# AN EFFICIENT PARALLEL POINTER MACHINE ALGORITHM FOR THE NCA PROBLEM

A. Dal Palú, E. Pontelli, D. Ranjan
*Department of Computer Science*
*New Mexico State University*
*Las Cruces, NM 88003*
{apalu,epontell,dranjan}@cs.nmsu.edu

## 1. Introduction

The Nearest Common Ancestor (NCA) Problem can be broadly defined as follows: Given a rooted tree $T$ and two nodes $x, y \in T$, find the common ancestor of $x$ and $y$ in $T$ that is furthest from the root. 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. NCA problem has been studied extensively [16, 21, 15, 3, 25, 1, 6, 5, 8, 4] .

We present an efficient parallel pointer machine algorithm for the NCA problem for trees in the static case. The algorithm assumes that the tree $T$ is known in advance. It requires $O(\lg n)$ 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 (TP) problem [20]. We provide an efficient parallel pointer machine algorithm to solve this problem as well.

The paper is organized as follows. In the next section, we give a brief description of the pointer machine and the parallel pointer machine models. In Section 3, we present an arithmetic-free compression scheme that was first introduced and used in [8] to obtain an optimal sequential solution for the NCA Problem on Pure Pointer Machines. We then discuss why straightforward parallelizations of this scheme fail (Section 4). In Section 5 we present our parallel algorithm. We show that our algorithm works correctly and that it requires $O(\lg n)$ parallel time. In Section 6, we compare and constrast our algorithm with other parallel NCA algorithms. Our NCA algorithm requires an efficient parallel solution of the TP problem [20]. An efficient parallel pointer machine algorithm for this problem is presented in Section 7.

R. Baeza-Yates et al. (eds.), *Foundations of Information Technology in the Era of Network and Mobile Computing* © Springer Science+Business Media New York 2002

## 2. Pointer Machines

Pointer Machines have been defined in various different ways [2]. All models of pointer machines share the common characteristic of disallowing indexing into an array (i.e., pointer arithmetic), as opposed to RAM models. The *Pure Pointer Machine (PPM)* model also disallows constant-time arithmetic operations. The PPM model is essentially the Linking Automaton model proposed by Knuth [18]. Further details on PPMs can be found in [2, 18, 23, 22].

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 (see [7] for an extensive discussion). All models rely on the presence of a number of processors; each processor is essentially a sequential pointer machine, and all processors execute the same program in a synchronous fashion. At one end of the spectrum we have the *CRCW Parallel Pointer Machine* [14], 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 [7]. 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: a finite state machine may redirect one of its links to point to another unit at a "pointer distance" no more than two from it. It has been shown that Parallel PPMs are surprisingly powerful. The details of what exactly constitutes a parallel PPM can be found in [7].

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 [13]. 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 [10].

## 3. A Sequential Compression Scheme for Trees

The starting point of our parallel algorithm is an optimal sequential method for solving the NCA problem on PPMs that was first presented in [8]. This method is based on a *compression scheme* aimed at creating a new tree with logarithmic depth that preserves the ancestor structure of the original tree. It 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 of $T$. This allows a fast solution of nca queries.

Given $T_i$, $T^L_{i+1}$ the result of *leaf-compression* of $T_i$: it is obtained by merging each leaf of $T_i$ with its parent. If a leaf $\ell$ is merged with its parent $parent(\ell)$, then $parent(\ell)$ is said to be the *direct representative* of $\ell$. A *path-compression* of a tree $T^L_{i+1}$ returns a tree $T_{i+1}$, where each path containing only nodes with a single child and ending in a leaf of $T^L_{i+1}$ 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 $T^L_{i+1}$, and $T^L_{i+1}$ is the result of a leaf-compression on $T_i$. Fig. 1 shows an example of repeated compression of $T$.

