Massively Parallel and Distributed Dictionaries on AVL and Brother Trees

J. Gabarró*
Llenguatges i Sistemes Informàtics
Univ. Politècnica de Catalunya
E-Barcelona 08028
Spain

X. Messeguer*
Llenguatges i Sistemes Informàtics
Univ. Politècnica de Catalunya
E-Barcelona 08028
Spain

Abstract

We present a set of local rules to deal with distributed dictionaries, having as a main advantage their possible scheduling in a highly synchronized way to get massively parallel dictionaries on AVLs. Recall that, 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) to reconstruct the tree bottom-up in a very regular fashion, as a pipeline of *straight plane weaves* moving up. On AVL trees the situation looks different because leaves can have different depth, therefore any weave 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 T. Ottmann and D. Wood in the same way.

1 AVL Trees

They are a basic data structure [1, 5]. However it left open how to design massively parallel dictionaries on them. This problem was first solved for other classes of trees (2–3 trees) in 1983 by W. Paul, U. Vishkin and H. Wagener [8]. These ideas were applied on B trees by L. Higham and E. Schenk [3] in 1994 and on Skip lists by J. Gabarró, C. Martínez and X. Messeguer [2] in 1995.

In AVL trees every node n stores a key $\mathbf{key}(n)$ and $\mathbf{left}(n)$, $\mathbf{right}(n)$, $\mathbf{father}(n)$ points to their sons and father. The height is $\mathbf{height}(nil) = 0$ and $\mathbf{height}(n)$ is $1 + \max(\mathbf{height}(\mathbf{left}(n)), \mathbf{height}(\mathbf{right}(n)))$. The *depth* is $\mathbf{depth}(root) = 0$ and $\mathbf{depth}(n) = 1 + \mathbf{depth}(\mathbf{father}(n))$. The *balance* is:

$$\mathbf{bal}(n) = \mathbf{height}(\mathbf{right}(n)) - \mathbf{height}(\mathbf{left}(n)).$$

In AVLs $\mathbf{bal}(n) \in \{-1, 0, +1\}$. Node n is *lower* if $\mathbf{lower}(n) \equiv (\mathbf{height}(n) < \mathbf{height}(\mathbf{brother}(n))$. Finally, let us define a new depth so called *virtual depth* such that $\mathbf{virtual\text{-}depth}(root) = 0$ and

$$\begin{aligned} \mathbf{virtual\text{-}depth}(n) &= 1 + \mathbf{lower}(n) \\ &\quad + \mathbf{virtual\text{-}depth}(\mathbf{father}(n)). \end{aligned}$$

This new depth give us the invariant:

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

Therefore it is possible design *virtually plane waves* such that every pair of nodes p and q into the wave front verify: $\mathbf{height}(p) = \mathbf{height}(q)$ and $\mathbf{virtual\text{-}depth}(p) = \mathbf{virtual\text{-}depth}(q)$. Based on this (if we assume that information's flow does not modifies the tree) we can design a pipeline flowing bottom-up

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

2 Massively parallel and distributed insertions on AVLs

Following J.L.W. Kessels [4] we assume that a newly inserted node does not count to compute the height. A way to do it consists to color it. This color is stored into a register $\mathbf{color}(n)$.

*Partially supported by the ESPRIT Long Term Research Project no. 2044 (ALCOM IT).

†Contact author: messeguer@lsi.upc.es

2.1 Red relaxed AVLs

Old nodes are white and the new ones are red, $\mathbf{color}(nil) = \mathbf{red}$. We have the predicates $\mathbf{red}(n) \equiv (\mathbf{color}(n) = \mathbf{red})$ and $\mathbf{white}(n) \equiv (\mathbf{color}(n) = \mathbf{white})$. Red nodes mean two things, first they do not count to compute the height [4] 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 $\mathbf{dheight}(n) = 0$, if n is a white leaf $\mathbf{dheight}(n) = 1$, otherwise:

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

Based on this height we have 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* when any node n verifies $\mathbf{dbal}(n) \in \{-1, 0, 1\}$, look [4]. When all the nodes are white, a red relaxed AVL is an AVL. We define the *virtual dynamic depth*, $\mathbf{virtual\text{-}ddepth}(n)$, such that $\mathbf{virtual\text{-}ddepth}(root) = 0$ and

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

The following lemma gives us a precise statement of the preceding intuitions.

