A Unified Approach to Concurrent and Parallel Algorithms on Balanced Data Structures*

Joaquim Gabarró   Xavier Messeguer
Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Campus Nord–Mòdul C6
C/ Jordi Girona Salgado, 1–3
08034 Barcelona, Spain
{gabarro, peypoch}@lsi.upc.es

Abstract

Concurrent and parallel algorithms are different. However, in the case of dictionaries, both kinds of algorithms share many common points. We present a unified approach emphasizing these points. It is based on a careful analysis of the sequential algorithm, extracting from it the more basic facts, encapsulated later on as local rules. *We apply the method to the insertion algorithms in AVL trees. All the concurrent and parallel insertion algorithms have two main phases. A* percolation *phase, moving the keys to be inserted down, and a* rebalancing *phase. Finally, some other algorithms and balanced structures are discussed.*

1 A brief history of some balanced data structures and their algorithms

Computer science deals with the management of data sets. A good example is the dictionary data type which is defined by the following operations: *testing* of a membership in the set, *insertion* of elements into the set and *deletion* of elements from the set. Dictionaries can be represented by lists, hash tables and search trees. The choice of one of them depends on the time and space requirements. We restrict ourselves to balanced schemes.

When the dictionary is consulted by only one user we have a sequential approach. If there are many users, we can follow a sequential strategy, such that one user works after another without any concurrency. However, we can find more efficient strategies because the *simultaneous* use of the dictionary by many users is desirable. For this purpose concurrent and parallel algorithms have been designed. When users work in an asynchronous way we have concurrent algorithms. When users work in a synchronous way we have a parallel algorithm (see Figure 1). These approaches have been the subject of very active research areas. Let us consider them with more detail.

*This work has been partially supported by ESPRIT LTR Project no. 20244 — ALCOM-IT and also DGICYT under grant PB95-0787 (project KOALA) and also CICIT TIC97-1475-CE and also ACI with Universidad de Chile DOG 2320–30.1.1997.

Sequential approach. In the sequential case, algorithms on trees take time proportional to the height of trees. Therefore, balanced search trees (having logarithmic height) provide an excellent basis for very efficient implementations (in fact they have optimal performance among comparison based data structures). We will also consider a slight variant of balanced search trees, namely, skip lists, where randomization is used to (probabilistically) balance the data structure. In the case of balanced trees, the elements denoted keys are ordered in a depth-first-left-right traversal. The balance criteria give us different approaches.

- AVL trees, were designed by G.M. Adel'son-Vel'skiĭ and Landis [2] in 1962. They were the first balanced trees. On AVL trees, the balance is achieved by allowing a maximum difference of 1 between the heights of the sons of any node in the tree.

- 2-3 trees, were developed by J. E. Hopcroft [1] in 1970. In these trees, all the leaves have the same depth and any internal node has 2 or 3 sons. They are considered the precursors of B-trees, introduced by R. Bayer and C. McCreight in 1972 [4].

- Red-Black trees were defined by L. Guibas and R. Sedwick [17] in 1978. In these trees, nodes can be red or black and any leaf (nil node) is black. If a node is red both its children are black. Finally, every simple path

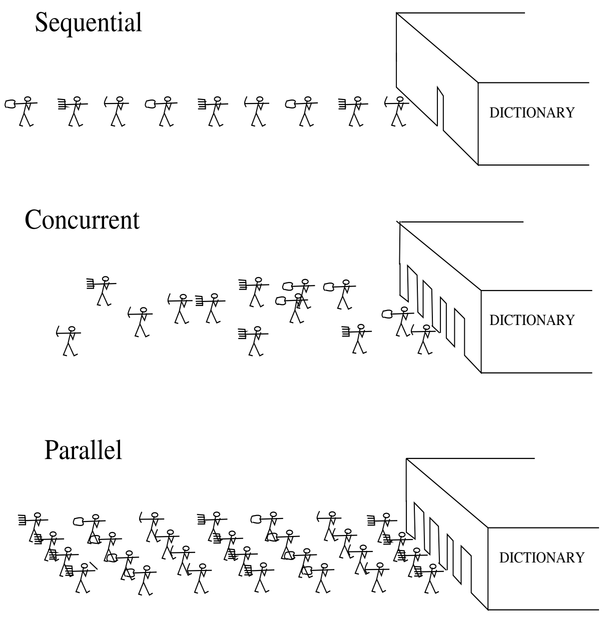

Figure 1. An intuitive view of the sequential, concurrent and parallel algorithms.

from a node to a leaf contains the same number of black nodes.

- Brother trees, were introduced by T. Ottmann, H. Six, and D. Wood [36, 35] in 1979. 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. Brother trees are quite close to AVL trees.

- Skip lists, were introduced by W. Pugh [40] in 1990. As they are composed of a set of linked lists, they are not actually search trees. However, Skip lists behave very much like trees for searching, insertion and deletion. Balancing this data type is achieved through randomization.

- Skip trees, were introduced by X. Messeguer [31] in 1997. They are quite close to skip lists. On skip trees all the leaves have the same depth and the number of sons of an internal node is determined randomly. These trees have many similarities with Brother and B-trees.

Concurrent approach. For the last fifteen years, there has been a number of attempts to design concurrent management schemes mainly for balanced trees: the goal is to allow concurrent insertions and deletions, at least as long as no race condition may occur. Locking groups of nodes in the tree during the critical updates can obviously be not avoided, but the goal is to keep those groups as small as possible, and to lock them for a time as short as possible.

The early works of R. Bayer and M. Schkolnick [5] and C.S. Ellis [11, 10] develop several solutions based on complete path optimizations. Concurrent accessing is obtained with a sophisticated locking technique with roll-backs in update. These attempts have often resulted in complex descriptions and the number of subtle details to be mastered is actually so large that proving correctness becomes hardly possible.

Later on, to avoid the preceding problems, most of the solutions are described by a set of evolution rules. In such a description, the control is kept as non-deterministic as possible. Any rule can be selected and applied to the global structure in any order as soon as its guard is satisfied. The rules assume: *temporal atomicity* (an action should correspond to a fixed, small number of assignments and tests) and *spatial atomicity* (an action should necessitate the exclusive access to a fixed, small set of neighboring nodes). The correctness can be derived from a small number of invariants. The *safety* property expresses that, if no rule can apply, then a satisfactory final state has been reached. The *liveness* property expresses that eventually no rule applies [23]. The *independence* property expresses that rules with disjoint support commute: they may safely be executed concurrently. This approach was first undertaken by J.L.W. Kessels [19] in 1983 with the design of a concurrent algorithm to deal with insertions in AVL trees. This work has been a good starting point. For AVL trees, consider for instance the work of O. Nurmi, E. Soisalon-Soininen and D. Wood [33] dealing with concurrent insertions and deletions, the work of K.S. Larsen [24] bounding the number of rules to by applied to balance the tree, or the work of L. Bougé, J. Gabarró, X. Messeguer and N. Schabanel [7] dealing with fine grain models. The method has been successfully applied to other classes of trees. Red-Black trees have been generalized to work in a concurrent environment by O. Nurmi, E. Soisalon-Soininen [32] or J. Gabarró, X. Messeguer and D. Riu [14].

Parallel approach. Parallel dictionaries have been widely studied in the recent years. In a systolic framework, priority queues and search trees algorithms were designed by C.E. Leiserson in [27]. Later, M.J. Atallah and S.R. Kosaraju [3] developed a generalized dictionary where a sequence of operations can be pipelined at a constant rate. The situation changed in 1983 when W. Paul, U. Vishkin and H. Wagener proposed efficient PRAM algorithms to dynamically maintain a parallel dictionary on 2–3 trees working on "batches" of k keys simultaneously [37]. They have considered an EREW PRAM machine with k processors. Parallel search, insertion and deletion algorithms for k items in a 2–3 tree storing n items were shown to take time $O(\log n + \log k)$ in the worst-case. Both the insertion and deletion are rather sophisticated. Higham and Schenk have studied parallel algorithms for the dynamic maintenance of a dictionary on B-trees [18] that exhibit a performance comparable to that of the algorithms for 2–3 trees. This approach has been extended to skip lists by J. Gabarró, C. Martínez and X. Messeguer [12] and, more recently, to AVL and Brother trees by J. Gabarró and X. Messeguer [13].

Implementations. *Sequential case.* The sequential dictionaries have been widely implemented. For instance, the LEDA [28] Platform implemented in C++ by K. Mehlhorn and S. Näher contains dictionaries.

Concurrent algorithms. The work of C.S. Ellis [11, 10] on AVL and 2-3 trees contains a experimental evaluation of the proposed algorithms. More recently, N. Schabanel has evaluated the experimental behavior of a fine grained AVL algorithm [42, 7]. Also D. Riu [41, 14] has studied the experimental behavior of HyperRed-Black trees (a variation of the Red-Black trees).