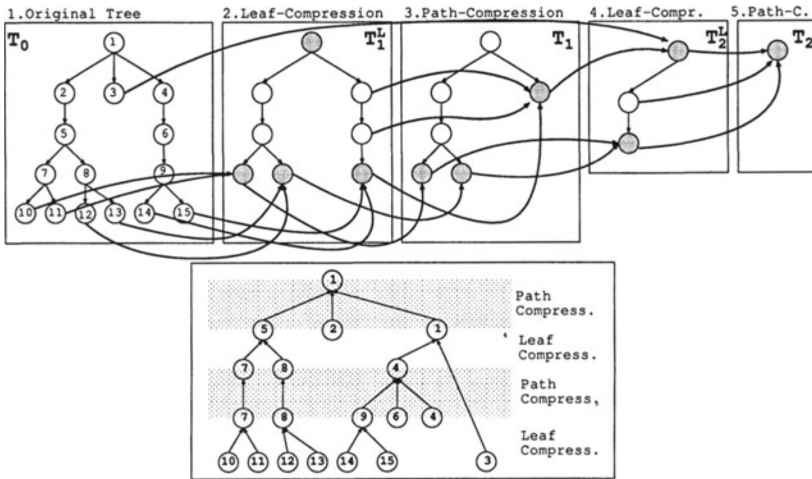


*Figure 1.* Building the $H$-tree

## 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 Fig. 1).

If a leaf-compression is applied to node $v$ in tree $T_i$ and $\ell$ is the direct representative of $v$ in such compression, then node $v$ is connected to the last occurrence of $\ell$ in a tree $T_j$ $(i < j)$, where $\ell$ appears in $T_j$ as a direct representative of a leaf-compression. If all the children of a node $w$ in $T_i$ are leaf-compressed at the same time, then the representative of such children is node $w$ in $T^L_{i+1}$ (as for leaves $10, 11$ in Fig. 1). If the children of $w$ are leaf-compressed at different

points in time (e.g., the children of 1 in Fig. 1), then the representative of such leaf is the last occurrence of its direct representative in a tree as representative in a leaf-compression. If a path-compression is applied, then all nodes in the path are connected to the head of the list in the next tree, as shown in Fig. 1. 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 Fig. 1) as the root and using the links between nodes and representatives as edges (e.g., the dark edges in Fig. 1). In [8] the following result was established:

**Theorem 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.*

The $H$-tree preserves enough *nca* information from $T$, so that it is possible to answer the query $nca(x, y)$ in $T$ by answering a suitable NCA query in $H$. In fact, since the height of $H$ is $O(\lg n)$, it is possible to compute the *nca* of any two nodes in $T$ with worst-case time complexity $O(\lg \lg n)$[8].

## 4. From Sequential to Parallel

The 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 Fig. 2. 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 the next path compression to 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 $k^2 = \Theta(\lg^2 n)$.

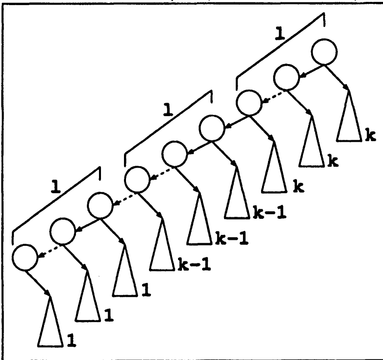


*Figure 2.* An example of slow computation

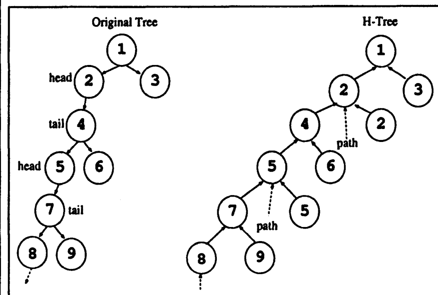


*Figure 3.* An example of bad H-tree

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 Fig. 3, 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.

# 5. A Parallel Compression Scheme for Trees

The compression algorithm is a sequential iteration of parallel phases. Each parallel phase is composed of two parallel steps. The first step is compression of leaves (*leaf compression*) in the current tree and the second step contributes to the compression of paths (*path compression*) in the current tree using a step of pointer doubling [11].

Additionally, our efficient parallel solution for the NCA problem requires the ability to efficiently solve the TP problem [20] in parallel. A parallel version of the problem and an efficient parallel solution to it is presented in Sect. 7.

