_y)))$.[1] Otherwise, the result is $\mathsf{nca}_H(parent(micro(z_x), parent(micro(z_y))$.

The algorithm nca_H requires $O(\lg \lg n)$ time and the p-list algorithm used for the MMTree 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 total time is $O(\lg \lg n)$.

4.6.2 The nca_S 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 μTrees.

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

4.6.3 Handling deletions

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 preprocessing. We don't update the counters when nodes are deleted. This doesn't affect our analysis, because the number of operations is greater than the number of nodes in T.

4.7 From Sequential to Parallel

We can translate the sequential algorithm described in the previous sections to a parallel algorithm that works in parallel time $O(\lg^2 n)$ and uses $O(n)$ processors. Assume that a processor has been assigned to each node of the tree; a leaf compression can then be simulated in constant parallel time, because each processor can check independently if its node is a leaf. Path compression can be realized using the technique of pointer doubling [19]. Thus, the parallel time to realize a path compression is $O(\lg l)$, where l is the length of the longest path. Since l is bounded by n, each sequential path compression step can be simulated in $O(\lg n)$ parallel time. Since the total number of compressions in the sequential algorithm is $O(\lg n)$, this direct simulation of the sequential algorithm requires $O(\lg^2 n)$ parallel time. Unfortunately, this direct simulation may also require $\Omega(\lg^2 n)$ time. Consider, for example, the situation in Figure 4.8. 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

[1] The case $z = z_y$ is symmetrical.

take place; this path compression will require $\lg l$ time. The process is repeated k times, hence the total parallel time is $k \lg l$. If l is chosen equal to 2^k, then the total number of nodes $n = \Theta(2^{2k})$, thus $k = \Theta(\lg n)$ and the parallel time is $k^2 = \Theta(\lg^2 n)$.

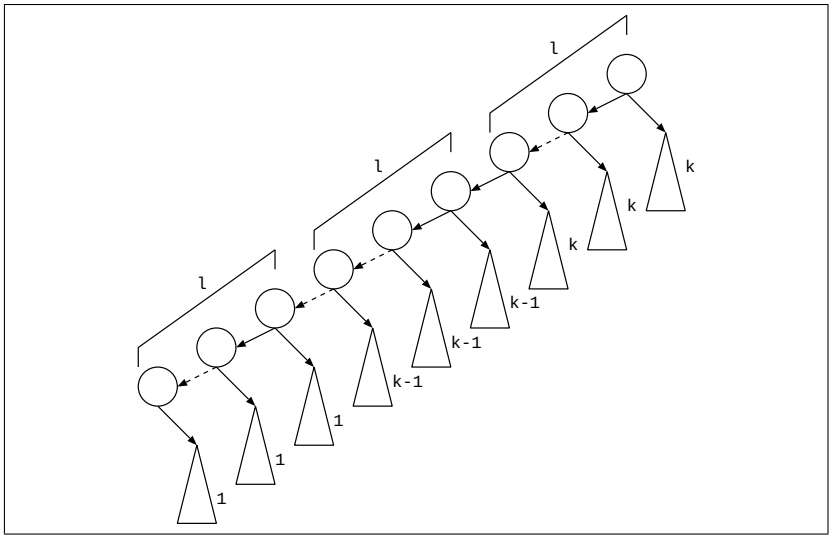

Figure 4.8: nca : An example of slow computation

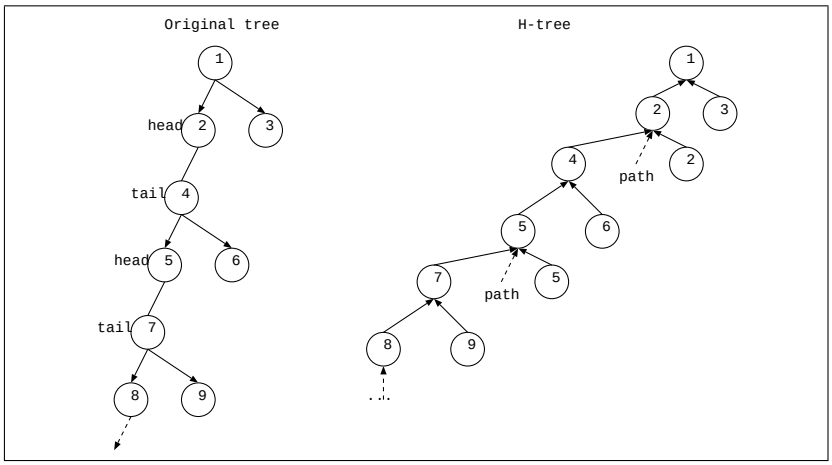

Figure 4.9: nca : An example of bad H-tree

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 Figure 4.9, 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 loosing 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.

4.8 A Parallel Compression Scheme for Trees

The compression algorithm we propose starts from the initial tree T and repeatedly performs two types of compressions, thus generating a sequence of trees: $T_0, T_1^a, T_1, T_2^a, T_2, \ldots$ until the compression process is finished. The final tree in this sequence is the H-Tree.

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 [19].

The development of our efficient parallel solution for the nca problem requires also the ability to solve in parallel the *Temporal Precedence Problem* [47]. This problem can be defined as follows: given a list L with l nodes representing an ordered sequence of objects, we want to answer the query precedes(x, y), where x, y are pointers to nodes in that list. We show that with l processors and $O(\lg l)$ parallel time preprocessing, each query can be answered in parallel time $O(\lg \lg l)$ by a single processor. This solution is presented in Section 5.4.

4.9 An Algorithm for Parallel nca

We start by introducing some notation. 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. Given a tree T_i, the result of step a applied to T_i is the tree T_{i+1}^a and the result of step b applied to T_i^a is T_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 $m_i^a(v) = L$ $(m_i(v) = L)$. 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 π to an ancestor of v at parallel time $i(a)$ $(i(b))$ and we denote it with $\pi_i^a(v)$ $(\pi_i(v))$.

A *leaf compression* of a tree T_i is executed during a phase a and returns a tree T_{i+1}^a such that for each node v in T_i (see Figure 4.10):

(i) if $(m_i(v) = L$ and v currently has no sibling) then
$$\pi_{i+1}^a(v) \leftarrow \pi_i(p(v)) \text{ and } m_{i+1}^a(\pi_{i+1}^a(v)) \leftarrow L;$$
(ii) if $(m_i(v) = L$ and v has a sibling z and $m_i(z) =?)$ then

v is merged with its parent $\pi^a_{i+1}(v) \leftarrow NULL$;

(iii) if $(m_i(v) =?$ and $((v$ has a sibling z and $m_i(z) = L)$ or $(v$ currently has no sibling$)))$ then

$$\pi^a_{i+1}(v) \leftarrow \pi_i(p(v));$$

(iv) if $(m_i(v) = L$ and v has a left sibling z and $m_i(z) = L)$ then

v is merged with its parent and $\pi^a_{i+1}(v) \leftarrow NULL$;

(v) if $(m_i(v) = L$, v has a right sibling z and $m_i(z) = L)$ then

$$\pi^a_{i+1}(v) \leftarrow \pi_i(p(v)) \text{ and } m^a_{i+1}(\pi^a_{i+1}(v)) \leftarrow L.$$

A *path compression* of a tree T^a_i is executed during a phase b and returns a tree T_i, such that for each v in T^a_i, $\pi_i(v) \leftarrow \pi^a_i(\pi^a_i(v))$ and if $m^a_i(v) = L$ then $m_i(\pi_i(v)) \leftarrow L$ (see Figure 4.11).