Parallel dictionaries They have been implemented on massively parallel machines. T. Duboux, A. Ferreira and M. Gastaldo [9] have implemented a MIMD dictionary in a Volvox IS860 with 8 nodes using sequential algorithms on 2–3–4 trees as local data structures. More recently, M. Gastaldo [16] has implemented a parallel dictionary in a SIMD machine, the MasPar MP-1. The parallel algorithms for Skip lists have been implemented by X. Messeguer [29] in C* an the programs were tested in a CM 200. J. Petit [38, 15] has developed ParaDict, a data parallel library for dictionaries having two different interfaces. The first interface is written in C* for data parallel users and the second interface in C, for users that want to use a parallel library but not to write parallel programs. Also the programs where ran in a CM 200. The references [29, 38, 15] provide good examples of the transition of theoretical PRAM algorithms into readable and efficient machine-executable programs written in C*. Finally C. Kessler and J. Träff are developing PAD, a general purpose PRAM library written in Fork95 [20].

2 A common design approach

We were a little bit puzzled by the great number of (apparently different) approaches in concurrent and parallel algorithms. However, a more accurate reading, reveals the first quite obvious fact: all the algorithms (concurrent and parallel) were inspired by their sequential counterpart. A main difference between both classes is the role of the time. In a concurrent environment the processes (or processors) work asynchronously while in a parallel algorithm all the processes (or processors) are synchronized. But, both classes

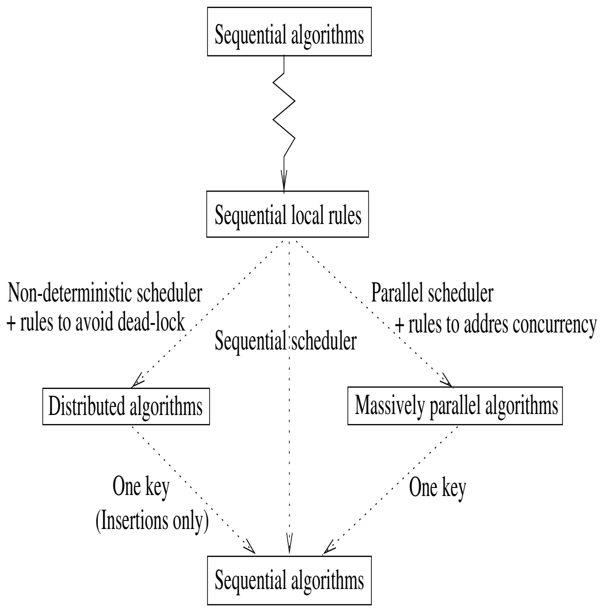

Figure 2. A common design approach to sequential, concurrent and parallel algorithms.

have another common point, if the "time scale" were rough and we were seeing a big data structure from the "distance", we would be seeing "zillions" of small local changes happening "at the same time". We would like to consider massively parallel and distributed algorithms in a very general and common setting. Local rules (issued from distributed algorithms) are a very good starting point to design massively parallel algorithms.

LOCAL RULE: It is composed by a small number of instructions which access a small and fixed number of neighbor nodes.

Quite often these rules are obtained by a careful inspection of the sequential case. This suggest the following *methodology* (see Figure 2) [30] to get a *common design framework*:

1. By a careful *analysis of the sequential algorithms* try to isolate the basic parts of the algorithm (pieces of text) updating the data structure. Write these pieces in the form of local rules. This analysis will give us a very hight level version of the sequential algorithm. This new view is a good starting point for concurrent and parallel developments.

2. To *design a concurrent algorithm* we start from the preceding rules. Massage them, quite often new rules need to be added (in order to avoid deadlock). In other cases, the rules need to be slightly modified (to keep some invariant). In this way, we get a concurrent algorithm just applying the rules concurrently. By coupling

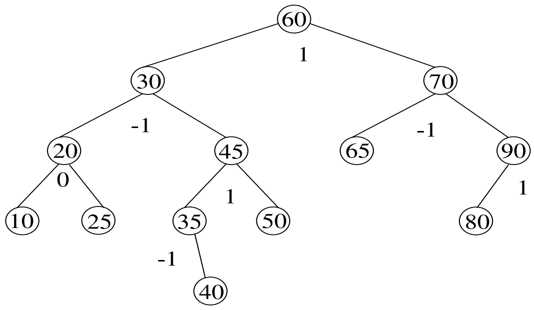

Figure 3. An example of AVL tree.

or slicing some rules, we can control the granularity of the algorithm. The proof of safety is (usually) easy but the proof of termination is more difficult.

3. To *design a parallel algorithm*, take also the basic rules and try to apply them in tightly synchronized way. If the structure is quite irregular, the main problem will be to avoid collisions between the different parts of the evolving structure.

For a more extensive development look at [30]. In the following we will apply this methodology to the insertion algorithms in AVL trees. Later on, deletion algorithms in AVL trees and dictionaries on other data structures will be sketched.

3 Extraction of local rules from the sequential insertion on AVL trees.

Let n be a node of the search tree. We denote respectively by $n \rightarrow \mathsf{p}$, $n \rightarrow \mathsf{ls}$, $n \rightarrow \mathsf{rs}$ the parent, the left son and the right son of n in the tree. The empty tree is denoted 'nil' and the root of the tree 'root'. The *real height* $\mathsf{realh}(n)$ is defined as usual:
$\mathsf{realh}(\mathsf{nil}) = 0$
$\mathsf{realh}(n \neq \mathsf{nil}) = 1 + \max(\mathsf{realh}(n \rightarrow \mathsf{ls}), \mathsf{realh}(n \rightarrow \mathsf{rs}))$
With the prefix *real* in $\mathsf{realh}(n)$ we mean that this height corresponds to the usual mathematical definition of height in binary search trees. In the case of concurrent or parallel algorithms we will find other heights with other prefixes like local, dynamical or virtual. Using the real height we define the (real)balance of a node n as

$$\mathsf{bal}(n) = \mathsf{realh}(n \rightarrow \mathsf{ls}) - \mathsf{realh}(n \rightarrow \mathsf{rs})$$

In the sequential algorithm, every node n holds a local register $\mathsf{bal}(n)$.

Be careful, other works can define the balance as $\mathsf{bal}(n) = \mathsf{realh}(n \rightarrow \mathsf{rs}) - \mathsf{realh}(n \rightarrow \mathsf{ls})$.

AVL TREE. A binary search tree is an AVL iff any node verifies $\mathsf{bal}(n) \in \{-1, 0, +1\}$ (see Figure 3).

Let us consider the *sequential insertion* algorithm [21]. Recall that this algorithm has two main phases percolation and rebalancing. In order to extract (a first version of a useful) set of local rules we will look closely at the rebalancing phase. As a convention the final state of a node n after application of a rule is denoted n'.

Percolation Phase. The key k moves down and finally arrives at the bottom of the tree. Then k is allocated "inside" an empty nil node. We give only the left version, corresponding to $k < \mathsf{key}(n)$, the right version corresponds to $k \geq \mathsf{key}(n)$. For the sake of clarity, the final state of any node n once a rule has been applied is denoted n'. Unless specified, it is identical to the initial state. The figures are drawn with the same convention.

Rule : Left Percolation

Guard: The key k points at n and $k < \mathsf{key}(n)$.
Behavior:
(1) If $n \rightarrow \mathsf{ls} \neq \mathsf{nil}$, the key k moves downward pointing at $n \rightarrow \mathsf{ls}$.
(2) If $n \rightarrow \mathsf{ls} = \mathsf{nil}$ then a new node p is allocated such that $p = n \rightarrow \mathsf{ls}$, $\mathsf{key}(p) = k$, $\mathsf{bal}(p) = 0$, $p \rightarrow \mathsf{ls} = \mathsf{nil}$, $p \rightarrow \mathsf{rs} = \mathsf{nil}$. The balance of n is updated as follows:
If $n \rightarrow \mathsf{rs} \neq \mathsf{nil}$ then $\mathsf{bal}(n') = 0$.
If $n \rightarrow \mathsf{rs} = \mathsf{nil}$ then $\mathsf{bal}(n') = 1$.
Spatial scope: Node n and the new node p.

At the end of this phase the new key has been attached and the search tree can be a little bit unbalanced. Worse, some bal registers contain unfaithful information. To solve this problem the second phase starts.

Rebalancing Phase. This phase reconstructs the tree bottom up in order to maintain balances. Recall that key k has just been added to the tree at the end of the *percolation* process. We will apply a bottom up *propagation* process which changes the value of $\mathsf{bal}(n)$ along the nodes of the restructuring path (the path going from the new leaf to the last node of the insertion path with a non-zero balance [11]). When we arrive at the critical node (last node having non zero balance), only a *rotation* will be applied if ever necessary. In the following we redefine both processes, propagation and rotation, in the form of local rules.

Propagation Rule. Let us encapsulate the change of the $\mathsf{bal}(n)$ along the structuring path by a local rule. One application of this rule just updates the balance of one node.

Rule : Left Propagation
Guard: Node n with $\mathsf{bal}(n) \in \{0, 1\}$ and $\mathsf{bal}(n \rightarrow \mathsf{ls}) \neq 0$.
Behavior: If $\mathsf{bal}(n) = 0$ then $\mathsf{bal}(n') = 1$.
If $\mathsf{bal}(n) = -1$ then $\mathsf{bal}(n') = 0$.
In both cases other registers are not modified.
Spatial Scope: Nodes n and $n \rightarrow \mathsf{ls}$.