## 5.1. The Algorithm

We start by introducing some notation. For a node $v$ in $T$, $T_v$ denotes the subtree of $T$ rooted at node $v$. A parallel *phase* $i$ of the algorithm is the sequence of two parallel *steps* called $a$ and $b$, which are executed at parallel time $i(a)$ and $i(b)$ respectively. For an integer $i$, $T_i$ denotes the tree after the $i^{th}$ parallel phase. Given a tree $T_i$, the result of step $a$ applied to $T_i$ is the tree $T^a_{i+1}$ and the result of step $b$ applied to $T^a_i$ 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 $m^a_i(v) = L$ $(m_i(v) = L)$. 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 $\pi$ to an ancestor of $v$ at parallel time $i(a)$ $(i(b))$ and we denote it with $\pi^a_i(v)$ $(\pi_i(v))$.

A *leaf compression* of a tree $T_i$ is executed in step $i(a)$ and returns a tree $T^a_{i+1}$ such that for each node $v$ in $T_i$ (see Fig. 4):

($i$) if ($m_i(v) = L$ and $v$ currently has no sibling) then
$\pi^a_{i+1}(v) \leftarrow \pi_i(p(v))$ and $m^a_{i+1}(\pi^a_{i+1}(v)) \leftarrow L$;

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

A *path compression* of a tree $T^a_i$ is executed in step $i(b)$ 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 Fig. 5).

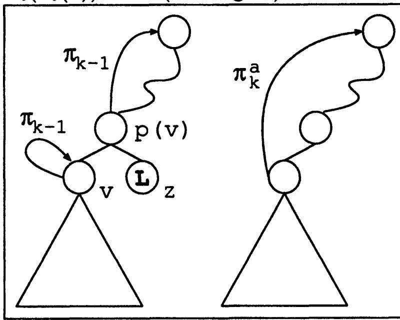


*Figure 4.* Example of Leaf compression for node $x$

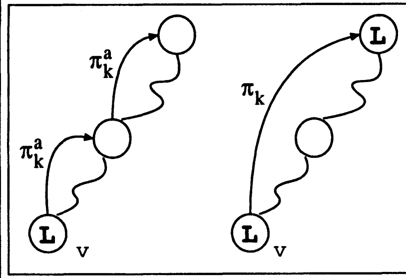


*Figure 5.* Example of Path compression for node $x$

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: $\pi_0(root) = root$.

Fig. 6 provides an example of a compression. The nodes marked represent the nodes labeled $L$ and the dashed pointers are the $\pi$ pointers. The pointers $\pi$ pointing to $NULL$ are not shown.

**Definition 2** *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. *$\pi_k(x) = NULL$.*

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

**Theorem 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 $|T_{p(x)}| \geq 2^k + 1$;*

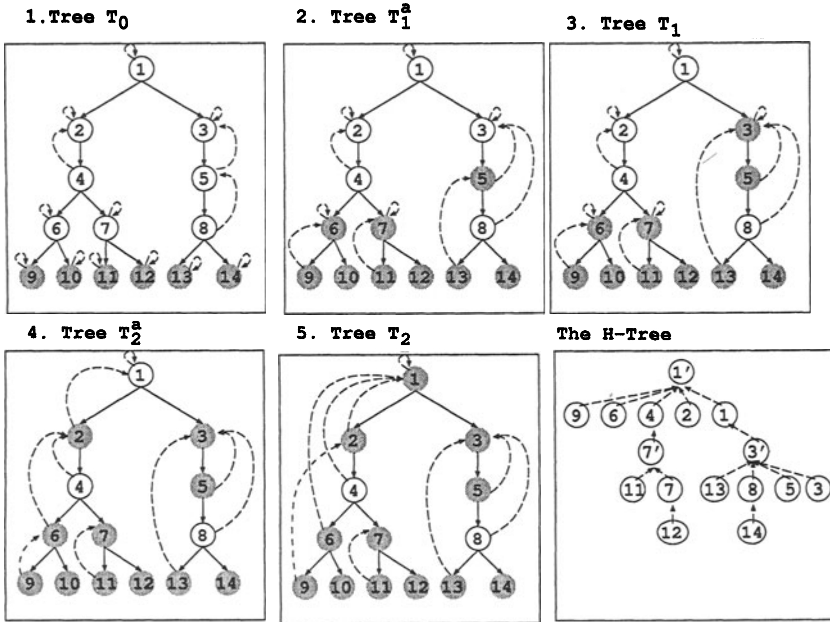


*Figure 6.* An example of parallel compression

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 $|T_x| \geq 2^k + 1)$.*

The complete proof can be found in [9].