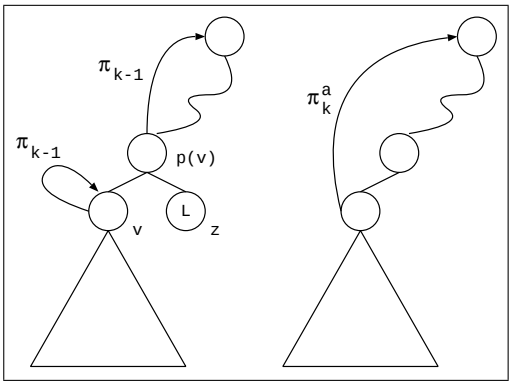

Figure 4.10: nca : Example of Leaf compression for node x

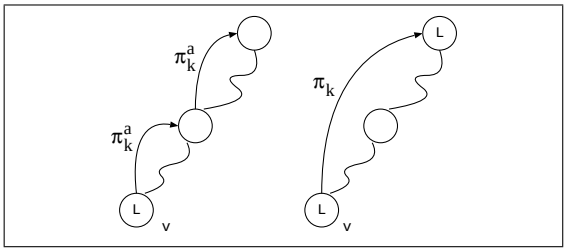

Figure 4.11: nca : Example of Path compression for node x

If T is the initial tree, then the tree T_0 is a copy of T, such that for each v in T_0, $\pi_0(v) = v$ if v has a sibling, else $\pi_0(v) = p(v)$; in addition, for each leaf l of T_0, $m_0(l) = L$. The root is the only exception: $\pi_0(root) = root$.

Figure 4.12 provides an example of a compression. The nodes marked represent the nodes labelled L and the dashed pointers are the π pointers. The pointers π pointing to $NULL$ are not shown.

Definition 4.9.1 *A node x is* finished *after step k if one of the following holds:*

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

Observe that once a node x is finished, during next phases it will remain finished.

Definition 4.9.2 *A set S of nodes of T is a* frontier *if every path from root to a leaf passes*

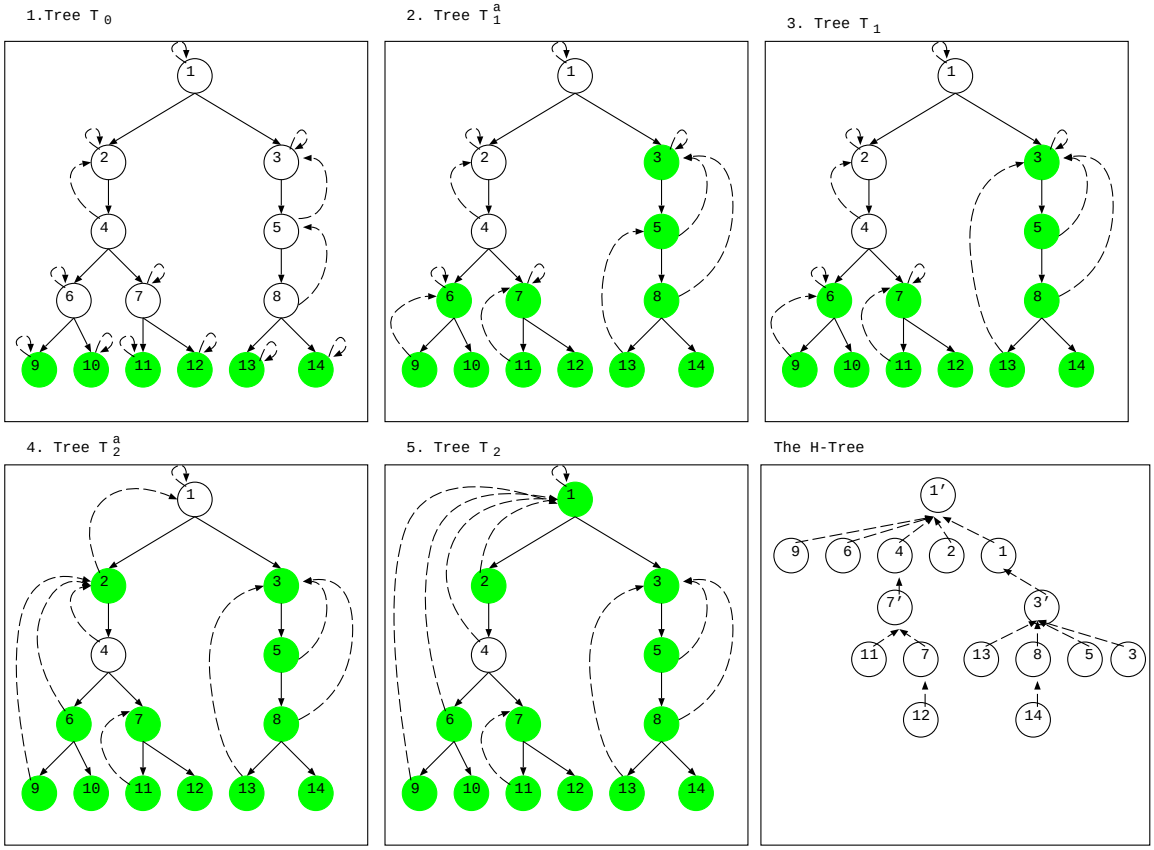

Figure 4.12: nca : An example of parallel compression

exactly through one of the nodes in S.

Observe that the set of nodes U that are unfinished and marked L is a frontier of T. Moreover, if x is unfinished after phase k, then there exists a descendent y of x such that $m_k(y) = L$ and that is not finished after phase k. In fact, any one of the nodes in the part of the frontier U included in T_x is such a node.

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

Theorem 4.9.3 *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 $|T_{p(x)}| \geq 2^k + 1$;*
3. *x is unmarked and unfinished after parallel step k, and either $\pi_k(x) = root$ or $(|T_{p(\pi_k(x))}| - |T_x| \geq 2^{k-1}$ and $|T_x| \geq 2^k + 1)$.*

Proof. We prove this result by induction on k.

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 $|T_{p(x)}| \geq 2^0 = 1$. If x is unmarked and unfinished, x is an internal node in T. In this case if x is the root then $\pi_0(x) = root$ else $|T_{p(x)}| - |T_x| \geq 2^{0-1} = 2^{-1}$ and $|T_x| \geq 2^0 = 1$.

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 marked L after phase $k' \leq k - 1$.

   In this case, one of the following is true:

   (*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 $\pi_{k'}(x) = NULL$;

   (*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.

2. 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 $x' = p(x)$. 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 step k. If x has a sibling z after step $k(a)$, then z could not have been marked L after step $k-1$, 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, $|T_{p(x)}| = |T_{x'}| = |T_z| + |T_x| + 1 \geq 2^k + 1$.

   Case 2: x is marked L during step $k(b)$. If x is the root, then it is finished after step k and the claim holds. Otherwise let $x' = p(x)$.

   Suppose that x does not have any sibling after step $k(a)$. It follows that $\pi^a_k(x)$ 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 $\pi^a_k(\pi^a_k(u)) = x$. Let us denote with u' the node $\pi^a_k(u)$. Clearly $u' \neq x$, otherwise x would be marked in step $k(a)$. Note that u' will be marked L after step $k(a)$. It follows that $\pi_k(u') = \pi^a_k(\pi^a_k(u')) = \pi^a_k(x)$ which is a proper ancestor of x—since x does not have siblings after step $k(a)$. $\pi_k(u')$ is marked L during step $k(b)$. Therefore x is finished after phase k.