Rotation Rules. The rotation around the *critical node* n can be also encapsulated as a local rule. As it is well known we have to perform a single or double rotation depending on the balances of n and its updated son. A single rotation is needed when n and its son have the same balance. It can be rewritten as follows (we give only one case, the symmetrical one is similar). We adopt the following notation: $n(A, B)$ denotes the (sub)tree with root n, left son A and right son B.

Rule : Single Right Rotation
Guard: A subtree $n(p(A, B), C)$ such that the balances of n and p verify $\mathsf{bal}(n) = \mathsf{bal}(p) = +1$.
Behavior: Restructure the tree into $p'(A, n'(B, C))$ with the usual updating for keys and left and right pointers, nodes n' and p' become balanced.
Spatial Scope: Nodes n and p.

Also the double rotation can be clearly rewritten as local rules. We give only one case.

Rule : Double Left Right Rotation
Guard: A subtree $n(p(A, q(B, C), D)$ with the condition $\mathsf{bal}(n) = +1$ but $\mathsf{bal}(p) = -1$.
Behavior: Restructure the tree into $q'(p'(A, B), n'(C, D))$ with the usual updating for keys and left and right pointers. Nodes p' and q' become balanced.
If $\mathsf{bal}(q) = -1$ then $\mathsf{bal}(p') = +1$.
If $\mathsf{bal}(q) = +1$ and $\mathsf{bal}(n') = -1$.
Comment: The case $\mathsf{bal}(q) = 0$ cannot appear in the sequential case.
Spatial Scope: Nodes n, p, q.

Guidelines for concurrent and parallel insertion algorithms. The preceding analysis has suggested a design in two phases:

- *Percolation phase.* In this part the set of keys move down and finally they are added to the tree. This addition can generate a highly unbalanced date structure. This process is supported by a local rule generically called *percolation*.
- *Rebalancing phase.* In this part the data structure is reconstructed. This is done with *propagations* and *rotations*.

Some freedom is needed. The rules isolated in the sequential algorithm give us hints about the structure of the algorithms to be designed. However, these rules are not "fixed" objects. They can be (in fact they will be) modified, augmented or adapted.

4 Concurrent insertion algorithms on AVL trees

Our goal is to design a general rebalancing strategy based on sets of local atomic actions applied concurrently. To ensure good concurrency, each action should lock as few nodes as possible for a time as short as possible. Thus, no reliable knowledge on the current global shape of the tree can be assumed. Each node stores in local registers its best local knowledge on the tree.

4.1 A fine grained algorithm

In this section we survey our common work with L. Bougé and N. Schabanel [6, 7, 42]. As concurrent modifications in the tree prevent from maintaining realh on each node, each node $n \neq \mathsf{nil}$ encodes its *local* knowledge of the state of the structure in two *private* registers.

In addition to the key register, $\mathsf{lefth}(n)$ and $\mathsf{righth}(n)$ are respectively the *apparent heights* of the left and right sons of n, at the best of the knowledge of n.

The application of a rule modifies the values stored into the local registers. Observe that these quantities may be arbitrarily different from their *real* values. We do not try to define accurately what we mean by *real*. Informally, to get the *(real) height* or *(real) balance* we freeze the tree and we compute these values as usual. Whenever a local register is updated with the information sent to it (at some preceding moment) we call this information *apparent*. Of course, the value of the apparent information can be very different from the value of the real information. In order to guarantee a correct final result, we need to anchor the correct values of the local height at some nodes. This means that (at least) the border nodes have an accurate knowledge about their height. These ideas are contained in the following definition

HEIGHT-RELAXED SEARCH TREE. We call *height-relaxed search tree (HRS-tree)* a search tree whose nodes are equipped with the two private registers lefth and righth satisfying the following consistency condition: $\mathsf{lefth}(n) = 0$ (resp. $\mathsf{righth}(n) = 0$) for any node n with an empty left (resp. right) son.

The following auxiliary functions on the nodes of HRS-trees will be useful.

- $\mathsf{localh}(n)$ is the *apparent local height* of u, as computed from the two previous registers :

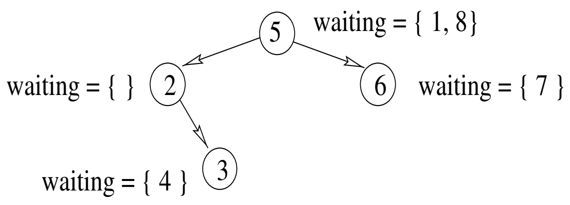

Figure 4. A strongly sorted tree with its waiting bags.

$$\mathsf{localh}(n) = 1 + \max(\mathsf{lefth}(n), \mathsf{righth}(n))$$

- $\mathsf{car}(u)$, the *carry* of u, is the gap of knowledge between u and its parent:

$$\mathsf{car}(n) = \begin{cases} \mathsf{lefth}(n \to p) - \mathsf{localh}(n) \\ \qquad\qquad \text{if } n \text{ is the left son} \\ \mathsf{righth}(n \to p) - \mathsf{localh}(n) \\ \qquad\qquad \text{otherwise} \end{cases}$$

  The car function measures the inconsistency of local information on the structure of the tree. A node u is said *reliable* if $\mathsf{car}(u) = 0$. By convention $\mathsf{car}(\mathsf{root}) = 0$.

- $\mathsf{abal}(n)$ of n is the *apparent balance* of n, defined as follow:

$$\mathsf{abal}(n) = \mathsf{lefth}(n) - \mathsf{righth}(n)$$

  A node n is said *apparently balanced* if $\mathsf{abal}(n) \in \{-1, 0, +1\}$.

If each node of an HRS-tree T is reliable and apparently balanced, then T is an AVL.

Percolation Phase. The idea is to simulate the percolation of keys in the original sequential algorithm with a new register $\mathsf{waiting}(n)$ which holds the keys waiting at node n for downwards percolation. To handle the possibility of equal keys, $\mathsf{waiting}(n)$ is managed as a *bag*. Operation $+$ adds a key to the bag. Operation $-$ removes it. This register is called the *waiting bag* at n.

If we like to build a height-relaxed tree starting from the set of keys $\{k_1, \ldots, k_N\}$, we can start with the tree having only the node n such that $\mathsf{key}(n) = k_1$, $\mathsf{waiting}(n) = \{k_2, \ldots, k_N\}$ and $\mathsf{lefth}(n) = \mathsf{righth}(n') = 0$. Later on we apply the following percolation rule. We describe this rule with a daemon. The life of a daemon runs as follow: it wakes up at some point, selects a set of nodes satisfying one of its guards and locks it while it applies the appropriate action. The selection step may be roughly implemented by a random draw among all the nodes.

Rule : HR Percolation
Guard: Node n, key $k \in \mathsf{waiting}(n)$ and $k < \mathsf{key}(n)$.
Behavior: Restructure $\mathsf{waiting}(n') = \mathsf{waiting}(n) - k$. If n has a left son, $\mathsf{waiting}(n' \to \mathit{ls}) = \mathsf{waiting}(n \to \mathit{ls}) + k$. Otherwise, create a new node p, left son of n. The apparent heights of p are set to 0, $\mathsf{key}(p) = k$ and $\mathsf{waiting}(p) = \emptyset$.
Spatial scope: Node n and the potential new node p.
Note: Symmetrically with $k \geq \mathsf{key}(n)$ and node q the right son of n.

We say that a distributed, search tree is *strongly sorted* if the following condition holds: If n is in the left (resp. right) subtree of m, and $k \in \mathsf{waiting}(n)$, then $k < \mathsf{key}(m)$ (resp. $a \geq \mathsf{key}(m)$) (see Figure 4). Of course the tree generated for $\{k_1, \ldots, k_N\}$ is strongly sorted. This tree is height-relaxed, but the local information concerning the height is really unfaithful (any node n verifies $\mathsf{localh}(n) = 1$). This is not a big surprise because *all the nodes "think" they are leaves*. To get better knowledge information, needs to flow up in the the rebalancing phase starts.

Rebalancing phase. As this phase starts at the end of the percolation phase, all the update registers are empty. Does not have sense to take care of them. In the following, update registers does not appear. We describe this phase giving a set of daemons.

Propagation rule. It propagates information upwards from a son to its parent. We only presents the variations of lefth and righth from which the registers localh, car and bal are computed.

Rule : HR Left Propagation
Guard: The left son of n isn't reliable, $\mathsf{car}(n \to \mathit{ls}) \neq 0$.
Behavior: The apparent left height of n is updated, $\mathsf{lefth}(n') = \mathsf{localh}(n \to \mathit{ls})$
Note: $\mathsf{abal}(n') = \mathsf{abal}(n) - \mathsf{car}(n \to \mathit{ls})$.
Spatial scope: n and its son.

Rotation Rules. These rules are inspired from the sequential case [2] but extended to the case where the balances of the nodes may exceed 2. These relaxed preconditions allow to rebalance any tree with any initial local knowledge. The rotation rules tend to reduce the apparent balance, but of course, can worsen not only the consistency of the local heights but also the real balance if the apparent balance was wrong.

