Parallel Dictionaries with Local Rules on AVL and Brother Trees [2]

Joaquim Gabarró, Xavier Messeguer [1]

Universitat Politècnica de Catalunya, C/ Jordi Girona 1–3, Mòduls C5-C6, E-Barcelona 08034, Spain.

Abstract

We present a set of local rules to deal with dictionaries. Their main advantage is that they can be scheduled in a highly synchronized way to get parallel dictionaries on AVL trees. Up to now trees used in massively parallel dictionaries needed to have all the leaves at the same depth, such as 2–3 trees. Therefore, it was possible (in insertions and deletions) the bottom-up reconstruction of the tree in a very regular fashion, as a pipeline of plane waves moving up. On AVL trees the situation looks different because leaves can have different depth, therefore any wave in a pipeline is highly irregular. To solve this problem we define *virtual* plane waves allowing us to develop an EREW dictionary for k keys with k processors and time $O(\log n + \log k)$. Later on we generalize the sequential algorithms on brother trees presented by Ottmann and Wood in the same way.

Key words: Parallel algorithms, Parallel dictionaries, AVL trees, Brother trees.

1 Introduction

AVL trees are an important basic data structure [1,5]. An open problem is the design of parallel dictionaries based on them. A similar problem was solved for 2–3 trees in 1983 by Paul, Vishkin and Wagener [8]. They design parallel algorithms to maintain dynamically a parallel dictionary working on "batches" of k keys simultaneously. Let us sketch their insertion algorithm. In 2-3 trees

[1] Partially supported by the ESPRIT Long Term Research Project no. 20244 (ALCOM IT) and DGICYT under grant PB95-0787 (project KOALA) and CICIT TIC97-1475-CE and CIRIT 1997SGR-00366.

[2] Expanded version of a talk presented at the 9th International Conference on Parallel and Distributed Computing Systems, ISCA, 1996.

all the internal nodes have 2 or 3 sons and all the leaves have the same depth. We assume that any leaf has two *nil* nodes as left and right sons. As usual, the *height* of a node n has a bottom-up definition. We take $\mathbf{height}(nil) = 0$, therefore all the leaves have height 1 and so on. The *depth* can be defined top-down starting with $\mathbf{depth}(root) = 0$. The 2-3 trees hold the following invariant:

$$\mathbf{height}(root) = \mathbf{depth}(nil) = \mathbf{depth}(n) + \mathbf{height}(n).$$

Based on it, Paul, Vishkin and Wagener designed pipelines of processes to deal with parallel insertions and deletions. These pipelines can be described intuitively in terms of traveling plane waves. Recall, for instance, the basic insertion case in which every leaf incorporates at most one new key. Then, something like a *straight wave* (or *plane wave*) is generated at the bottom of the tree and is sent up in further steps. The wave goes to the root step by step increasing the height and decreasing the depth; it looks like a straight line going up in a picture of the tree. Moreover, the evolution of the wave needs only a local computation and every pair of nodes p and q belonging to the wave verify:

$$\mathbf{height}(p) = \mathbf{height}(q) \qquad \text{and} \qquad \mathbf{depth}(p) = \mathbf{depth}(q).$$

In the general insertion case, a pipeline of traveling waves is generated. These ideas were applied on B trees by Higham and Schenk [3] and on Skip lists [2] by Gabarró, Martínez and Messeguer.

In AVL trees, every node n stores a key $\mathbf{key}(n)$ and three pointers $\mathbf{left}(n)$, $\mathbf{right}(n)$ and $\mathbf{parent}(n)$ point to the sons and parent. We recall the definitions of the *height, depth, balance* and *lower* of a node. The height is defined as $\mathbf{height}(nil) = 0$ and

$$\mathbf{height}(n) = 1 + \max(\mathbf{height}(\mathbf{left}(n)), \mathbf{height}(\mathbf{right}(n))).$$