Lemma 1 *In a* red relaxed AVL *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. Thus it is possible to start and move up a *virtual dynamic plane wave* (also called *virtual plane waves* or *plane waves* or just *waves*). When the wave moves up the *dynamic height* increases and the *virtual dynamic depth* decreases. We can chain waves to get a pipeline in red relaxed AVL.

2.2 Local rules for insertions

When *just one* red node moves up, we get the set of rules given by J.L.W. Kessels [4]. There are two cases (propagations or rotations), we recall only the so called red rotation:

Rule 1 (Red Rotation)

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

Behavior: *Restructure into $p'(A, n'(B, C))$ with the usual updating for keys and left and right pointers. The color of n' and p' is white and its dynamic balance is 0.*

Spatial scope: *Node n and its left son p.*

Note: *Symmetrically for the right-right case: $n(A, q(B, C))$, $\mathbf{dbal}(n) = \mathbf{dbal}(q) = +1$, n is white and q is red. Colored double rotations will also appear.*

We consider now the rules issues from the parallel case, extending the J.L.W. Kessels [4] approach. We have three cases, parallel red propagation, parallel red rotation and consecutive red nodes. We give only one case:

Rule 2 (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 and:* $\mathbf{color}(n') := \mathbf{red}$, $\mathbf{color}(p') := \mathbf{white}$ *and* $\mathbf{color}(q') := \mathbf{white}$.

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

White nodes are stable while red nodes are bubbles going up. We "give the control" to the white nodes and call a node n *active* if it is white but has at least a red son. All the preceding rules have been designed to get the following lemma.

Lemma 2 *Any application of a rule, keeps the tree dynamically balanced. A propagation rule verifies*

$$\mathbf{dheight}(m') = \mathbf{dheight}(m).$$

A colored rotation maintains the dynamic height of the subtree, for instance in single right rotation

$$\mathbf{dheight}(p') = \mathbf{dheight}(n),$$

when the root changes from red to white it holds

$$\textbf{dheight}(root') = 1 + \textbf{dheight}(root).$$

Let n be an active node in a red relaxed AVL. Any propagation rule *keeps constant the dynamic balanced depth of any node different from n and*

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

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

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

2.3 Algorithms for insertions

Massive parallel algorithm. Just, take the preceding set of rules and apply them in a synchronized way along the lines given in W. Paul, U. Vishkin and H. Wagener [8]. It holds:

Theorem 1 *Take a relaxed AVL having red nodes at the bottom. The nodes that belong to the front wave have the same* virtual dynamic depth. *Moreover, 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*

$$\textbf{dheight}(w) - \textbf{dheight}(w') = \lambda = \textbf{virtual-ddepth}(w') - \textbf{virtual-ddepth}(w).$$

while w and w' are moving up. Using this we can get a massively parallel insertion of k keys into a red relaxed AVL tree with n keys takes time $O(\log n + \log k)$ using k processors on an EREW PRAM.

Distributed algorithm. In this case we take the set of rules and keep the control as non-deterministic as possible, that means that any rule can be selected and applied to the structure in any order as soon as its guard is satisfied. It holds:

Lemma 3 *The insertion distributed algorithm is deadlock free. Let T be an AVL tree with red nodes at the bottom, when no rule applies all nodes are white. Define outside(n) as the number of nodes not contained into the subtree rooted at n, NUMBER $= \sum_{red(n)} 1$ and RED $= \sum_{red(n)} outside(n)$, for any application of a rule, the variant $(NUMBER, RED)$ strictly decreases.*

3 Massively parallel and distributed deletions on AVLs

When some nodes are deleted the tree can become lower. In order to maintain the dynamic magnitudes we introduce *green nodes* which count twice. Let us sketch the version of the sequential deletion algorithm we will use to develop all the others.