Rule : HR Right Rotation, Unbalanced case
Guard: Node p is the left son of node n, p is reliable, $\mathsf{abal}(p) > 0$ and $\mathsf{bal}(n) \geq 2$
Action: p and n execute a right rotation with the obvious updating:

$$\begin{array}{ll} \mathsf{lefth}(p') = \mathsf{lefth}(p) & \mathsf{righth}(p') = \mathsf{localh}(n') \\ \mathsf{lefth}(n') = \mathsf{righth}(p) & \mathsf{righth}(n') = \mathsf{righth}(n) \end{array}$$

Note: $\mathsf{localh}(p') = \mathsf{localh}(n) - 1$, so $\mathsf{car}(p') = \mathsf{car}(n) + 1$.
Spatial Scope: p and its parent n.

The set of rules needs to be designed in order to cover all the possible situations. For instance, we can have a subtree $n(p(A, B), q)$ with $\mathsf{abal}(p) = 0$ et $\mathsf{bal}(n) \geq 2$. This situation is new because it cannot appear in the sequential case. The set of rules needs to be extended to cover this case, otherwise we could have deadlocks.

Rule : HR Right Rotation, Balanced case
Guard: Node p is the left son of node n, p is reliable, $\mathsf{abal}(p) = 0$ et $\mathsf{bal}(n) \geq 2$
Action: p and n execute a right rotation with the obvious updating:

$$\begin{array}{ll} \mathsf{lefth}(p') = \mathsf{lefth}(p) & \mathsf{righth}(p') = \mathsf{localh}(n') \\ \mathsf{lefth}(n') = \mathsf{righth}(p) & \mathsf{righth}(n') = \mathsf{righth}(n) \end{array}$$

Note: $\mathsf{localh}(p') = \mathsf{localh}(n)$, so $\mathsf{car}(p') = \mathsf{car}(n)$.
Spatial Scope: p and its parent n.

Rule : HR Left-Right Double Rotation
Guard: Node q is the right son of the left son p of node n, q and p are reliable, $\mathsf{abal}(p) < 0$ and $\mathsf{abal}(n) \geq 2$.
Action: p, n and q execute a left-right double rotation with the obvious updating:

$$\begin{array}{ll} \mathsf{lefth}(p') = \mathsf{lefth}(p) & \mathsf{righth}(p') = \mathsf{lefth}(q) \\ \mathsf{lefth}(n') = \mathsf{righth}(q) & \mathsf{righth}(n') = \mathsf{righth}(n) \\ \mathsf{lefth}(q') = \mathsf{localh}(p') & \mathsf{righth}(q') = \mathsf{localh}(n') \end{array}$$

Note: $\mathsf{localh}(q') = \mathsf{localh}(n) - 1$, so $\mathsf{car}(q') = \mathsf{car}(n) + 1$.
Spatial Scope: p, its parent n and its right son q.

Safety and liveness properties. The rebalancing algorithm is *safe* because the following property holds. Let T be an HRS-tree. If T' is obtained by applying on T any one of the rules described above, then T' is an HRS-tree holding the same keys than T. Moreover if no rule applies on T, T is an AVL. To prove *liveness* we consider two separate cases. First, we will take care of *negative carries.* Negative carries *flow upward* to the root where they vanish. To catch this phenomenon, we shall introduce $\mathrm{Out}(n)$, the *number of nodes* of the tree which are *not in the subtree* rooted in n, as proposed by Kessels in [19]. $\mathrm{Out}(n)$ is a kind of distance from node n to the root of the tree whose advantage is that it is left unchanged outside the spatial scope of any rule. Let us denote by NEG:

$$\mathsf{NEG} = \sum_{\mathsf{car}(n)<0} \mathrm{Out}(n) \cdot |\mathsf{car}(n)|$$

Second, we consider *positive carries.* Propagations show that car and bal seem to be correlated: their respective variations appear to have close magnitudes. We introduce the POS and BAL quantities which respectively measure the positive inconsistency of the local heights and the apparent global imbalance of the tree:

$$\mathsf{POS} = \sum_{\mathsf{car}(n)>0} \mathsf{car}(n) \quad \text{and} \quad \mathsf{BAL} = \sum_{n} |\mathsf{abal}(n)|$$

finally we introduce $\mathsf{RBAL} = \sum_{|\mathsf{abal}(n)|\geq 2} |\mathsf{abal}(n)| - 1.$

The proof of the following two facts can be found in [7, 42].

Theorem 1 (Variant) $6\,(\mathsf{NEG} + \mathsf{POS}) + 2\,\mathsf{BAL} + \mathsf{RBAL}$ *is a valid variant for the algorithm.*

Let c_{max} and b_{max} respectively denote the maximum absolute values of car and bal initially. We get the following worst convergence time bound:

Corollary 1 (Worst convergence time) *The algorithm applies at most* $6\,\mathsf{c}_{max} n(n+1) + 3\,\mathsf{b}_{max} n$ *rules to rebalance any arbitrary HRS-tree with any initial state.*

Experimental average convergence time. Intensive simulations of the rebalancing algorithm has been made by N. Schabanel [42], look also [7]. He show that the regular zigzag trees appear to be the most difficult to rebalance among the linear trees. Again, intensive simulations on the regular zigzag trees with up to $5,000$ nodes yield a worst convergence time of $\gamma.n$ rules (n measures the size of the tree) applications, where $\gamma \cong 4$. The quadratic executions are thus likely to be extremely singular. A more precise analysis of the convergence time distribution confirms the above assumption (see Figure 5). The behavior of our algorithm appears to be very smooth: the convergence time seems to follow a "Gaussian-like" distribution as well as the number of rotation rule applications. The average convergence time appears to be $\alpha.n$ with $\alpha \cong 3.5$ with a standard deviation of $\beta\sqrt{n}$ with $\beta \cong 4.1$. This Gaussian-like distribution confirms the previous result on practical worst cases: the probability of convergence time greater than $4.n$ tends to 0 as n grows. Thus, our scheme rebalances in practice a arbitrary binary tree after at most $O(n)$ rule applications.

Concurrent insertion and rebalancing. Up to now we have assumed two different phases. First, a percolation phase moves new keys to be inserted down and (when all the keys are located at the bottom the rebalancing process starts.

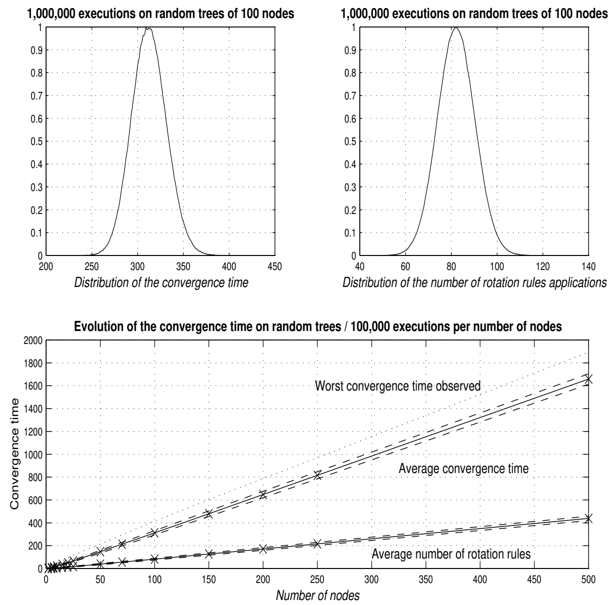

Figure 5. Average convergence time observed on $100,000$ executions over random trees of size $n = 2, 5, 7, 10, 15, 20, 25, 50, 70, 100, 150, 200, 250, 500$. The dotted lines represent the dispersion intervals.

The two phases can be interleaved. Some attention has to be paid to the relationship between percolation (moving new keys down) and rotations (moving some new keys to up). However, with a little bit of thought, the rules can be redefined.

4.2 A little bit more coarse grained approaches

We will consider the works of J.L.W. Kessels [19], O. Nurmi, E. Soisalon-Soininen and D. Wood [34] and K.S. Larsen [24]. In all these works the main idea consists on *coupling* the propagation of the information with a local rebalancing in order to maintain a relaxed version of AVL trees.

(1) Now we develop the approach taken by J.L.W. Kessels in [19]. If $\mathsf{car}(n) \in \{-1, 0\}$, we can associate colors to the nodes. This color will be stored into a register $\mathsf{color}(n)$ and can be defined as:

$$\mathsf{red}(n) \equiv (\mathsf{color}(n) = \mathsf{red}) \equiv (\mathsf{car}(n) = -1)$$
$$\mathsf{white}(n) \equiv (\mathsf{color}(n) = \mathsf{white}) \equiv (\mathsf{car}(n) = 0)$$

We may assume that a newly inserted node does not count to compute the height. A way to do it consists to color it red. Therefore, old nodes are white and the new ones are red, $\mathsf{color}(\mathsf{nil}) = \mathsf{red}$. Red nodes mean two things, first they do not count to compute the height [19] and second, they represent an unstable perturbation to be propagated up or erased as soon as possible. We recall from [19] the *dynamic height*. If n is a red leaf $\mathsf{dheight}(n) = 0$, if n is a white leaf $\mathsf{dheight}(n) = 1$, otherwise:

$$\mathsf{dheight}(n) = \mathsf{white}(n) + \max(\mathsf{dheight}(n \to \textsf{\textit{ls}}), \mathsf{dheight}(n \to \textsf{\textit{rs}}))$$

The relationship between the dynamic height and the local height is $\mathsf{dheight}(n) = \mathsf{car}(n) + \mathsf{localh}(n)$. Based on this height we have the *dynamic balance*

$$\mathsf{dbal}(n) = \mathsf{dheight}(n \to \textsf{\textit{ls}}) - \mathsf{dheight}(n \to \textsf{\textit{rs}}).$$

In this case the local knowledge of the structure is encoded in the following two registers.