Suppose instead that x has a sibling z after step $k(a)$ (see Figure 4.13). Then $m_{k-1}(z) \neq L$ otherwise z would have been leaf compressed at step $k(a)$. 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 $|T_{p(x)}| = |T_{x'}| = |T_x| + |T_z| + 1 \geq 2^k + 1$.

3. x unmarked and unfinished after phase k.

   If x is the root, then $\pi_k(x) = root$ 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 Figure 4.14). If u is the root, the $\pi_k(x) = root$ and the claim holds. Otherwise, let u' be the parent of u, and $v = \pi_{k-1}(u')$. If v is the root, then $\pi_k(x) = v = root$ and the claim holds. Otherwise let $v' = p(v)$. By inductive hypothesis (case (2)) $|T_{p(\pi_{k-1}(x))}| - |T_x| \geq 2^{k-2}$ and $|T_{p(\pi_{k-1}(u'))}| - |T_{u'}| \geq 2^{k-2}$. If u has no siblings or has a sibling marked L after phase $k-1$, then $\pi^a_k(u) = \pi_{k-1}(u') = v$ (because of the leaf compression) and $\pi_k(x)$ is at least v or above. Then $|T_{p(\pi_k(x))}| \geq |T_{p(v)}| = |T_{p(\pi_{k-1}(u'))}| \geq |T_{u'}| + 2^{k-2} = |T_{p(\pi_{k-1}(x))}| + 2^{k-2} \geq |T_x| + 2^{k-1}$. Therefore $|T_{p(\pi_k(x))}| - |T_x| \geq 2^{k-1}$. 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 $|T_{p(\pi_k(x)}| - |T_x| = |T_{u'}| - |T_x| \geq |T_z| \geq 2^{k-1}$. 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 $|T_x| \geq |T_{p(y)}| \geq 2^k + 1$.

□

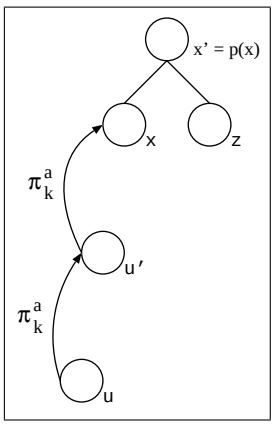

Figure 4.13: nca : x marked during step $k(b)$

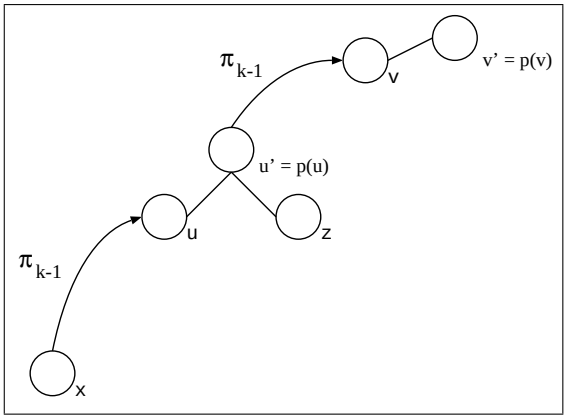

Figure 4.14: nca : x unmarked and unfinished after phase k

Corollary 4.9.4 *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 $\lg(n-1)+1$ *phases.*

Proof. The root r of T is unfinished (then unmarked) after phase $k-1$, otherwise k would not be the minimum integer. Let us consider a node x that is marked L and is unfinished after phase $k-1$. The node x must be a proper descendant of r. From Theorem 4.9.3 it follows that $|T| \geq |T_{p(x)}| \geq 2^{k-1}+1$.

□

If we have n processors (n being the number of nodes in the original tree) that have been assigned to the different nodes of T, then both leaf and path compressions will be performed in constant parallel time. From Corollary 4.9.4, the total number of compressions is at most $\lg n$, thus the total parallel time is $O(\lg n)$.

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.

4.9.1 The H-Tree

We provide now details of the construction of the H-Tree. The H-Tree built in the parallel scheme is the same described in Section 4.3. It is possible to reuse the π pointers to build H in constant parallel time. Once a node v is leaf compressed into its parent, $\pi(v)$ is set to NULL. At that time for every node w in the path having v as head, we have $\pi(w) = v$. We want to keep this information during the successive phases simply avoiding pointer doubling if a node is finished (line 26 in Figure 4.15). Once the compression is completed, every head of a path x has $\pi(x) = NULL$ and the π pointer for every other node y is pointing to the head of the path containing y.

Let us introduce another pointer p_H pointing to the parent of a node in H. During a leaf compression the pointer $p_H(v)$ is set to point to $p(v)$. Since the root is not leaf compressed, $p_H(root)$ is set to point to NULL. For each head x of a path l in T, a new node x' is created in H, $\pi(x') = x'$, $p_H(x') = p_H(x)$ and $\pi(x) = x'$. In a step of pointer doubling applied to the π pointers, every node in a path points to the new 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 Temporal Precedence Problem is set up. In constant time it is possible to identify the tails of path as follows. For each node v, 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 5.4.

The complete algorithm is presented in Figure 4.9. The lines marked with * are the ones necessary to setup the H tree.

4.9.2 Answering Parallel nca Queries Using H-Trees

The H-Tree can be used to answer nca queries in the same way as in the sequential case. Given the query $\mathsf{nca}(x, y)$, where $x, y \in T$, it is possible to answer the query using p_H and π pointers. In particular, computation of 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 now show that the H-Tree preserves enough nca information from T. Let z be the $\mathsf{nca}_H(x, y)$. If z is a representative of a leaf-compression, then z is also the 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) In this case, the nca of x and y is the highest node between z_i and z_j. Thus, answering an nca query in T boils down to computing an nca query in H.

From Corollary 4.3.1, we can infer that the height of H is $O(\lg n)$. In [44] it was shown that there is a PPM algorithm that, given a tree with height h, preprocesses 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. Although the preprocessing scheme presented in [44] is sequential, it can easily translated into a parallel scheme that uses n processors and $O(\lg h)$ parallel time for preprocessing. Using this result, we can preprocess the H-Tree in parallel time $O(\lg \lg n)$ using n processors. Then, the nca of x, y in H can be computed in worst-case time $O(\lg \lg n)$ using a single processor. This allows the computation of the nca in T with worst-case time complexity $O(\lg \lg n)$ using a single processor.

4.10 Parallel nca Algorithms

The algorithm described in Section 4.9 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 Figure 4.15). 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 3.8) because it would correspond to an unbound 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 $\pi_{down}(u)$ which is updated to point to $\pi_{down}(v)$ whenever the node has only one child v. In addition, if $\pi_{down}(v)$ 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.

Our algorithm requires n processors to perform the $O(\lg n)$ parallel time preprocess-

ing. After preprocessing, 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 (e.g., the one proposed by Schieber and Vishkin [51] and the one proposed by Berkman and Vishkin [5]) require $O(n/\lg n)$ processors and works in $O(\lg n)$ parallel time for preprocessing. After preprocessing, 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 don't incur a penalty in parallel time for preprocessing.

It is also important to observe that if we have any CROW 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 [17]) 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 always 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.

Note that if we are allowed only one processor to answer an nca query, then the time required must be at least $\Omega(\lg \lg n)$ [47]. Hence our algorithm is optimal in that regard. Observe also that the parallel time $O(\lg n)$ used to perform preprocessing is the best known for any parallel nca algorithm (including PRAM algorithms). If one were allowed arbitrary (e.g., n^3) number of processors, then it is possible to devise a Parallel PPM algorithm that requires $O(\lg n)$ parallel time for preprocessing and answers nca queries in time $O(1)$ [17]. This can be simply accomplished by precomputing all the answers in parallel (in time $O(\lg \lg n)$) and making a different processor responsible for each different possible query.