**Corollary 1** *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.*

If we have $n$ processors that have been assigned to the $n$ different nodes of $T$, then both leaf and path compressions will be performed in constant parallel time. From Corollary 1, the total number of compressions is at most $\lg n$, thus the total parallel time required by the algorithm is $O(\lg n)$.

## 5.2. The $H$-tree

The $H$-Tree built in the parallel scheme is the same described in Section 3. It is possible to reuse the $\pi$ pointers to build $H$ in constant parallel time. Once a node $v$ is leaf compressed into its parent, $\pi(v)$ is set to $NULL$. At that time for every node $w$ in the path having $v$ as head, we have $\pi(w) = v$. The goal is to maintain this information in the successive phases, avoiding avoiding pointer doubling if a node is finished (line 26 in Fig. A.1). Once the compression is completed, every head of a path $x$ has $\pi(x) = NULL$ and the $\pi$ pointer for every other node $y$ points to the head of the path containing $y$.

Let us introduce another pointer $p_H$, that will be used as the parent pointer in $H$. During a leaf compression the pointer $p_H(v)$ is set to point to $p(v)$. 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 $H$, with $\pi(x') = x'$, $p_H(x') = p_H(x)$ and $\pi(x) = x'$. After each step of pointer doubling (applied to the $\pi$ pointers), every node in a path points to a newly created copy of the head. For every node $x$ in a path $p_H(x) = \pi(x)$, and this completes the building of $H$.

Finally, for each path the auxiliary data structure for the $\mathcal{TP}$ Problem is set up. It is possible to identify the tails of path in $O(1)$ time as a node $v$ is a tail iff for all children $w$ of $v$ $\pi(v) \neq \pi(w)$. Given a tail $t$, the corresponding list is processed as described in Section 7. The complete algorithm is presented in Appendix A. The lines marked with * are the ones necessary to set up the $H$ tree.

The $H$-tree can be used to answer nca queries in the same way as in the sequential case. In [19] 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. The sequential scheme presented in [19] can be 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 in $H$, and hence in $T$, can be computed in time $O(\lg \lg n)$ using a single processor.

## 6. Discussion

The algorithm described above clearly can be directly implemented on a CRCW Parallel PPM. A problem arises if we were not allowed concurrent writes, because too many processors may attempt to update the $L$ mark of the same node in the tree at the same time (e.g., line 10 in Fig. A.1). 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 Sect. 2) 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.

The algorithm requires $n$ processors to perform the $O(\lg n)$ parallel time preprocessing. 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 require $O(n/\lg n)$ processors and work in $O(\lg n)$ parallel time for preprocess-

ing. 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 do not incur *any* penalty in parallel time for preprocessing.

It is also important to observe that if we have any CROW PRAM NCA algorithm which solves the problem in parallel time $O(\lg n)$ with $f(n)$ processors and answers a query in $O(1)$ time, then for a generic translation of this algorithm to a Parallel PPM algorithm (as illustrated in [10]) 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 constaint-time addition is needed), then we can compute the centroid path and the $H$-tree based on it in $O(\lg n)$ parallel time. This is obtained by keeping a count of the number of nodes in the subtree rooted in each node during the algorithm execution. Each time we have a leaf compression phase where both children of a node are marked $L$, instead of leaf compressing the right child, we leaf compress always the child with a smaller count. It is easy to show that this will build the centroid path tree.

Note that if we are allowed only one processor to answer an nca query, then the time required must be at least $\Omega(\lg\lg n)$ [20]. 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 $O(1)$ [10]. 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.