The *depth* is $\mathbf{depth}(root) = 0$ and $\mathbf{depth}(n) = 1 + \mathbf{depth}(\mathbf{parent}(n))$. The *balance* is: $\mathbf{bal}(n) = \mathbf{height}(\mathbf{right}(n)) - \mathbf{height}(\mathbf{left}(n))$. The AVL trees are defined such that $\mathbf{bal}(n) \in \{-1, 0, +1\}$. Node n is *lower* if

$$\mathbf{lower}(n) \equiv (\mathbf{height}(n) < \mathbf{height}(\mathbf{brother}(n)).$$

We define the *virtual depth* such that $\mathbf{virtual\text{-}depth}(root) = 0$ and

$$\mathbf{virtual\text{-}depth}(n) = 1 + \ \mathbf{lower}(n) + \mathbf{virtual\text{-}depth}(\mathbf{parent}(n)).$$

This new depth give us the invariant:

$$\mathbf{height}(root) = \mathbf{virtual\text{-}depth}(nil) = \mathbf{virtual\text{-}depth}(n) + \mathbf{height}(n).$$

which resembles the invariant of 2–3 trees. Therefore, in AVL trees it is also possible the design of *virtually plane waves* such that every pair of nodes p and q into the wave verify:

$$\textbf{height}(p) = \textbf{height}(q) \qquad \text{and} \qquad \textbf{virtual-depth}(p) = \textbf{virtual-depth}(q).$$

Then, if we assume that information flow does not modifies the tree, we can design a pipeline of waves flowiing upward:

PIPELINE SCHEME FOR STATIC AVL TREES: using *virtual depth*, the leaves become aligned at the bottom of the AVL trees. It is possible to start and move up a *virtual plane wave*. When a wave moves up the *height* increases and the *virtual depth* decreases. We can organize virtual plane waves to get a pipeline in a static AVL tree.

This pipeline can also be applied to brother trees, introduced by Ottmann, Six and Wood [6]. In brother trees, all the leaves have the same depth, internal nodes can have one or two sons, but each node with only one son has a brother with two sons. In our version, information is stored in the leaves and internal nodes act as routers.

The rest of the paper is organized as follows. In sections 2 and 3 we introduce the parallel insertion and deletion algorithms on AVL trees. Finally in section 4 we sketch the algorithms for brother trees.

2 Parallel insertions on AVL trees

Following Kessels [4] we force newly inserted nodes not to count in computing the height. To achieve that we introduce a new register $\textbf{color}(n)$.

2.1 Red relaxed AVL trees

We color white the old nodes and red the new ones. Also we color red the *nil* nodes. We define the predicates

$$\textbf{red}(n) \equiv (\textbf{color}(n) = \textbf{red}) \qquad \text{and} \qquad \textbf{white}(n) \equiv (\textbf{color}(n) = \textbf{white}).$$

Red nodes mean two things, first they do not count to compute the height and second, they represent an unstable perturbation to be propagated up or erased as soon as possible. We recall from [4] the *dynamic height*. If n is a red leaf $\textbf{dheight}(n) = 0$, if n is a white leaf $\textbf{dheight}(n) = 1$, otherwise:

$$\textbf{dheight}(n) = \textbf{white}(n) + \max(\textbf{dheight}(\textbf{left}(n)), \textbf{dheight}(\textbf{right}(n))).$$

Based on this height we can introduce the *dynamic balance* $\mathbf{dbal}(n)$ and the predicate *dynamic lower*, such that

$$\mathbf{dlower}(n) \equiv (\mathbf{dheight}(n) < \mathbf{dheight}(\mathbf{brother}(n)).$$

A tree is a *red relaxed AVL tree* if any node n verifies $\mathbf{dbal}(n) \in \{-1, 0, 1\}$. Note that when all nodes are white, a red relaxed AVL tree is an AVL tree. As in static AVL trees, we define the *virtual dynamic depth* in the following way: $\mathbf{virtual\text{-}ddepth}(root) = 0$ and

$$\mathbf{virtual\text{-}ddepth}(n) = \mathbf{white}(n) + \mathbf{dlower}(n) + \mathbf{virtual\text{-}ddepth}(\mathbf{parent}(n)).$$