4.11 nca : Pointer Machines vs PPMs

The commonly used Pointer Machine model allows constant time arithmetic on $\Theta(\lg n)$ 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 $\Theta(\lg n)$ sized lists. This entails that a generic translation (that just simulates the arithmetic) of Pointer Machine algorithms to PPMs will incur a polylog penalty. More precisely an algorithm $\mathcal{A}$ that runs in time $t(n)$ on

a Pointer Machine and uses any arithmetic at all, will take time $t(n)\lg^k n$ for some $k > 0$ on a PPM. We present an interesting result about the nca problem. With the next theorem we show that any optimal Pointer Machine algorithm for the nca problem can be converted into a PPM algorithm without incurring *any* penalty.

Theorem 4.11.1 *A PPM algorithm $\mathcal{A}$ solving the* nca *problem with amortized cost of $O(\lg^k n)$ per insertion and worst-case cost $O(\lg\lg n)$ per query, can be translated into an algorithm on PPMs with an amortized cost of $O(1)$ per insertion and worst-case cost $O(\lg\lg n)$ per query.*

Proof. The tree T can be partitioned in μTrees with maximal depth of $\lg^k n$. Then the MTree collecting the μTrees has $n/(\lg^k n)$ nodes and the algorithm $\mathcal{A}$ requires a total of $O(n)$ time for insertions and $O(\lg\lg n)$ for the nca queries. The MicroMacroUniverse approach is applied again to the μTrees. Each μTree T_m is partitioned in $\mu\mu$Trees of depth at most $\lg\lg n$. The corresponding MμTree contains $\lg^k n/(\lg\lg n)$ nodes. The preprocess for a MμTree consists of attaching an $O(\lg\lg n)$ sized p-list to each node and can be accomplished in time $O(\lg^k n/(\lg\lg n)\lg\lg n) = O(\lg^k n)$. The total cost of preprocessing all the MμTrees, therefore, is $O(n/lg^k n) * O(\lg^k n) = O(n)$. The cost of an nca query on a $\mu\mu$Tree is $O(\lg\lg n)$ (brute force), on a MμTree is $O(\lg\lg n)$, because the depth of the tree is $O(\lg^k n)$, and on the MTree is $O(\lg\lg n)$ by hypothesis. In Section 4.6 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 $O(\lg\lg n)$.

□

Corollary 4.11.2 *The theorem remains true even if $\mathcal{A}$ was a Pointer Machine algorithm that made use of constant-time arithmetic for $\lg n$ size integers.*

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

□

```
1 :  pardo
2 :    if (v=root or v has a sibling) π_0(v) = v else π_0(v) = p(v);
3 :    if (v is leaf) m_0(v) = L else m_0(v) =?;
4 :  i:=0;
5 :  iterate
6 :    %leaf compression
7 :    pardo
8 :       if (m_i(v) = L and v has currently no sibling)
9 :            π^a_{i+1}(v) = π_i(p(v));
10:           m^a_{i+1}(π^a_{i+1}(v)) = L;
11:      elseif (m_i(v) = L and v has a sibling z and m_i(z) =?)
12:           π^a_{i+1}(v) = NULL;
13:*          p_H(v) = p(v);
14:      elseif (m_i(v) =? and ((v has a sibling z and m_i(z) = L) or
                  (v currently has no sibling)))
15:           π^a_{i+1}(v) = π_i(p(v));
16:      elseif (m_i(v) = L and v has a left sibling z and m_i(z) = L)
17:           π^a_{i+1}(v) = NULL;
18:*          p_H(v) = p(v);
19:      elseif (m_i(v) = L and v has a right sibling z and m_i(z) = L)
20:           π^a_{i+1}(v) = π_i(p(v));
21:           m^a_{i+1}(π^a_{i+1}(v)) = L;
22:      else π^a_{i+1}(v) = π_i(v);
23:    %path compression
24:    pardo
25:       π_{i+1}(v) = π^a_{i+1}(π^a_{i+1}(v));
26:*      if (π_{i+1}(v) = NULL) π_{i+1}(v) = π^a_{i+1}(v);
27:      if (m^a_{i+1}(v) = L) m_{i+1}(π_{i+1}(v)) = L;
28:    i:=i+1;
29:  loop until m_i(root) = L;
30:* p_H(root) = NULL;
31:* pardo x head of a path
32:*   create x' and π(x') = x';        % x' is a copy of x
33:*   p_H(x') = p_H(x);
34:*   π(x) = x';
35:* pardo p_H(x) = π(π(x));     % Each node in the path of x points to x'
36:* pardo if (v is tail)
37:*   TP preprocess on the list starting at v and ending at π_i(v)
```

Figure 4.15: nca : Parallel preprocessing Algorithm

Chapter 5

Implementation Details

5.1 An $O(\lg n)$ PPM Solution for nca

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 $anc(x, d)$ the ancestor of vertex x lying at depth d in the tree. Thus, if we have two nodes x and y and $anc(x, d) = anc(y, d)$, then we can infer that $\mathsf{nca}(x, y)$ is at a depth at least d in the tree. Otherwise:

$$\mathsf{nca}(x, y) = \mathsf{nca}(anc(x, d), anc(y, d))$$

In our solution, with each node in the dynamic tree we store the *depth* of the node. The depth is encoded as a binary number—by using a list of records containing *nil* and *non-nil* pointers to represents zeros and ones respectively. Each time a new node is created (expand operation) we can calculate its depth by adding one to the depth of the parent node. All this can be accomplished in worst-case time complexity $O(\lg h)$, where h is the depth of the node. In addition, we maintain for each node in the tree a list of pointers to selected ancestor nodes (the *predecessors list* described in the next subsection). These pointers are used to perform a binary search leading to the identification of the nca. The resulting data structure, discovered independently, resembles the one used in [58], but does not assume constant time arithmetic capabilities for manipulation.

The rest of this section provides details for realizing these ideas and studies the complexity of the resulting solution. We start by describing some of the data structures which we need to maintain to efficiently perform the tree operations required by the nca problem.

5.2 Predecessors List

To support the efficient calculation of the nca operation we need an additional data structure super-imposed on the dynamic tree: a *predecessors list (p-list)* attached to each node of the tree. Each element of the p-list is a record with two fields called *data* and *direction* (in addition to the fields required to maintain the linked list).

The p-list of node x contains $\lfloor \lg h \rfloor + 1$ 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 $1, 2, 4, 8, \ldots, 2^{\lfloor \lg h \rfloor}$ from node x.

Let us denote with $jump(x, k)$ the ancestor of node x which is at a distance k from x in the tree. The data field of the ith element of the p-list of x contains a pointer to the node $jump(x, 2^i)$, for $i = 0, 1, \ldots, \lfloor \lg depth(x) \rfloor$. The direction field of the ith element of the p-list of x contains the value L (which stands for "left") if x belongs to the left subtree of the node $jump(x, 2^i)$, and the value R otherwise.

The management of the p-lists is somewhat involved. Each time we add a new leaf to the tree, we need to create the p-list for each new node created. Let us show how this can be done.