## 7. A Parallel Algorithm For The TP Problem

The TP Problem, first defined in [20], can be reformulated in the context of parallel computations as follows: given a list $L$ with $l$ nodes representing an ordered sequence of objects, we want to answer the query 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 a left to right ordering in each level, then the precedes$(x, y)$ query can be answered comparing the children of $nca(x, y)$ in $BT$. We maintain this order in each level of $BT$ using *sibl* pointers. $BT$ is constructed via a parallel level-by-level construction. During the construction, each node of $BT$ has one processor associated to it. The root

of $BT$ is created in the first step. Then, in parallel (for $\lg l$ steps), each new processor $p$ associated to a node $v$ in $BT$ executes the following operations: create two new nodes ($v_l$ and $v_r$) with new processors associated to them, set *sibl* pointer of $v_l$ to $v_r$ and set $sibl(v_r)$ to left child of $sibl(v)$.

The last level of $BT$ contains a list of nodes $S$. Since $L$ is the input, from the way inputs are presented in the Parallel PPM model, we can assume that active processors can be assigned to each element of $L$ in time $O(\lg l)$. We can also assume that these processors have pointers that points to the *previous* element in the list (see Fig. 7). 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 *previous* pointers of $L$. Each node of $L$ contains a pointer *map* that is used to point to the corresponding node in $S$. Initially none of the *map* pointers is set. In the first step processor assigned to the head of $L$ sets its *map* pointer to the head of $S$. At the same time, the second element of $L$ sets its *map* pointer to the second element of $S$ (see Fig. 8). 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 in $O(\lg l)$ parallel time the auxiliary data structures (called *p-lists*) using a straightforward parallelization of the algorithm presented in [19]. A precedes$(x, y)$ query is then answered by a single processor in $O(\lg \lg l)$ time using the algorithm presented in [19].

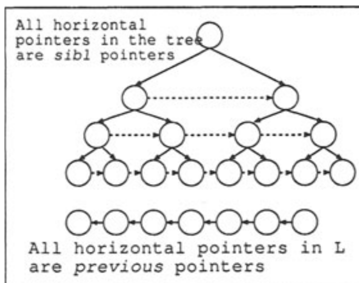


*Figure 7.* $BT$ tree and list $L$

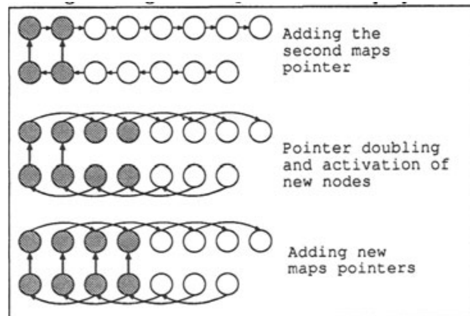


*Figure 8.* Example of mapping

## Acknowledgments

The research was supported by NSF grants CCR-9900320, EIA-0130887, CCR-9875279, CCR-9820852, and EIA-9810732.

# Appendix: Appendix A: Compression Algorithm

```
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 & 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 & v has a sibling z & m_i(z) =?)
12:           π^a_{i+1}(v) = NULL;
13:*          p_H(v) = p(v);
14:       elseif (m_i(v) =? & ((v has a sibling z & 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 & v has a left sibling z & m_i(z) = L)
17:           π^a_{i+1}(v) = NULL;
18:*          p_H(v) = p(v);
19:       elseif (m_i(v) = L & v has a right sibling z & 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' & 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 A.1.* Preprocessing Algorithm

# References

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

[2] A.M. Ben-Amram. What is a Pointer Machine? In *SIGACT News*, 26(2), 1995.

[3] M.A. Bender and M. Farach-Colton. The LCA Problem Revisited. In *Proceedings of LATIN 2000*, Springer Verlag, 2000.

[4] O. Berkman & U. Vishkin. Recursive *-Tree Parallel Data Structure. *FOCS*, 1989.

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

[6] 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.

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

[8] A. Dal Palú, E. Pontelli, D. Ranjan. An Optimal Algorithm for Finding NCA on Pure Pointer Machines. Tech. Rep., TR-CS-007/2001, NMSU, 2001.

[9] A. Dal Palú, E. Pontelli, D. Ranjan. An Efficient Parallel Pointer Machine Algorithm for Nearest-Common Ancestor Problem. Tech. Rep., TR-CS-009/2001, NMSU, 2001.

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

[11] S. Fortune and J. Wyllie. Parallelism in RAMs. In *STOC*, ACM Press, 1978.

[12] 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.

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

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

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

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

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

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

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

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

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

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

[23] 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.

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

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