- 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 $k \leq \textbf{key}(n)$ the value k is stored in the so called *left request register* otherwise it is stored in the *right request register*. A *request manager* of n (its father or grandfather) will take care of this request. Depending on the cases, the node n will be deleted, a *green node* will be generated (to maintain the AVL character), an *update set* written $\{\textbf{key}(n) := k\}$, meaning "the value $\textbf{key}(n)$ will substitute k on the tree" will be generated.
- Balance and update phase: a green node will move up the tree until it disappears or comes to the root. Update comes up until it arrives at a node holding the key k and substitutes k by $\textbf{key}(n)$.

3.1 Green relaxed AVLs

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 $\textbf{left-request}(n)$ and the second in $\textbf{right-request}(n)$. In deletions, the *dynamic height* of a node, is given by $\textbf{dheight}(n) = 2$ if n is a green leaf, $\textbf{dheight}(n) = 1$ if n is a white leaf, otherwise

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

When updates go up they can be pack together in register $\textbf{update}(n) = \{d_1 := k_1, d_2 := k_2, \ldots\}$. The *virtual dynamic depth* of a node n is given by $\textbf{virtual-ddepth}(root) = 0$ and for any other node:

$$\textbf{virtual-ddepth}(n) = 1 + \textbf{green}(n) + \textbf{lower}(n) + \textbf{virtual-ddepth}(\textbf{father}(n)).$$

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.

3.2 Local rules for deletions

There are four types of rules dealing with the following phenomena: request manager, update, green propagation and green rotation. Let us briefly consider the rule **Request Manager** dealing with request managers located at (dynamic) height 2 or 3. This rule processes only one request at any application. If necessary, the manager: (1) colors green a node at the bottom of the tree in order to maintain the dynamic height, (2) generates a new update set $\{d := k\}$ and (3) rotates in order to maintain the red relaxed AVL character. The following rule force the update sets and green nodes to move up.

Rule 3 (Update)

Guard: *A subtree* $n(p(A,B),q(C,D))$ *all three nodes have an update set.*

Behavior: *Changes to* $n'(p'(A,B),q'(C,D))$. *The nodes* p', q' *are equal to* p, q' *without updates. The nodes* n *and* n' *have the same color. As the update of* p *has the form* $\{n := k\}$, *therefore* $\mathbf{key}(n') = k$ *and* $\mathbf{update}(n') = \mathbf{update}(n) \cup \mathbf{update}(q)$.

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

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

3.3 Algorithms for deletions

Massively parallel case. There are two types types of waves, *Update-waves* dealing with the update sets and *green-waves* moving up green nodes. As before, we get EREW PRAM algorithm in time time $O(\log n + \log k)$ using k processors.

Distributed case. Any rule can be selected and applied as soon as its guard is satisfied. It is deadlock free. Define $REQUEST = \sum_{request(n)} 1$, $NUMBER = \sum_{green(n)}$, $GREEN = \sum_{green(n)} outside(n)$ and $UPDATE = \sum_{update(n)} outside(n)$. Any application of a rule makes the following variant strictly decrease

$$(REQUEST, NUMBER, GREEN, UPDATE).$$

4 Brother trees

In Brother trees, introduced by Ottmann, Six and Wood [6], 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.* To transform an AVL tree to a Brother tree we need to add a *white node*, denoted $\circ$, with no information between n and $\mathbf{father}(n)$ if $\mathbf{lower}(n)$ holds. In *Relaxed brother trees for insertions* 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 T. Ottmann and D. Wood [7] to deal with the parallel cases. As an example consider the following rule:

Rule 4 (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 get a EREW massively parallel dictionary in time $O(\log n + \log k)$ and k processors and also a distributed dictionary.

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 correspondende 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.