5.2.1 Creating the p-lists

Let us consider the execution of the operation $\mathsf{add\_leaf}(x, y)$ that adds a node y as child of x. The data fields of the p-list of node y can be calculated from the p-list of node x: if the data field of the ith element of the p-list of x is $jump(x, 2^i)$, then the data field of the ith element of the p-list of y is a child of $jump(x, 2^i)$. If the direction field associated to $jump(x, 2^i)$ in the p-list of x is L (R), then the data field of the ith element in the p-list of y is going to be the *left* (*right*) child of $jump(x, 2^i)$. Thus, we can create the data fields of the p-list of y by scanning the p-list of x and picking the appropriate child of the data field of each element. Since the p-list of x contains $O(\lg h)$ elements, and determining the corresponding element in the p-list of y can be done in constant time, this process requires $O(\lg h)$ time (h height of the tree).

The only exception to this simple rule occurs when $depth(y)$ is an exact power of two—in which case we need to extend the p-list by adding one more element. The new element introduced in the list will point to the root of the tree. The only problem is *how* to determine when this case occurs. This can be achieved in $O(\lg h)$ by simply scanning the list of records representing the *depth* of the parent node and ensuring that it is composed only by $non-nil$ pointers (i.e., the binary number representing the depth of the parent node is composed only by ones).

5.2.2 Updating the Directional Fields

In order to efficiently maintain the directional fields in the p-list, we need to slightly modify the current picture. We introduce in each element of the p-list an additional field called *right-p-link (rplink)*, that points to an element belonging to the p-list of another node of the tree. In particular, the ith element of the p-list of node x contains a pointer to the $(i+1)^{th}$ element of the p-list of the node $jump(x, 2^i)$, if it exists (it is nil otherwise)—see Figure 5.1.

Let us introduce some simplified notation. Given a node x in the tree, we will denote with $last(p\text{-}list(x))$ ($first(p\text{-}list(x))$) a pointer to the last (first) record in the p-list of node x. If

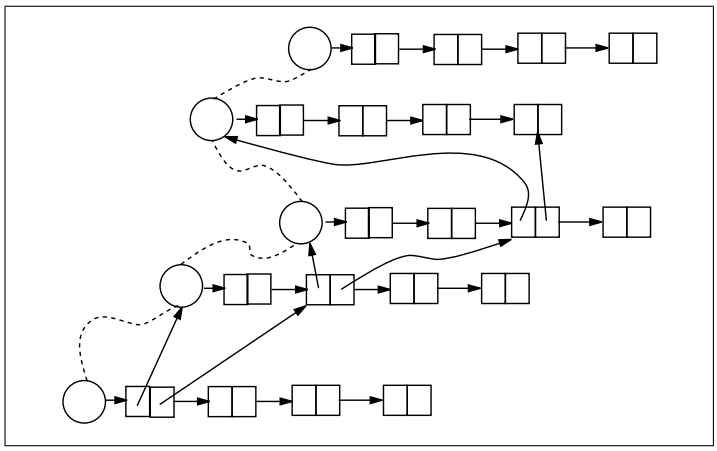

Figure 5.1: Right-p-links between Nodes of the p-lists

s is a pointer to a p-list element, then $data(s)$, $direction(s)$, $lplink(s)$ represent respectively the value of the *data*, *direction*, and *lplink* fields stored in the record pointed to by s. If s points to a p-list element, then $prev(s)$ ($next(s)$) is a pointer to the p-list element preceding (following) s.

The interesting property of this data organization is that the creation of a new p-list can be obtained by simply following the appropriate right-p-links and copying the p-lists nodes encountered. Let y be a new node which is inserted in the tree as the left (right) child of node x. The creation of the p-list for y can be performed as follows:

- the first element of the p-list for y contains *(i)* a pointer to the node x in the data field, *(ii)* the direction L (R) in the direction field, and *(iii)* a pointer to the second node in the p-list of x in the right-p-link field.
- all the other elements of the p-list for y can be obtained essentially by following the (right-p-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-p-links for this new list can be calculated simultaneously as illustrated in the following algorithm:

```
1:  scan := first(p-list)(x) ; // first element of p-list of x
2:  p-list(y) := new (); // create first element p-list of y
3:  data(p-list(y)) := x;
4:  rplink(p-list(y)) := next(p-list(x));
5:  if (y = left(x)) then
6:      direction(p-list(y)) := L;
7:  else
8:      direction(p-list(y)) := R;
9:  endif
10:  prev := p-list(y) ;
11:  while (scan ≠ nil) do
12:     next(prev) := copy(scan); // create copy of element scan
13:     rplink(prev) := next(scan);
```

```
14:     scan := rplink(scan);
15:     prev := next(prev);
16:  endwhile;
```

The p-list of the new element may require an additional element if the depth of the new node a power of 2 (this can be realized as already described in the previous subsection).

Example 5.2.1 *Consider the tree in Figure 5.2*[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 various nodes (the notation* m/L *means that the p-list element points to the node* m *and the direction is* L*). The Figure shows the p-list for 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-p-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-p-links of the new elements are shown as dashed lines.*

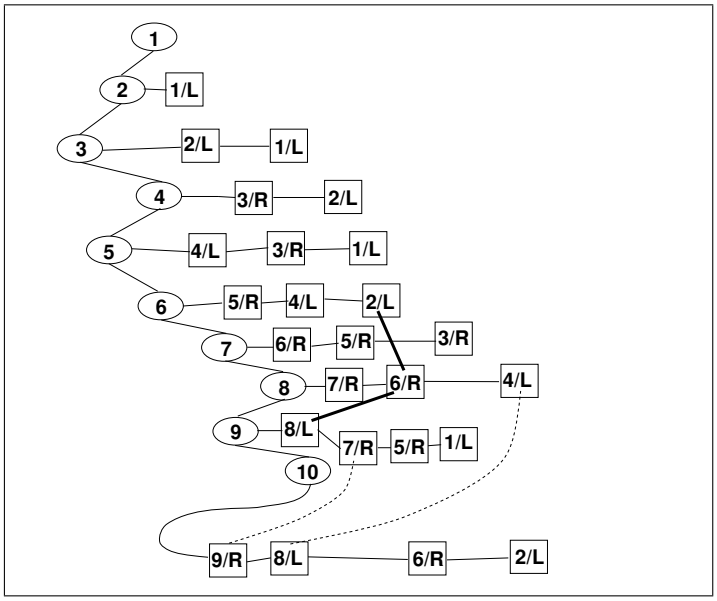

Figure 5.2: Creation of p-list: example

Note that the creation of the new p-list requires $\Theta(\lg h)$ time, where h is the depth of the newly created node. Hence, the expand operation altogether takes time $\Theta(\lg h)$.

The correctness of the algorithm follows by observing that the data and directional fields of $(i+1)^{th}$ element of p-list(y) are 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 p-list(y). Noting that $jump(jump(y, 2^i), 2^i) = jump(y, 2^{i+1})$, it is easy to establish inductively that the data and directional fields of p-list(y) are correctly computed. The fact that the right-p-links are correctly computed is also straightforward to establish.

[1] The Figure shows only the nodes on the path from the root to the new node.