The following lemma gives us a precise statement of the preceding intuitions. It can be proved by using

$$\mathbf{dheight}(n) = \mathbf{dheight}(\mathbf{parent}(n)) - \mathbf{white}(\mathbf{parent}(n)) - \mathbf{dlower}(n).$$

Lemma 1 *In a* red relaxed AVL tree *any node n verifies*

$$\mathbf{dheight}(root) + \mathbf{red}(root) = \mathbf{dheight}(n) + \mathbf{virtual\text{-}ddepth}(n) + \mathbf{red}(n).$$

All the nil nodes have the same virtual dynamic depth *because*

$$\mathbf{dheight}(root) + \mathbf{red}(root) = 1 + \mathbf{virtual\text{-}ddepth}(nil).$$

Based on the preceding lemma we come to our second design scheme. It will allow us to deal with waves perturbing the data structure dynamically.

RED DYNAMIC PIPELINE SCHEME: Using *virtual dynamic depth*, the leaves become aligned at the bottom of a red relaxed AVL tree. Thus it is possible to start and move up a *virtual dynamic plane wave*. When this wave moves up the *dynamic height* increases and the *virtual dynamic depth* decreases. We can organize waves to get a pipeline in a red relaxed AVL tree.

2.2 Local rules

They are an extension of those given by Kessels [4]. Essentially, the rules move red nodes up (maintaining the relaxed balance condition) until they come to the root and disappear. We write as $n(A, B)$ the (sub)tree with root n, left son A and right son B. The final state of n is denoted as n'. We accept balanced red nodes with positive dynamic height. Therefore, the invariant $\mathbf{dheight}(n) > 0 \wedge \mathbf{red}(n) \implies \mathbf{dbal}(n) \neq 0$ given in [4] does not hold.

We introduce the colored propagations and rotations. We consider two cases of redness propagation depending on the color of the sons.

Rule : Red Propagation

Guard: *A subtree* $n(p(A,B),C)$ *such that* n *is white,* p *is red, the brother of* p *is white and* $\mathbf{dbal}(n) \in \{0,1\}$.

Behavior: *The new tree is* $n'(p'(A,B),C)$. *If* $\mathbf{dbal}(n) = 0$, *we have* n' *red,* p' *white,* $\mathbf{dbal}(n') = -1$. *Otherwise* $\mathbf{dbal}(n) = 1$, *nodes* n' *and* p' *become white and* $\mathbf{dbal}(n') = 0$. *In both cases* $\mathbf{dbal}(p') = \mathbf{dbal}(p)$.

Spatial scope: *Node* p *and its father* n.

Note: *Symmetrically for the right son of* n.

Rule : Parallel Red Propagation

Guard: *A subtree* $n(p(A,B),q(C,D))$ *such that* n *is white and its sons* p *and* q *are red. There are no conditions on the dynamic balances.*

Behavior: *The dynamic balance of the nodes remain unchanged,* n' *becomes red,* p' *and* q' *become white.*

Spatial scope: *Nodes* n, p *and* q.

There are also the so called *colored rotations*, coupling the propagation of a red color with a usual rotation in order to keep the relaxed nodes balanced. We consider single and double rotations in separate rules.

Rule : Single Red Rotation

Guard: *A subtree* $n(p(A,B),C)$ *such that* n *and the root of* C *are white,* p *is red,* $\mathbf{dbal}(n) = -1$ *and* $\mathbf{dbal}(p) = \{0,-1\}$.

Behavior: *Restructure the tree into* $p'(A,n'(B,C))$ *with usual updating for keys and left and right pointers.*
If $\mathbf{dbal}(p) = -1$ *nodes* p' *and* n' *are white, their dynamic balance is* 0.
If $\mathbf{dbal}(p) = 0$ *then* p' *is red* n' *is white* $\mathbf{dbal}(p') = +1$ *and* $\mathbf{dbal}(n') = -1$.