Every node n holds two local registers, $\mathsf{dbal}(n) \in \{-1, 0, 1\}$ and $\mathsf{color}(n) \in \{\mathsf{white}, \mathsf{red}\}$.

Now we relax the usual definition of AVL tree in two ways. First, we replace the (real) balance by the dynamic balance in the balance property. Second, in the case of the sequential rebalancing algorithm, while the the propagation goes through an unstable node (in the critical path) this node become unbalanced. Therefore we assume that, unstable nodes (different from leaves) cannot be balanced.

RED-RELAXED AVL. A *red-relaxed AVL* is a search tree whose nodes are red or white satisfying two conditions. First, any node n is verifies $\mathsf{dbal}(n) \in \{-1, 0, +1\}$. Second, any red node n with $\mathsf{dheight}(n) \neq 0$ verifies $\mathsf{dbal}(n) \neq 0$.

When, in a red-relaxed AVL all the nodes are white, the dynamic height coincides with the real height and the dynamic balance with the balance. Therefore a red-relaxed AVL having only white nodes is an AVL.

Percolation phase. As in the case of height-relaxed trees we add a $\mathsf{waiting}(n)$ register. If we like to build a red-relaxed AVL tree starting from the set $\{k_1, \ldots, k_N\}$, we start with the tree having the node n such that $\mathsf{key}(n) = k_1$, $\mathsf{waiting}(n) = \{k_2, \ldots, k_N\}$ and $\mathsf{color}(n) = \mathsf{red}$ and we apply the following percolation rule.

Rule : Red Percolation

Guard: Node n, key $k \in \mathsf{waiting}(n)$ and $k < \mathsf{key}(n)$.
Behavior: Restructure $\mathsf{waiting}(n') = \mathsf{waiting}(n) - k$. If n has a left son, $\mathsf{waiting}(n' \to \textsf{\textit{ls}}) = \mathsf{waiting}(n \to \textsf{\textit{ls}}) + k$. Otherwise, create a new node p, left son of n with $\mathsf{color}(p) = \mathsf{red}$, $\mathsf{key}(p) = k$ and $\mathsf{waiting}(p) = \emptyset$.
Spatial scope: Node n and the potential new node p.
Note: Symmetrically with $k \geq \mathsf{key}(n)$ and node q the right son of n.

The tree builded applying this rule is a red-relaxed AVL, but the local information is unfaithful. Any node n verifies

$\mathsf{dheight}(n) = 0$ and $\mathsf{dbal}(n) = 0$, because *all of them "think" they are* red *nil* *nodes.*

Rebalancing phase. Let us consider the rebalancing problem transforming a red-relaxed AVL into an AVL. The set of rules has been designed to achieve two goals. First, to move redness up. Second to preserve the red-relaxed AVL character.

Rule : Left Red Propagation

Guard: Node n is white, $n \rightarrow ls$ is red and $\mathsf{dbal}(n) = \{-1, 0\}$.
Behavior: Node $n \rightarrow ls$ becomes white.
If $\mathsf{dbal}(n) = -1$ then n' becomes white and balanced.
If $\mathsf{dbal}(n) = 0$ then n' is red and unbalanced with $\mathsf{dbal}(n') = 1$.
Spatial Scope: Nodes n and $n \rightarrow ls$.

Rule : Single Right Red Rotation

Guard: A subtree $n(p(A, B), C)$ such that $\mathsf{dbal}(n) = \mathsf{dbal}(p) = 1$ the node n is white and p is red.
Behavior: Restructure into $p'(A, n'(B, C))$ with the usual updating. Nodes n' and p' are white and dynamically balanced.
Spatial Scope: Nodes n and p.

Rule : Double Left Right Red Rotation

Guard: A subtree $n(p(A, q(B, C), D)$ with n white, p red, $\mathsf{dbal}(n) = +1$ and $\mathsf{dbal}(p) = -1$.
Behavior: Restructure the tree into $q'(p'(A, B), n'(C, D))$ with the usual updating for keys and left and right pointers.

1. If q is white and $\mathsf{dbal}(q) = -1$ then n', p', q' are white, $\mathsf{dbal}(p') = 1$ and q', n' are balanced.
2. If q is white and balanced then n', p', q' are white and balanced.
3. If q is white with $\mathsf{dbal}(q) = 1$ then then n', p', q' are white, $\mathsf{dbal}(n') = -1$ and p', q' are balanced.
4. If q is red with $\mathsf{dbal}(q) = -1$ then q' is red and p', n' are white and $\mathsf{dbal}(q') = -1$, $\mathsf{dbal}(p') = 0$, $\mathsf{dbal}(n') = 1$.
5. If q is white with $\mathsf{dbal}(q) = 1$ then q' is red and p', n' are white and $\mathsf{dbal}(q') = 1$, $\mathsf{dbal}(p') = -1$, $\mathsf{dbal}(n') = 0$.

Spatial Scope: Nodes n, p, q.

We assume that root can always be updated as a white node (increasing by $+1$ the dynamic height of the tree).

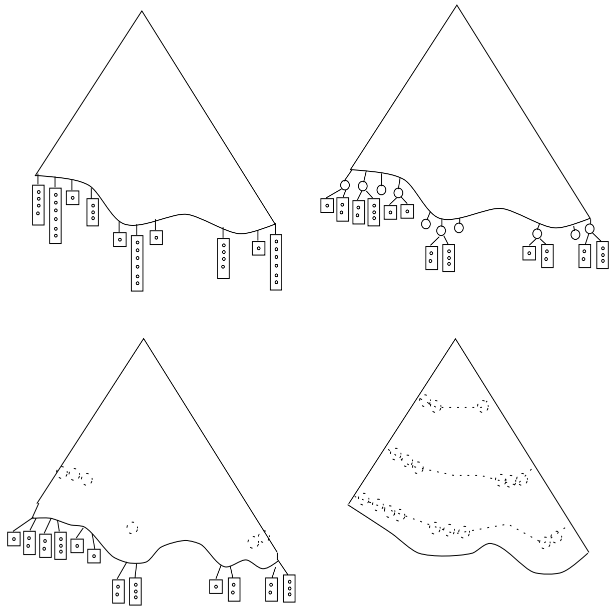

Figure 6. An intuitive view of the parallel rebalancing scheduler for insertions. First, packets are located at the bottom. Second, packets are split and the middle key is isolated. Third, middle keys start a wave. Four, the waves are chained into a pipeline.

Any application of a rule, keeps the tree dynamically balanced. A propagation rule verifies $\mathsf{dheight}(m') = \mathsf{dheight}(m)$. A colored rotation maintains the dynamic height of the subtree, for instance in single right rotation $\mathsf{dheight}(p') = \mathsf{dheight}(n)$, when the root changes from red to white it holds $\mathsf{dheight}(\mathsf{root}') = 1 + \mathsf{dheight}(\mathsf{root})$.

Safety and liveness. The algorithm is *safe* because any application of the preceding rules transforms any red-relaxed AVL into another red relaxed-AVL. Moreover all the red nodes, different from leaves, are maintained unbalanced. To prove *liveness* it is enough to verify that any application of a local rule strictly decreases the variant:

$$\mathsf{OUTSIDE} = \sum_{\mathsf{color}(n)=\mathsf{red}} \mathrm{Out}(n)$$

The degree of concurrency have two limitations. First, only red nodes having a white parent can be updated. Therefore only the top nodes of big clusters of red nodes can be updated. Second, nodes can have only two colors whiteand red, therefore the information flow up slowly.

(2) Now we consider the following extension of the J.L.W. Kessels' algorithm developed by O. Nurmi, E. Soisalon-Soininen and D. Wood in [34] (see also [33]). Every node holds the registers $\mathsf{dbal}(n) \in \{-1, 0, 1\}$ and

$\mathsf{car}(n) \in \{-1, 0, 1, 2, \ldots\}$ and they redefine the dynamic height as:

$$\mathsf{dheight}(n) = 1 + \mathsf{car}(n) + \max(\mathsf{dheight}(n \rightarrow \mathit{ls}), \mathsf{dheight}(n \rightarrow \mathit{rs}))$$

As before, the rules assume $\mathsf{car}(n) \neq 0$ and $\mathsf{car}(n \rightarrow p) = 0$.