The right-p-links are essential to guarantee a fast construction of the p-lists each time an expand operation is performed. To support the execution of the nca operation we will also require two additional fields in each element of the p-list. The first field is called *left-p-link (lplink)*. The left-p-link in the ith p-list element of the tree node x contains a pointer to the $(i-1)$th p-list element of the node $jump(x, 2^i)$. The second field is called *middle-p-link (mplink)*. The middle-p-link in the ith p-list element of the tree node x contains a pointer to the ith p-list element of the node $jump(x, 2^i)$. The left-p-links and middle-p-links can be maintained in a fashion similar to the right-p-links. Observe that it is not necessary to maintain all the three p-links; *e.g.*, the right-p-link can be determined from the middle-p-link. We use all the three p-links only for the sake of clarity.

```
1:  repeat
2:     stepx := last(p-list(x))
3:     stepy := last(p-list(y))
4:     if (data(stepx) ≠ data(step(y)))
5:          x = data(stepx);
6:          y = data(stepy);
7:     endif
8:  until (data(stepx) = data(stepy));
9:  while ( true ) do
10:    while ( data(stepx) = data(stepy) ) do
11:         stepx := prev(stepx);
12:         stepy := prev(stepy);
13:    endwhile;
14:    if (parent(data(stepx)) = parent(data(stepy)))
15:         then return ⟨ parent(data(stepx)), data(stepx), data(stepy) ⟩;
16:    else
17:         x := data(stepx);
18:         y := data(stepy);
19:         stepx := lplink(stepx);
20:         stepy := lplink(stepy);
21:    endif;
22:  endwhile;
```

Figure 5.3: Computation of the nca

5.3 Computing the nca

We subdivide the process of computing $\mathsf{nca}(x, y)$ 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 $depth(x) < depth(y)$ determine the ancestor y' of y such that $depth(y') = depth(x)$. It is clear that the ability to solve these two subproblems will provide a general solution to the task of determining $\mathsf{nca}(x, y)$ 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.

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 in the tree. Let h be the depth of these nodes in the tree. Then, the p-lists of x and y each contain $\lfloor \lg h \rfloor + 1$ elements. The computation of $\mathsf{nca}(x, y)$ is performed through the algorithm in Figure 5.3.

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 $jump(x, 2^i)$ and $jump(y, 2^i)$ for successively decreasing values of i (to avoid the use of arithmetic, we are simply moving pointers in the p-lists of x and y). The loop continues as long as the nodes reached are equal. All these are common ancestors for x and y. As soon as we reach two ancestors $jump(x, 2^i)$ and $jump(y, 2^i)$ that are different (see Figure 5.4 on the left) we can verify whether the nearest common ancestor has been reached or not at the previous step ($jump(x, 2^{i+1})$). This test can be easily performed by checking if the two nodes reached at jump i have the same parent. (line 14).

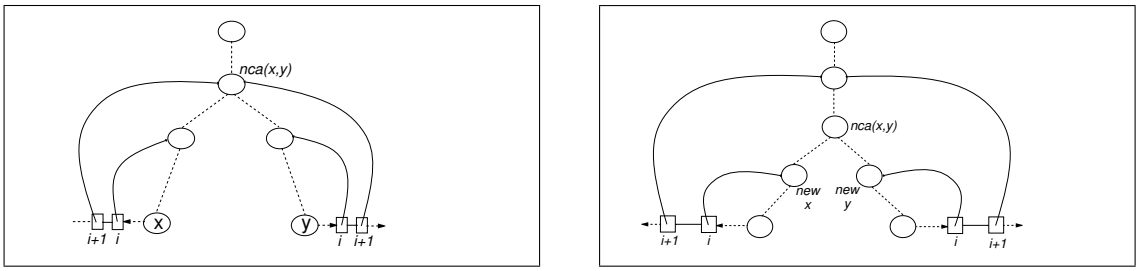

Figure 5.4: Searching for Nearest Common Ancestor

If the ancestor $jump(x, 2^{i+1})$ is not the nearest common ancestor (see Figure 5.4 on the right), then the algorithm repeats the computation by replacing the nodes x and y respectively with $jump(x, 2^i)$ and $jump(y, 2^i)$. From the tests made earlier, we can limit the search to the ancestors of distance up to 2^i—and this is accomplished by starting a new iteration of the loop in line 9 not by taking the last element of the p-lists of the new x and y but by taking the p-list elements pointed by the left-p-links (i.e., maintain the same jump distance).

Considering that the number of iterations performed is limited by the length of the longest p-list in the tree (that is $O(\lg h)$ for a tree of height h), the we can conclude that the algorithm has a complexity of $O(\lg h)$.

Determining Equal Height Ancestor

```
1:  j := depth(y) - depth(x);
2:  let j = j_t j_{t-1} ... j_0; // show digits of j
3:  step := p-list(y);
4:  current := y;
5:  for i := 0 to t do
6:      if ( j_i = 1 ) then
7           current := data(step);
8:          step := mplink(step);
9:      endif
10:     step := next(step);
11:  endfor;
12:  return current;
```

Figure 5.5: Determining Equal Height Ancestor

We provide a method that given two nodes x, y such that $depth(x) < depth(y)$, returns a node y' such that $y' \preceq y$ and $depth(y') = depth(x)$ (Figure 5.5). The method uses the depth information stored in the nodes to determine the appropriate jump necessary to find the ancestor y', and uses the p-lists to perform the jump to the correct ancestor.

The subtraction operation in line 1 creates a new list of records representing the binary number obtained from the subtraction of the depths of x and y. This operation can be performed in time $O(\lg h)$ (h is $depth(y)$). In the procedure, a jump of size 2^ℓ is performed from the `current` ancestor node of y if the digit j_ℓ is one. Therefore, all the jumps together add to a total jump of j from node y. The complexity of the procedure is $O(\lg h)$.

Comment: It is possible to work with p-lists where the directional information is not stored. In this case the p-lists of the nodes that are siblings are identical. These two facts can be used to save space and time as well as to extend the procedure to trees with unbounded arity.

For trees with unbounded degree, when an operation expand$(x, a_1, a_2, \ldots a_r)$ is performed only one p-list is created. Each of $a_1, a_2, \ldots a_r$ has a pointer to this p-list. Of course each of $a_1, a_2, \ldots a_r$ has a parent pointer to x. The time for this operation then clearly is $O(r + \lg h)$. Note that since the p-list is created only once this avoids the $\Omega(r \lg h)$ cost that the naïve method will have to incur. If we think of expand operation as adding leaves each add-leaf operation still takes only $O(\lg h)$ time. In fact, the $\Theta(\lg h)$ time is spent only for the first add-leaf operation at any node, thereafter each add-leaf takes only constant time.

5.4 A Parallel Algorithm For The TP Problem

The Temporal Precedence Problem was first defined in [47]. In the context of parallel computations, the Temporal Precedence Problem can be reformulated as follows: given a list L with l nodes representing an ordered sequence of objects, we want to answer the query

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 l processors, $O(\lg l)$ parallel preprocessing 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 all levels ordered, then the precedes(x, y) query can be answered comparing the children of nca(x, y) in BT.

All siblings in a level of BT are ordered according to a scan from left to right, using *sibl* pointers. BT is constructed by adding levels in parallel as follows. First of all, we need to determine the depth of BT: using a processor, L is scanned using the pointer doubling technique to compute $\lg l$. This number can be stored as a list of $\lg l$ nodes and is used to control the depth of BT.

Each node of BT has one processor associated to it. First the root of BT is created. Then, recursively and in parallel, each new processor p associated to a node t in BT checks its depth and, if a new level has to be added in BT, p executes the following operations: First, p adds two new nodes (t_l and t_r) with new processors associated to them. Then, the sibling pointer of t_l is set equal to t_r ($sibl(t_l) = t_r$), and $sibl(t_r)$ is set equal to the left child of $sibl(t)$.

The last level contains a list of siblings called S. 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 L (for simplicity we assume that the pointer to the next element in L is called *sibl* as well). 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 points to the *previous* element in the list. (see Figure 5.6).