Spatial scope: *Node* n, *its left son* p *and the roots of* B, C.

Note: *Symmetrically for the right-right case:* $n(A,q(B,C))$, $\mathbf{dbal}(n) = +1$ *and* $\mathbf{dbal}(q) = \{0,+1\}$ *with* n *white and* q *red.*

Rule : Double Red Rotation

Guard: *A subtree* $n(p(A,q(B,C),D)$ *such that* n *and the root of* D *are white,* p *is red,* $\mathbf{dbal}(n) = -1$ *and* $\mathbf{dbal}(p) = +1$.

Behavior: *Restructure the tree into* $q'(p'(A,B),n'(C,D))$ *with usual updating for keys and left and right pointers.*
If q *is white, nodes* n', p', q' *are white and* q' *is balanced. The balance of other nodes is updated as needed.*

Spatial scope: *Nodes* n, p, q *and the root of* B, C *and* D.

Note: *We have the symmetrical case.*

Finally, we need a rule to deal with a red root. The following rule allows the root of the whole tree to change from red to white (as we will see later, this

rule increases by one the dynamic height of the tree)

Rule : Root's Whitening
Guard: *The root of the whole tree is red.*
Behavior: *The root becomes white.*
Spatial scope: *The root.*

The rules have been designed to obtain the following lemmas. If all the rules were given explicitly, they could be proved by (lengthly) straightforward case analysis.

Lemma 2 *Any application of a rule keeps the tree dynamically balanced (we take the notations given in the rules).*

(1) A propagation rule verifies $\mathbf{dheight}(n) = \mathbf{dheight}(n')$.
(2) A colored rotation maintains the dynamic height of the subtree, for instance in single right rotation $\mathbf{dheight}(p') = \mathbf{dheight}(n)$.
(3) When the root goes from red to white $\mathbf{dheight}(root') = 1 + \mathbf{dheight}(root)$.

These rules are applied to white nodes having at least one red son. Then a node n is called **active** if it is white but has at least a red son. Then we "give the control" to this kind of nodes.

Corollary 1 *Let n be an active node in a red relaxed AVL tree.*

(1) Any propagation rule *keeps constant the dynamic virtual depth of any node different from n and*

$$\mathbf{virtual\text{-}ddepth}(n) = \mathbf{virtual\text{-}ddepth}(n') + \mathbf{red}(n').$$

(2) Any colored rotation *can only modify the dynamic virtual depth of nodes in its scope. If r' is the new root of the rotated subtree we have:*

$$\mathbf{virtual\text{-}ddepth}(n) = \mathbf{virtual\text{-}ddepth}(r') + \mathbf{red}(r').$$

2.3 Parallel insertion algorithm

We apply the rules in a synchronized way based on the previous red dynamic pipeline scheme. First we study the behaviour of red relaxed AVL trees. Second we apply the preceding ideas to get a parallel algorithm for AVL trees.

We consider two cases dealing with red relaxed AVL trees.

(1) Assume all the keys to be inserted are attached to a white node. More formally, any red node (different from nil) with dynamic height 0 has a white father. The parallel algorithm can be sketched as follows.

- All the active nodes with dynamic height 1 apply the corresponding rule.
- All the active nodes with dynamic height 2 apply the corresponding rule.
- Iterate, increasing the dynamic height at each step.

We can see the set of active nodes as a wave going upward the tree.

Lemma 3 *The nodes that belong to a wave have the same* virtual dynamic depth.