(3) Finally let us consider the approach taken by K.L. Larsen [24]. The precondition $\mathsf{car}(n) \neq 0$ and $\mathsf{car}(n \rightarrow p) = 0$ has been relaxed by Larsen accepting nonzero values for $\mathsf{car}(n \rightarrow p)$. To do it, K.L. Larsen modifies the preceding transformations to avoid the accumulation of negative values in $\mathsf{car}(n \rightarrow p)$. As before he also couples a propagation with the rotation. He defines 13 rules.

5 Parallel insertion algorithms on AVL trees

Let us consider the case where we have to insert into an AVL tree an ordered set of k new keys stored in a sorted array $a[0..k)$, look at [13, 30]. We have two phases.

Percolation Phase. To *percolate* keys down, we can use the search algorithm by packet routing [37]. At the very start of the algorithm an active packet $p_0 = a[0..k)$ containing all k (ordered) keys is "injected" into the root of the AVL tree. In each stage, each active packet is routed down, or it is split into two packets and at least one of them is moved down. The main loop of the search ends when all the packets become inactive. At most k processors are needed to execute this loop; we need one processor to route in parallel each active packet. We will say that a subpacket $p = a[f..l)$ located at node n *hits* this node if $a_f \leq \mathsf{key}(n) < a_l$. A packet p is *active* wile is moving down, initially p_0 is active.

Rule : AVL Packet Percolation
Guard: The active packets $p = a[f..l)$ points at node n.
Behavior: There are three cases.
(1) Case $n \rightarrow \mathit{ls} \neq \mathsf{nil}$ and $n \rightarrow \mathit{rs} \neq \mathsf{nil}$, then p active and:

- *Move down.* If p does not hit $\mathsf{key}(n)$ then:

  If $a_l < \mathsf{key}(n)$ the packet moves at $n \rightarrow \mathit{ls}$

  If $\mathsf{key}(n) \geq a_f$ the packet p moves at $n \rightarrow \mathit{rs}$.

- *Split and move.* Assume the packet p hits a label. Split $p = a[f..l)$ into the subpackets $p_1 = a[f..m)$ and $p_2 = a[m..l)$ with $m = \lceil \frac{f+l}{2} \rceil$. If p_i with $i = 1, 2$ hits no $\mathsf{key}(n)$ then it is sent to the appropriate son, else it remains in n.

(2) Case $n \rightarrow \mathit{ls} = \mathsf{nil}$ and $n \rightarrow \mathit{rs} = \mathsf{nil}$, the packet p is at the bottom of the tree, the routing stops and p becomes inactive.
(3) The case $n \rightarrow \mathit{ls} \neq \mathsf{nil}$ and $n \rightarrow \mathit{rs} = \mathsf{nil}$ (reciprocally $n \rightarrow \mathit{ls} = \mathsf{nil}$ and $n \rightarrow \mathit{rs} \neq \mathsf{nil}$) can be easily defined.
Spatial scope: Node n and its sons.

This search algorithm is formally identical to the search by packet routing on Skip lists. The results about read conflicts in Split lists apply here [12] and we have the following theorems:

Theorem 2 *At most three subpackets can remain on a node.*

Theorem 3 *Given an AVL tree with n nodes, the packet routing procedure for an ordered array $a[0..k)$ takes time $O(\log n + \log k)$ using k processors on an EREW (exclusive read exclusive write) PRAM.*

At the end of the percolation phase the original packet $p_0 = a[0..k)$ has been split into a set of packet attached at the leaves (first case in Figure 6).

Rebalancing Phase. When the subpackets are located at the bottom of the AVL the *rebalancing phase* can start. The divide and conquer approach given in [37] allows us to start a wave (second and third case in Figure 6). Finally the waves can be chained into a pipeline (last case in Figure 6). AVL trees are a highly irregular data structure, pipeline information bottom up seems to be rather difficult. Let us explain how to pipeline information on AVL trees.

Pipelines schemes. Let us consider with greater detail the pipelines in AVLs. We have two cases. First, we consider how to pipeline information in an (static) AVL tree. Second, we solve the same problem in a red relaxed case.

In a (static) AVL the usual definition of depth, so called *real depth* is

$$\mathsf{reald}(\mathsf{root}) = 0$$
$$\mathsf{reald}(n \neq \mathsf{root}) = 1 + \mathsf{reald}(n \rightarrow p)$$

Given a node n we denote the brother of n by $n \rightarrow \mathit{br}$. Node n is *lower* if $\mathsf{lower}(n) \equiv (\mathsf{realh}(n) < \mathsf{realh}(n \rightarrow \mathit{br}))$. Let us define a new depth so called *virtual depth* such that

$$\mathsf{virtuald}(\mathsf{root}) = 0$$
$$\mathsf{virtuald}(n \neq \mathsf{root}) = 1 + \mathsf{lower}(n) + \mathsf{virtuald}(n \rightarrow p)$$

This new depth give us the invariant:

$$\mathsf{realh}(\mathsf{root}) = \mathsf{virtuald}(\mathsf{nil}) = \mathsf{virtuald}(n) + \mathsf{realh}(n)$$

Therefore it is possible design "*virtually plane waves* such that every pair of nodes p and q into the wave front verify: $\mathsf{realh}(p) = \mathsf{realh}(q)$ and $\mathsf{virtuald}(p) = \mathsf{virtuald}(q)$. Based on this (if we assume that informations flow does not modifies the tree) we can design a pipeline flowing bottom-up (see Figure 7).

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.

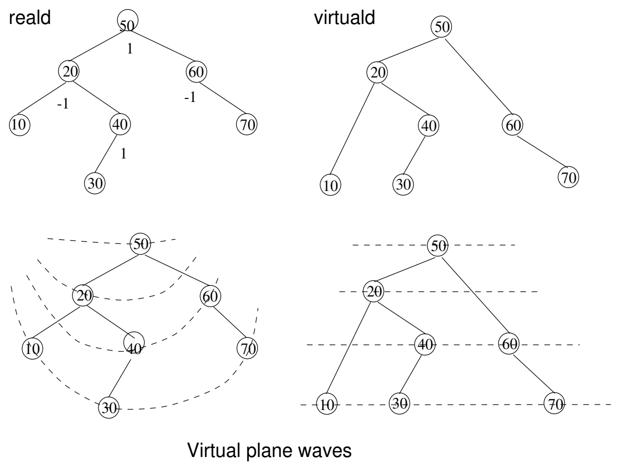

Figure 7. An example of real depth reald**, virtual depth** virtuald **and virtual plane waves in a static AVL tree.**

We would like to work in a parallel environment were a front of unstable nodes rises up. To deal with this phenomenon, accept red nodes. However, as the information rises up in parallel, does not have sense to consider red nodes unbalanced, therefore we assume the following weaker definition of red AVL trees.

WEAK-RED-RELAXED AVL. A *weak-red-relaxed AVL* is a search tree whose nodes are red or white and any node n is verifies $\mathsf{dbal}(n) \in \{-1, 0, +1\}$.

We define the *virtual dynamic depth* for relaxed AVLs,

$$\mathsf{virtual\text{-}ddepth}(\mathsf{root}) = 0$$
$$\mathsf{virtual\text{-}ddepth}(n \neq \mathsf{root}) = \mathsf{white}(n) + \mathsf{dlower}(n) + \mathsf{virtual\text{-}ddepth}(n{\to}\mathsf{p}).$$

In a *weak-red-relaxed AVL* any node n verifies

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

and all the nil nodes have the same *virtual dynamic depth*. Based on this we come to our second design scheme.

RED DYNAMIC PIPELINE SCHEME: Using *virtual dynamic depth*, the leaves become aligned at the bottom of a weak-red-relaxed AVL. 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 chain waves to get a pipeline in weak-red-relaxed AVL.

Description of daemons. We can easily get a set of rules in the same spirit of J.L.W. Kessels [19] dealing first, with weak-red-relaxed AVL trees and second, with parallel red nodes. In order to give a flavor of these rules consider a parallel red propagation.

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, p' and q' become white and n' becomes red.
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. The rules can be designed to get the following condition. 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

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

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:

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

Parallel scheduler for insertions. We will take the usual set of rules and apply them in a synchronized way based on our second design scheme. We have three cases having increasing complexity (as in the case of 2-3 trees [37]).

- All the keys to be inserted are attached to a white node.
- We have an AVL with a "red beard" on the bottom.
- We have to insert the sorted array $a[0..k)$.

(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 active nodes with dynamic height 1, applies the corresponding rule,
- all active nodes with dynamic height 2, applies the corresponding rule,
- iterate, increasing the dynamic height.

We can see the set of active nodes as a wave going bottom-up the tree.

Lemma 1 *The nodes that belong to the front wave have the same* virtual dynamic depth.

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 $\mathsf{dheight}(w)$ and $\mathsf{virtual\text{-}ddepth}(w)$.

(2) Let us consider the case of an AVL with a "red beard on 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, any virtual plane wave will not be affected by the behavior of other waves higher in the tree. Therefore It holds:

Theorem 4 *Take a relaxed AVL 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*

$$\begin{aligned}\lambda &= \mathsf{dheight}(w) - \mathsf{dheight}(w') \\ &= \mathsf{virtual\text{-}ddepth}(w') - \mathsf{virtual\text{-}ddepth}(w)\end{aligned}$$

while w and w' are moving up.