Each node of L contains a pointer *map* that will be used to point to the corresponding node in S. Initially none of the *map* pointers is set. We now show how to compute these pointers. At the beginning the 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 Figure 5.7). This is followed by a step of pointer doubling in both S and L. In S pointer doubling is accomplished using the *sibl* pointers, while in L it is performed using the *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 preprocessing constructs the auxiliary data structures (called *p-lists*) to support the $O(\lg \lg l)$ computation of the nca in BT, using a straightforward parallelization of the algorithm presented in [44]. This preprocessing requires $O(\lg l)$ parallel time.

A precedes(x, y) query is answered by a single processor in $O(\lg \lg l)$ time. The idea

is to access the BT tree and determine (in $O(\lg \lg l)$ time using the p-list nca algorithm described in [44]) the nca z of $map(x)$ and $map(y)$. The tree maintains the information about precedence for each of its levels. Let z_x be the child of z on the path from z to x and z_y the other child of z. In constant time it is possible to compare z_x and z_y using the *sibl* pointer and answer true iff z_x is left sibling of z_y. The computation of the nca is accomplished in $O(\lg h)$, where h is the height of BT. Since BT by construction is a complete tree, its depth is $\lg l$ and the query requires $O(\lg \lg l)$ time.

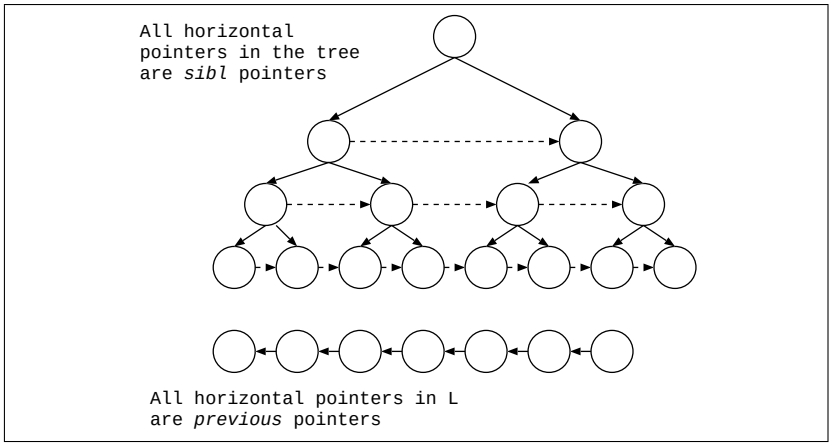

Figure 5.6: BT tree and list L

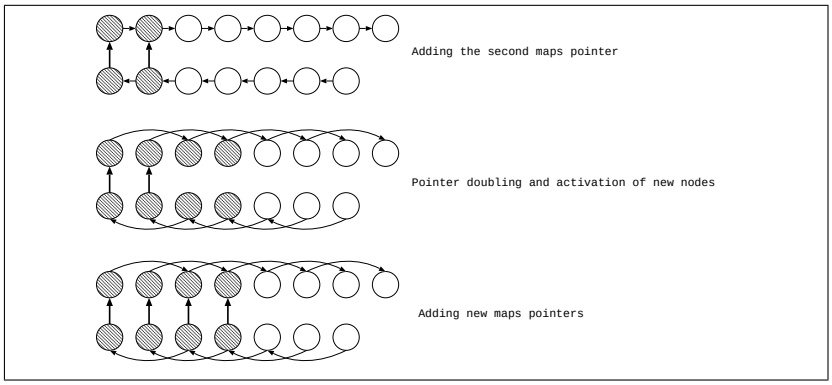

Figure 5.7: Example of mapping

Chapter 6

Conclusions

In this thesis we have revisited the OP-Problem, a fundamental problem arising in the management of non-deterministic and search-based computations—as those occurring in logic programming, constraint programming, parallel AI systems, as well as in the context of managing late bindings in Object Oriented Systems. The OP-Problem is a common abstraction of these real-life problems, expressed as a data structure problem on dynamic trees.

We have provided a new solution to the OP-Problem. The solution we propose is considerably more efficient than all previous solutions described in the literature, and it is *optimal* for the Pointer Machine model. It also provides an exponential improvement over the previously proposed solutions for this problem.

We have also shown how the solution can be extended to handle multiple occurrences of the same attribute along branches of the tree — this property is required when modelling other search problems, such as those arising from the management of late binding in an object-oriented system. The resulting algorithm provides, in this generalized problem, the same complexity as for the original OP-Problem — thus providing an optimal solution also for this problem on Pointer Machines.

The proposed solution is constructed using fairly standard data structures (e.g., precedence lists and balanced search trees) and it is very practical. The data structure we propose is currently being introduced in the PASL system — a fast or-parallel implementation of Prolog.

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 algorithm requires a linear preprocessing time in the static case and $O(\lg \lg n)$ time to answer a query. 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 remarkable theorem 4.11.1, making use of the MicroMacroUniverse scheme presented in Section 4.6.

We also presented an efficient Parallel Pointer Machine algorithm for the nca problem. 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. The solution presented is a parallelization the optimal sequential solution for the nca problem presented in Section 4.4. Our nca algorithm required an efficient parallel solution of a parallel version of the Temporal Precedence problem [47]. We provided an efficient Parallel Pointer Machine algorithm to solve this problem.

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 pointer machines with arithmetic 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 Pointer Machines with arithmetic. As of now, we believe that the worst such known penalty incurred is $O(\lg \lg n)$ [47]. 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 always non-optimal.

Acknowledgments

I would like to thank my relator Prof. Roberto Giacobazzi, for his kindness and help during the last phase of this work which began in the United States.

I wish to thank Prof. Enrico Pontelli and Prof. Desh Ranjan, for the great and productive time I spent with them at New Mexico State University. I cannot forget our talks and full blackboards, while we were working at our papers, basis of this thesis.

I also would like to thank Prof. Agostino Dovier, who motivated my interest in doing research and gave me the chance to spend a semester at New Mexico State University. He always helped and supported me in my choices during my formation.

List of Figures

Bibliography

[1] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *1990 N. American Conf. on Logic Prog.*, pages 757–776. MIT Press, 1990.

[2] S. Alstrup and M. Thorup. Optimal Pointer Algorithms for Finding Nearest Common Ancestors in Dynamic Trees. *Journal of Algorithms*, 35:169–188, 2000.

[3] A.M. Ben-Amram. What is a Pointer Machine? Technical report, DIKU, University of Copenhagen, 1995.

[4] M.A. Bender and M. Farach-Colton. The LCA Problem Revisited. In *Proceedings of LATIN 2000: Theoretical Informatics, Latin American Symposium*, pages 88–94. Springer Verlag, 2000.

[5] O. Berkman and U. Vishkin. Recursive *-Tree Parallel Data Structure. In *FOCS*, 1989.

[6] N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM Journal on Computing*, 15(4):1021–1024, 1986.

[7] L. Brownston. *Programming Expert Systems in OPS5: an introduction to rule-based programming*. Addison-Wesley, 1985.

[8] A.L. Buchsbaum et al. Linear-Time Pointer-Machine Algorithms for Least Common Ancestors, MST Verification, and Dominators. In Procs. ACM STOC, ACM Press, 1998.

[9] A. Church, *A Note on the Entscheidungsproblem*. Journal of Symbolic Logic, 1, 40-41, 1936.

[10] R. Cole and R. Hariharan. Dynamic LCA Queries on Trees. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, pages 235–244. ACM/SIAM, 1999.

[11] S.A. Cook and P.W. Dymond. Parallel Pointer Machines. Computational Complexity, 3:19–30, 1993.