Proof: In the preceding algorithm two active nodes p and q belonging to the front wave verify $\mathbf{dheight}(p) = \mathbf{dheight}(q)$. Since throughout the whole algorithm the tree is a red relaxed AVL tree, from lemma 1 any node n verifies

$$\mathbf{dheight}(n) + \mathbf{virtual\text{-}ddepth}(n) + \mathbf{red}(n) = \mathbf{dheight}(root) + \mathbf{red}(root).$$

As any active node n in the wave is white, this concludes the proof. □

The expression *virtual red plane wave* makes sense because the wave behaves as a plane wave using the dynamic height and the virtual dynamic depth. As all the active white nodes have the same dynamic height and dynamic virtual depth, it makes sense to assign a dynamic height and a dynamic virtual depth to the virtual plane wave w written $\mathbf{dheight}(w)$ and $\mathbf{virtual\text{-}ddepth}(w)$.

(2) Let us consider a special case of a red relaxed AVL tree consisting on an AVL tree with a "red beard at the bottom". By this we mean that all the red nodes, containing a new key to be inserted, are at the bottom of the tree and all the other nodes are white. To solve this case we would like to pipeline different virtual plane waves. To do this we need to prove that two different waves do not collide if they are initially separated from one another. But, from lemma 1 any virtual plane wave will not be affected by the behavior of other waves higher in the tree. Therefore it holds:

Theorem 1 *Take a relaxed AVL tree having red nodes at the bottom. If we start from the bottom a virtual plane wave w moving up and λ (take for instance $\lambda = 10$) steps later we start another one w' moving up at the same rate, the wave w' will remain virtually plane and moreover*

$$\lambda = \mathbf{dheight}(w) - \mathbf{dheight}(w') = \mathbf{virtual\text{-}ddepth}(w') - \mathbf{virtual\text{-}ddepth}(w)$$

while w and w' are moving up.

Finally let us link red relaxed AVL trees to AVL trees. We consider the insertion case where we have an AVL tree with n keys and we have to insert an ordered set of k new keys stored in the array $a[1..k]$. First, we employ the search algorithm by packet routing [8,3,2]. This search algorithm is formally identical to the search by packet routing on Skip lists. The results about read conflicts in Skip lists apply here [2] and, at most three subpackets can remain

on a node. When the subpackets are located at the bottom of the AVL, the divide and conquer approach given in [8] allows us to start a pipeline of waves as in (2). When all red nodes have become white, the red relaxed AVL tree is an AVL tree and the following theorem holds:

Theorem 2 *The massively parallel insertion of k keys into an AVL tree with n keys takes time $O(\log n + \log k)$ using k processors on an EREW PRAM.*

3 Parallel deletion algorithm on AVLs

Here we address the deletion of nodes. Recall that in the insertion case the newly inserted nodes do not count to compute the height. To take into account this fact the nodes are colored red. When a deletion is performed the height of a subtree can decrease, therefore in order to maintain the dynamic height, some nodes need to count twice. We color these nodes *green*. Let us sketch the version of the sequential deletion algorithm which is composed of two phases:

- **Request phase:** A *request to delete* the key k is introduced at the root and percolated down through the tree until it stops at a node n at the bottom of the tree. If $k \leq \mathbf{key}(n)$ the value k is stored in the so called **left-request**(n) register, otherwise it is stored in the **right-request**(n) register.

  The *request manager* of n (its father or grandfather) take care of this request. Depending on the cases, the node n is deleted, a green node is generated (in order to maintain the relaxed AVL character) and an *update set* is created. This update is denoted $\{\mathbf{key}(n) := k\}$ and means "key k should be substituted by $\mathbf{key}(n)$".
- **Balance and update phase:** the greenness rises up through the tree until it disappears or reaches the root. The update set comes up until it arrives at a node holding the key k and substitutes k by $\mathbf{key}(n)$.

The parallel deletion algorithm starts by percolating down any deletion request. At the end of this percolation phase any deletion request will be located at one request register. The following easy property holds,

Lemma 4 *If the set of keys to be deleted does not contain any repetition and all these keys belong to the tree, any leaf n will contain at most two deletion requests: one located at* **left-request**(n) *and the second in* **right-request**(n).