(3) Finally let us consider the case where we have an AVL with n keys (now n is a number not a node) and we have to insert the array $a[0..k)$. First, we employ the search algorithm by packet routing [37, 18, 12]. When the subpackets are at the bottom of the tree, the divide and conquer strategy [37] allows us to start a pipeline (see Figure 6) and the results of (2) assure that

Theorem 5 *The 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.*

6 Deletions in AVL trees

The unified approach can also be extended to deletion algorithms. Some hints, in the case of concurrent algorithms, can be found in [33, 6]. In the case of parallel algorithms the method has been developed in [13].

7 Other balanced structures

The study of concurrent and parallel dictionaries is a broad subject of research. In the following we comment briefly some other approaches. An interesting complementary survey on concurrent algorithms has been recently written by E. Soisalon-Soininen and P. Widmayer [43].

Distributed algorithms *Red-Black trees.* These trees have been a good starting point for the study of concurrent algorithms based on local rules. We consider only two extension, the HyperRed-Black trees [14, 41] and the Chromatic trees introduced by O. Nurmi, E. Soisalon Soininen [32]. The HyperRed-Black trees [14] are defined to work well with insertions. This can be done accepting *negative* values, therefore nodes can have color $\mathsf{color}(n) \in \{1, 0, -1, -2, -3, \cdots\}$. Node n is *black* if $\mathsf{color}(n) = 1$, *red* if $\mathsf{color}(n) = 0$ and *hyper-red* if $\mathsf{color}(n) \in \{-1, -2, -3, \cdots\}$.

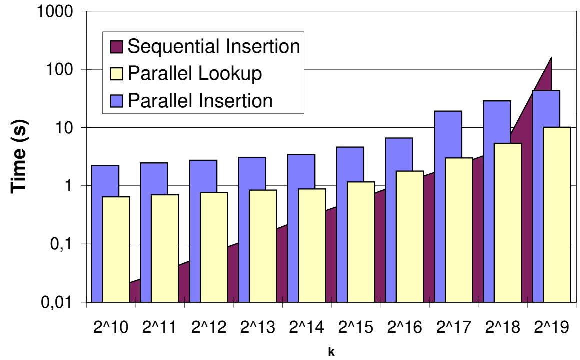

Figure 8. Running time for searching / inserting k keys in a dictionary of size n=150000 on a CM 200 with 16K processors. In the background, insertions on a Sun Sparc 10.

HYPER RED-BLACK tree is a binary search tree such that, every node is either red, black or hyper-red, every leaf (NIL) is black and every simple path from a node to a leaf has the same sum of colors.

Chromatic trees were designed to work efficiently with deletions. Then *positive* colors are accepted, nodes can have color $\mathsf{color}(n) \in \{0, 1, 2, 3, 3, \cdots\}$. Node n is *red* if $\mathsf{color}(n) = 0$, *black* if $\mathsf{color}(n) = 1$ and *overweighted* if $\mathsf{color}(n) \in \{1, 2, 3, \cdots\}$.

CHROMATIC tree is a binary search tree such that, every node is either red, black or overweighted, every leaf (NIL) is black and every simple path from a node to a leaf has the same sum of colors.

An efficient rebalancing procedure for Chromatic trees has been found by J. Boyar and K. Larsen [8].

D. Riu has experimentally compared [41] HyperRed-Black trees with Chromatic trees evaluating the (average) time needed to transform a linked list into a Red Black tree. This time is computed by counting the number of steps needed to obtain a Red Black tree. It is possible to choose a *locally stable* node, where no rule applies. In this case, the rebalance of the tree does not progress at all and we count this step as a *failure*. Otherwise, the node is *locally unstable* and can be updated. We make progress and we count this step as a *success*. The Figure 9 plots failures and successes in the case of a linked list rebalancing into a Red Black tree. *2-3 trees and B-trees.* The evolution of the 2-3 trees can be

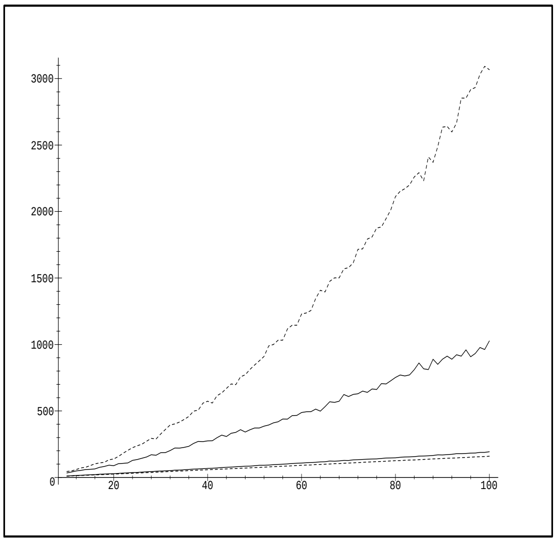

Figure 9. Number of failures (two upper lines) and successes (two down lines) needed to rebalance a linked list of $0 \leq n \leq 100$ nodes into a Red-Black tree. Solid lines depict the HyperRed-Black case. Dotted lines depict the Chromatic case.

easily coded in the form of local rules. Other works in B trees are [26, 22, 25].

Skip Lists and Skip trees. Concurrent algorithms on skip lists are not always very clear [39]. In [31] X. Messeguer has introduced the skip trees a data structure which resembles to B trees (but are actually isomorphic to skip lists). It is rather straightforward to derive concurrent algorithms for skip lists along the lines described here.

Parallel algorithms. *2-3 and B trees.* The first parallel dictionary was designed by W. Paul, U. Vishkin and H. Wagener [37] in 1983. This dictionary runs over 2–3 trees. The search is based on packet routing, and insertion and deletion are based on a pipeline of "insertion waves". Recall that, any node n has 3 or 2 sons, written as $n \rightarrow$ls, $n \rightarrow$ms, $n \rightarrow$rs. Each node has labels $L(n)$ and possibly $M(n)$. Let us sketch the basic *packet routing* algorithm as is given in [37]. Initially, we have all the keys ordered in a packet $p_0 = a[0..N)$. The packet p_0 is located at the root of the 2-3 tree. Along the algorithm p_0 will be split into several subpackets located at different nodes of the tree. We will say that a subpacket $p = a[f..l)$ pointing at node n *hits* a label X if $a_f \leq X < a_l$.

Rule : 2-3 Packet Percolation
Guard: The packets $p = a[f..l)$ points at node n.
Behavior: There are two cases.

• *Move down.* The packet p does not hit any label of n. Then p moves down to the appropriate son of n, more precisely:
If $a_l \leq L(n)$ the packet moves at $n \rightarrow$ls
If $L(n) < a_f$ and $a_l \leq M(n)$ and n has 3 sons, p moves at $n \rightarrow$ms.
If $M(n) < a_f$ and n has 3 sons or $L(n) < a_f$ and n has 2 sons, p moves at $n \rightarrow$rs.

• *Split and move.* The packet p hits a label. Split $p = a[f..l)$ into the subpackets $p_1 = a[f..m)$ and $p_2 = a[m..l)$ with $m = \lceil \frac{f+l}{2} \rceil$. If p_i with $i = 1, 2$ hits no label, then it is sent to the appropriate son, else it remains in n.

Spatial scope: Node n and its sons.

This routing procedure verifies the following theorem

Theorem 6 *No more than two packets may pass each edge of the 2-3 tree at any single step.*

In the case of AVL trees, using dynamic depth we can define *virtual straight plane waves*. However, in 2-3 trees, these waves are *really* "straight line" because the dynamic depth is the real depth (if you make a picture of a weave you get a straight line). Following the previous ideas we can get:

Theorem 7 *There is a EREW massively parallel dictionary on 2-3 trees working in time $O(\log n + \log k)$ and k processors.*

J. Petit [38] (look also [15]) has implemented ParaDict, which is a parallel library for dictionaries, using the algorithms given in [37]. ParaDict is written in C*. In order to evaluate the performance of some usual operations of ParaDict, J. Petit measured and analyzed their running time on a CM 200. Experiments have been repeated enough times to yield significant figures; results shown below are a mean of a large number of runs and the variances are not substantial. The experimental results obtained for searching or inserting k keys in a dictionary storing n elements are shown in Figure 8. For comparison with a well-known workstation, we also show the times needed for the equivalent sequential insertions. We conclude that, with our machines, even if the sequential implementation is faster than the parallel one for reasonable values of k, the time increase is smoother, making clear the scalability of the parallel library.

The algorithms given in [37] were extended to B trees by L. Higham and E Schenk [18].