[12] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[13] A. Dal Palú, E. Pontelli, and D. Ranjan. An Optimal Algorithm for Finding NCA on Pure Pointer Machines. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, 2002 (to appear)

[14] A. Dal Palú, E. Pontelli, and D. Ranjan. An Efficient Parallel Pointer Machine Algorithm for Nearest-Common Ancestor Problem. In *IFIP International Conference on Theoretical Computer Science*, 2002 (to appear)

[15] K. Driesen and U. Holzle. Minimizing Row Displacement Dispatch Tables. In *Conf. on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 1995.

[16] K. Driesen and U. Holzle. The Direct Cost of Virtual Function Calls in C++. In *Conf. on Object Oriented Programming Systems, Languages, and Applications*. ACM Press, 1996.

[17] P.W. Dymond, F.E. Fich, N. Nishimura, P. Ragde, and W.L. Ruzzo. Pointers versus Arithmetic in PRAMs. In *Structures in Complexity Theory Conf.*, IEEE, 1993.

[18] R. Finkel, V. Marek, N. Moore, and M. Truszczyński. Computing Stable Models in Parallel. In A. Provetti and S.C. Tran, editors, *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, pages 72–75, Cambridge, MA, 2001. AAAI/MIT Press.

[19] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *STOC*, ACM Press, 1978.

[20] H.N. Gabow and R. E. Tarjan A linear-time algorithm for a special case of disjoint set union *J. Comput. System Sci* 30 (1985), 209-221.

[21] A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.

[22] L.M. Goldschlager. A Universal Interconnection Pattern for Parallel Computers. In Journal of the ACM, 29, 1982.

[23] M.T. Goodrich and S.R. Kosaraju. Sorting on a Parallel Pointer Machine with Applications to Set Expression Evaluation. In FOCS, IEEE, 1989.

[24] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In P. van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming*, pages 93–109, Cambridge, MA, 1994. MIT Press.

[25] G. Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM Transactions on Programming Languages and Systems*, 15(4):659–680, 1993.

[26] G. Gupta, E. Pontelli, M. Carlsson, M. Hermenegildo, and K.M. Ali. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 2001. (to appear).

[27] G. Gupta, V. Santos Costa, and E. Pontelli. Shared Paged Binding Arrays: A Universal Data-structure for Parallel Logic Programming. Proc. NSF/ICOT Workshop on Parallel Logic Programming and its Environments, CIS-94-04, University of Oregon, Mar. 1994.

[28] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1999.

[29] D. Harel and R.E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestor. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[30] U. Holzle and D. Ungar. A Third Generation SELF Implementation. In *Conf. on Object Oriented Programming Systems, Languages, and Applications.* ACM Press, 1994.

[31] J.W. Hong. On Similarity and Duality of Computation. In FOCS, 1980.

[32] N.D. Jones Constant time factors do matter. Proc. 25th Annual ACM Symp. on Theory of Computing, 602–611, 1993.

[33] D.E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[34] A.N. Kolmogorov and V.A. Uspenskii. On the Definition of an Algorithm. *Uspehi Mat. Nauk.* 13, 1958.

[35] H. LaPoutré. Lower Bounds for the Union-Find and the Split-Find Problem on Pointer Machines. *Journal of Computer and System Sciences*, 52:87–99, 1996.

[36] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Heidelberg, 1987.

[37] E. Lusk, R. Butler, T. Disz, R. Olson, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, P. Brand, M. Carlsson, A. Ciepielewski, B. Hausman, and S. Haridi. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2/3):243–271, 1990.

[38] J. Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages.* PhD thesis, Uppsala University, 1997.

[39] E. Pontelli and O. El-Kathib. Construction and Optimization of a Parallel Engine for Answer Set Programming. In I. V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 288–303, Heidelberg, 2001. Springer Verlag.

[40] E. Pontelli, D. Ranjan, and G. Gupta. On the Complexity of Parallel Implementation of Logic Programs. In S. Ramesh and G. Sivakumar, editors, *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 123–137, Heidelberg, 1997. Springer Verlag.

[41] E. Pontelli, D. Ranjan, and G. Gupta. The Complexity of Late Binding in Dynamic Object-Oriented Programming Languages. *Journal of Functional and Logic Programming*, 1999. (to appear).

[42] D. Ranjan, E. Pontelli, and G. Gupta. On the Complexity of Or-Parallelism. *New Generation Computing*, 17(3):285–308, 1999.

[43] E. Pontelli and D. Ranjan. A Simple Optimal Solution for the Temporal Precedence Problem on Pure Pointer Machines. Tech. Report TR-CS-006/2001, New Mexico State University, 2001.

[44] E. Pontelli and D. Ranjan. Ancestor Problems on Pure Pointer Machines. In *LATIN*, Springer Verlag, 2002 (to appear).

[45] D. Ranjan, E. Pontelli, and A. Dal Palú. An Optimal Data Structure to Handle Dynamic Environments in Non-deterministic Computations. Tech. Rep. TR-CS-005/2001, New Mexico State University, 2001.

[46] D. Ranjan, E. Pontelli, and G. Gupta. Data Structures for Order-Sensitive Predicates in Parallel Nondetermistic Systems. *ACTA Informatica*, 37(1):21–43, 2000.

[47] D. Ranjan, E. Pontelli, L. Longpre, and G. Gupta. The Temporal Precedence Problem. *Algorithmica*, 28:288–306, 2000.

[48] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie Mellon, Pittsburgh, 1989. School of Computer Science.

[49] J.H. Saunders. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, 1(6), 1989.

[50] C. Schulte. Parallel Search Made Simple. In N. Beldiceanu et al., editor, *Proceedings of Techniques for Implementing Constraint Programming Systems, Post-conference workshop of CP 2000*, number TRA9/00, pages 41–57, University of Singapore, 2000.

[51] B. Schieber and U. Vishkin. On Finding Lowest Common Ancestors. *SIAM J. Comp.*, 17:1253–1262, 1988.

[52] A. Schönhage. Storage Modification Machines. *SIAM Journal of Computing*, 9(3):490–508, August 1980.

[53] K.V. Shvachko. Different Modifications of Pointer Machines and their Computational Power. *Proc. Symp. MFCS, Springer Verlag*, 1991.

[54] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In E. Lusk and R. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 713–732, Cambridge, MA, October 1989. MIT Press.

[55] A. Taivalsaari. On the Notion of Inheritance. *Computing Surveys*, 28(3), 1996.

[56] R.E. Tarjan. A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets. *Journal of Computer and System Sciences*, 2(18):110–127, 1979.

[57] A. Tsakalidis. Maintaining Order in a Generalized Linked List. *ACTA Informatica*, (21):101–112, 1984.

[58] A. Tsakalidis. The Nearest Common Ancestor in a Dynamic Tree. *ACTA Informatica*, 25:37–54, 1988.

[59] A. Turing, On Computable Numbers, With an Application to the Entscheidungsproblem Proceedings of the London Mathematical Society, Series 2, Volume 42, 1936.

[60] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[61] J. von Neumann. The general and logical theory of automata. In *L. A. Jeffress, editor, Cerebral Mechanisms in Behavior*, pages 1–41. Wiley, New York, 1951.

[62] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1), 1990.

[63] H. Westphal, P. Robert, J. Chassin de Kergommeaux, and J. Syre. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Proceedings of the Symposium on Logic Programming*, pages 436–448, Los Alamitos, CA, 1987. IEEE Computer Society.

[64] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. In *Conf. on Object Oriented Programming Systems, Languages, and Applications.* ACM Press, 1997.