As green nodes counts twice, the *dynamic height* of a green leaf is 2, white leaf has dynamic height 1, otherwise

$$\mathbf{dheight}(n) = 1 + \mathbf{green}(n) + \max(\mathbf{dheight}(\mathbf{left}(n)), \mathbf{dheight}(\mathbf{right}(n))).$$

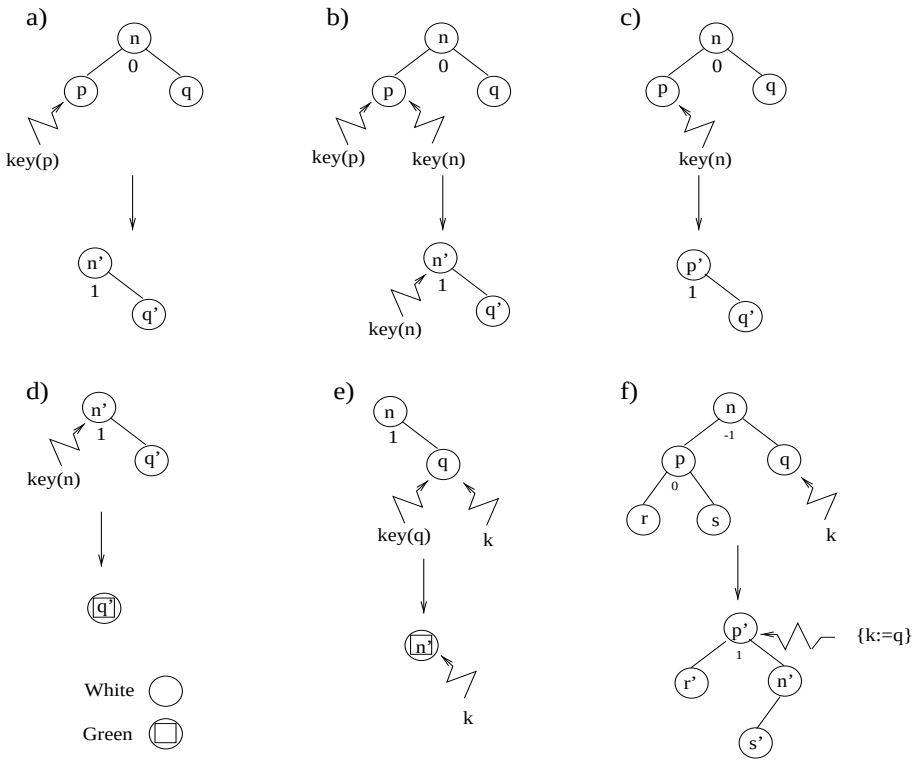

Fig. 1. Request Manager

The *virtual dynamic depth* of a node n is given by $\mathbf{virtual\text{-}ddepth}(root) = 0$ and for any other node:

$$\mathbf{virtual\text{-}ddepth}(n) = 1 + \mathbf{green}(n) + \mathbf{lower}(n) + \mathbf{virtual\text{-}ddepth}(\mathbf{parent}(n)).$$

Lemma 5 *In a* green relaxed AVL *any node* n *verifies*

$$\mathbf{dheight}(n) + \mathbf{virtual\text{-}ddepth}(n) - \mathbf{green}(n) = \mathbf{dheight}(root) - \mathbf{green}(root).$$

All the nil nodes have the same dynamic balanced depth *because*

$$\mathbf{dheight}(root) - \mathbf{green}(root) = \mathbf{virtual\text{-}ddepth}(nil).$$

As in the case of red relaxed AVLs we can introduce a pipeline scheme based on dynamic magnitudes.

GREEN DYNAMIC PIPELINE SCHEME: With the *virtual dynamic depth* we can pipeline *virtual waves* in green relaxed AVL trees.

Let us *sketch* the set of rules to deal with deletions. Recall that we need three kind of rules to take into account requests, green color and update sets. We do not describe the rules that deal with green nodes because they can be

obtained from their red counterparts. As in the insertions case, rules keep the tree dynamically balanced.