Skip lists. A parallel dictionary on skip lists has been designed by J. Gabarró, C. Martínez and X. Messeguer [12]. The search is made by packet routing and insertion and deletion use prefix sum and pointer jumping. The algorithms are

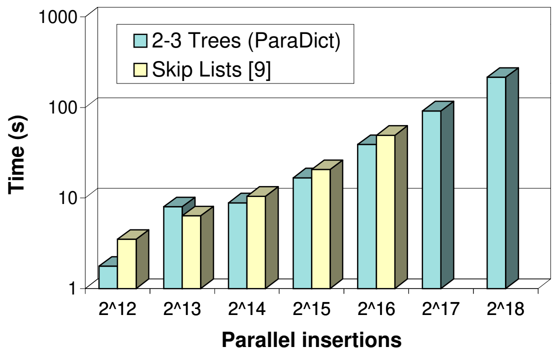

Figure 10. Running times for inserting k elements in a dictionary of size 150000 when using 2-3 trees (ParaDict) or skip lists on a CM 200 with 2K processors.

presented using a top-down design. These algorithms have been implemented by X. Messeguer [29] in C* and were also ran in a CM 200. Figure 10 compares 2-3 trees and skip lists. It seems that ParaDict's implementation with 2-3 trees is slightly more efficient than implementation based on skip lists.

Brother trees. Let us briefly comment parallel algorithms on Brother trees [36, 35]. Recall that 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. *Usually, 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 $n \rightarrow p$ if $\mathsf{lower}(n)$ holds. Therefore, any internal node n different from $\circ$, has one of the following forms, $n(A, B)$, $n(A, \circ(B))$ or $n(\circ(A), B)$ such that A, B are brother trees having a binary node as a root.

It is possible to define *Relaxed brother trees for insertions*. These trees are intuitively close to the 2–3 trees because we accept nodes with two or three sons. As before, binary nodes can have one of the forms $n(A, B)$, $n(A, \circ(B))$ or $n(\circ(A), B)$. Only two forms are allowed for 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 [36] to deal with the parallel cases. These rules allow us to move up ternary nodes.

In *Relaxed brother trees for deletions* we can have things like $n(\circ(A), \circ(\circ(B)))$ where chains of two consecutive $\circ$ appear. Extending the rules given for deletion by T. Ottmann and D. Wood [36] with the parallel cases we can get another complete set of rules to deal with deletions. Following these ideas we have been able to find algorithms with performances similar to those of other search trees.

Acknowledgments. We thank L. Bougé, C. Martínez, J. Petit, N. Schabanel and D. Riu to let to use parts of our joint work. We have got a lot of pleasant hours working all together. The Figure 5 was made by N. Schabanel, the Figure 9 was made by D. Riu and Figures 8 and 10 were made by J. Petit. Thanks to all of them. C. Martínez, has read and improved the final version, thanks for your help.

References

[1] J. U. A. Aho, J. Hopcroft. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

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

[3] M. J. Atallah and S. Rao Kosaraju. A generalized dictionary machine for VLSI. *IEEE Trans. on Comp.*, C-34(2):151–155, 1985.

[4] R. Bayer and C. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[5] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[6] L. Bougé, J. Gabarró, and X. Messeguer. Concurrent AVL revisited: self-balancing distributed search trees. Technical Report Research Report 95-45, Ecole Normale Superièure de Lyon, Laboratoire de l'Informatique du Parallelisme, 1995. Appeared with the same title as: INRIA, Rapport de Recherche 2761.

[7] L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Concurrent rebalancing of AVL trees: a fine grained approach (extended abstract). In *Proc. Euro-Par'97, LNCS*. Springer-Verlag, 1997. A extended version of the preceding work (including the proofs) can be found, with the same title, as UPC-LSI Report LSI-97-10-R.

[8] J. Boyar and K. Larsen. Efficient rebalancing of chromatic search trees. *J. Comp. Sys. Sci.*, 49(3).

[9] T. Duboux, A. Ferreira, and M. Gastaldo. MIMD dictionary machines from theory to practice. In Y. Robert and D. Trystram, editors, *Proc. of the 2nd Joint Int. Conf. on Vector and Parallel Processing (Conpar92-VappV)*, LNCS 634, pages 545–550. Springer-Verlag, 1992.

[10] C. Ellis. Concurrent search and insertion in 2-3 trees. *Acta Informatica*, 14:63–86, 1980.

[11] C. Ellis. Concurrent search and insertion in AVL trees. *IEEE Trans. Comp.*, C-29(9):811–817, Sept. 1980.

[12] J. Gabarró, C. Martínez, and X. Messeguer. A design of a parallel dictionary using skip lists. *Theoretical Computer Science*, 158(1):1–33, 1996. A preliminary version is: Parallel Update and Search in Skip Lists, in R. Baeza-Yates, editor, *Computer Science 2: Research and Applications*, pages 15–26. Plenum Press, New York 1994.

[13] J. Gabarró and X. Messeguer. Massively parallel and distributed dictionaries on AVL and Brother Trees. In Y. K. and S. Hariri, editors, *Parallel and Distributed Computing*

Systems, pages 14–17. ISCA, 1996. Look also: Parallel dictionaries with local rules on AVL and Brother trees, Report LSI-96-31-R, 1997.

[14] J. Gabarró, X. Messeguer, and D. Riu. Concurrent rebalancing on HyperRed-Black trees. Technical Report Report-LSI-97-24-R, Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistèmes Informàtics, 1997. Appeared in these proceedings.

[15] J. Gabarró and J. Petit. ParaDict, a data parallel library for dictionaries. In *Euromicro Workshop on Parallel and Distributed Processing*, pages 163–170. IEEE Comp. Soc., 1997.

[16] M. Gastaldo. *Contribution à l'algorithmique parallèle des structures de données discrètes: machines dictionnaire et algorithmes pour les graphes*. PhD thesis, ENS-Lyon, 1993.

[17] L. Guibas and R. Sedwick. A dichromatic framework for balanced trees. In *Proc. Ann. Symp. Foundations of Computer Science*, number 19, pages 8–21. IEEE Comp. Soc., 1978.

[18] L. Higham and E. Schenk. Maintaining B-trees on an EREW PRAM. *J. Parallel and Distributed Computing*, 22:329–335, 1994.

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

[20] C. W. Kessler and J. L. Traff. Language and library support for parctical pram programming. In *Euromicro Workshop on Parallel and Distributed Processing*, pages 216–221. IEEE Comp. Soc., 1997.

[21] D. Knuth. *The art of computer programming, Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[22] Y. Kwong and D. Wood. A new method for concurrency in B-trees. *IEEE Trans. Soft. Eng.*, SE-8(3):211–222, May 1982.

[23] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Soft. Eng.*, 3(2):125–143, 1977.

[24] K. Larsen. AVL trees with relaxed balance. In *Proc. Int. Parallel Processing Symposium*, number 8, pages 888–893. IEEE Comp. Soc., 1994.

[25] K. Larsen and R. Fagerberg. Eficient rebalancing of B-trees with relaxed balance. *Int. Jour. of Foundations of Comp. Sci.*, 7(2):169–186, 1996.

[26] P. Lehman and S. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. on Database Systems*, 6(4):650–670, 1981.

[27] C. E. Leiserson. *Area-Efficient VLSI Computation*. PhD thesis, Carnegie-Mellon University, 1981. ACM Doctoral Dissertation Award 1982. The MIT Press, 1983.

[28] K. Mehlhorn and S. Näher. Leda: A platform for combinatorial and geometric computing. *Comm. ACM*, 38(1):96–102, Jan. 1995.

[29] X. Messeguer. A sequential and parallel implementation of skip lists. Technical Report LSI-94-41-R, LSI-UPC, 1994.

[30] X. Messeguer. *Distributed and massively parallel algorithms with local rules on balanced search trees*. PhD thesis, UPC-LSI, 1996.

[31] X. Messeguer. Skip trees, an alternative data structure to Skip lists in a concurrent approach. *RAIRO, Theoretical Informatics and Applications*, 1997. To appear.

[32] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees, a structure for concurrent rebalancing. *Acta Informatica*, 33(6):547–557, 1996.

[33] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *ACM PODS*, pages 170–176. ACM, 1987.

[34] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrent balancing and updating of AVL trees. Technical Report 1992ITKO-B76, Helsinki University of Technology, Department of Computer Science, 1992.

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

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

[37] W. Paul, U. Vishkin, and H. Wagener. Parallel computation on 2–3 trees. In J. Díaz, editor, *Proc. 10th International Colloquium on Automata, Programming and Languages, LNCS 154*, pages 597–609. Springer-Verlag, 1983. Look also: Parallel computation on 2–3 trees, *RAIRO Informatique Théorique*, pages 398–404, 1983.

[38] J. Petit. Llibreria de diccionaris parallels: disseny, implementació, avaluacio. FIB-UPC Computing Project, January 1996. Look also: Paradict, a data parallel implementation of dictionaries with 2-3 trees, Report LSI-96-7-T, 1996.

[39] W. Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD, Apr 1989. Also published as UMIACS-TR-90-80.

[40] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Comm. ACM*, 33(6):668–676, 1990.

[41] D. Riu. Arbres roig-i-negre, algoritme distribuït. FIB-UPC Computing Project, July 1997.

[42] N. Schabanel. Équilibrage AVL distribué d'arbres binaires de recherche. Ecole Normale Superièure de Lyon, LIP, Stage de DEA, 1996.

[43] E. Soisalon-Soininen and P. Widmayer. Relaxed balancing in search trees. In D. Du and K. Ko, editors, *Advances in Algorithmics, languages and Complexity: Essays in Honor of Ronad V. Book*, pages 267–283. Kluber Academic, 1997.