We informally describe the rule **Request Manager** addressing the request located at (dynamic) height 2 or 3. This rule processes only one request at any application. If necessary,

- the manager colors green a node at the bottom of the tree in order to maintain the dynamic height (Figure 1, cases d and e),
- it generates a new update set $\{k := q\}$ (Figure 1, case f),
- it rotates in order to maintain the red relaxed AVL character (Figure 1, case f).

To reduce the number of cases, if both sons have requests we always give priority to the left son over the right one. Cases from a to e of Figure 1 explain in detail the behaviour of a request manager of height 2 with p and q leaves. A request manager of dynamic height 3 (Figure 1, case f) is considered if leaves r and s have no request.

The **Update** rule forces the update sets to move up. It does not change the "topology" of the AVL tree nor the color of nodes.

Rule : Update

Guard: *A subtree $n(p(A,B),q(C,D))$ with, at least, one son with an update set.*

Behavior: *Changes to $n'(p'(A,B),q'(C,D))$. If p has the update set $\{n := k\}$, therefore* $\mathbf{key}(n') = k$. *If q has an update set then it is forwarded up and located at n'.*

Spatial scope: *Nodes n, p, q.*

Note: *All other cases are quite similar and easy.*

The parallel deletion algorithm is a pipeline of two types of waves. When rule **Request manager** ends, two types of waves have been created: an *Update-wave* dealing with the update sets and a *green-wave* moving up the green nodes. Few steps later the rule **Request manager** acts again and again until all requests have been satisfied.

As before, we get the following theorem for AVL trees:

Theorem 3 *The massively parallel deletion of k keys in an AVL tree with n keys takes time $O(\log n + \log k)$ using k processors on an EREW PRAM.*

4 Relaxed Brother Trees

To transform an AVL tree to a brother tree we need to add a *white node*, denoted $\circ$, with no information between n and $\mathbf{parent}(n)$ if $\mathbf{lower}(n)$ holds. Only two forms are allowed to ternary nodes, $n(A, B, C)$ or $n(A, \circ(B), C)$ (with $\circ$ in the middle). In any case all the leaves have the same depth. We can extend the set of rules given by Ottmann and Wood [7] to deal with the parallel cases. As an example, consider the following rule:

Rule : Brother Rotation

Guard: *A subtree* $n(p(A\circ(B), C), \circ(D))$. *Assume* $\mathbf{key}(n) = k$ *and* $\mathbf{keys}(p) = \{u, v\}$.

Behavior: *Change to* $n'(p'(A, \circ(B)), q'(C, D))$ *with* $\mathbf{key}(n') = v$, $\mathbf{key}(p') = u$, $\mathbf{key}(q') = k$

Spatial scope: *Node* n, *the roots of* A *and* C, *and the node* $\circ$.

Following the previous ideas we can design a massively parallel EREW dictionary based on brother trees in time $O(\log n + \log k)$ and k processors.

References

[1] G.M. Adel'son-Vel'skiĭ and Y.M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklay*, (3):1259–1263, 1962.

[2] J. Gabarró, C. Martínez, and X. Messeguer. A design of a parallel dictionary using skip lists. *Theoretical Computer Science*, 158:1–33, 1996.

[3] L. Higham and E. Schenks. Maintaining B-trees on an EREW PRAM. *J. of Parallel and Dist. Comp.*, 22:329–335, 1994.

[4] J.L.W. Kessels. On-the-fly optimization of data structures. *Comm. ACM*, 26(11):895–901, 1983.

[5] D.E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Mass., 1973.

[6] T. Ottmann, H. Six, and D. Wood. On the correspondence between AVL trees and brother trees. *Computing*, 23(1):43–54, 1979.

[7] T. Ottmann and D. Wood. 1-2 brother trees or AVL trees revisited. *The Computer Journal*, 23(3):248–255, 1979.

[8] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In J. Díaz, editor, *Proc. 10th ICALP , LNCS 154*, pages 597–609. Springer-Verlag, 1983.