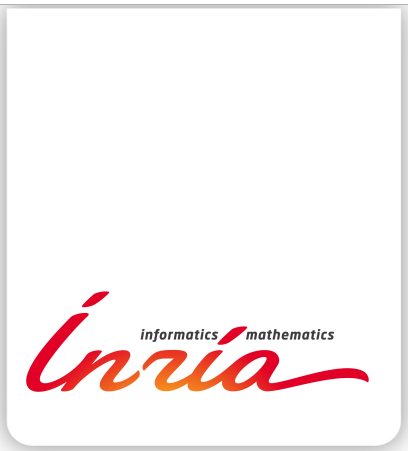
Parallelizing Structural Transformations on Tarbres

Paul IANNETTA, Laure GONNORD, Gabriel RADANNE

RESEARCH REPORT
N° 9405
April 2021
Project-Team CASH

ISSN 0249-6399 ISRN INRIA/RR--9405--FR+ENG



Parallelizing Structural Transformations on Tarbres

Paul IANNETTA[*†], Laure GONNORD[*†], Gabriel RADANNE[‡]

Project-Team CASH

Research Report n° 9405 — April 2021 — 21 pages

Abstract: Trees are used everywhere, yet their internal operations are nowhere as optimized as arrays are. Our work is in the continuity of the research on cache-oblivious algorithms for trees. In this article, we investigate the parallel properties of Tarbres–an implicit representation for AVL trees. Our first contribution is a new set of structural low-level operations which are needed to efficiently manipulate Tarbres in-place. These operations expose opportunities for further optimisations and parallelization. We provide as a second contribution new implementations which take advantage of these opportunities and proceed to a comparison with the actual expressivity of the polyhedral model framework for loop optimisation. Finally, experimental evaluation highlights the performance advantages of Tarbres.

Key-words: HPC, index trees, AVL, memory layout, cache-oblivious, library, parallelism

* University of Lyon, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA),F-69000 Lyon, France
† This work was partially supported by the ANR CODAS Project
‡ Inria, LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA),F-69000 Lyon, France

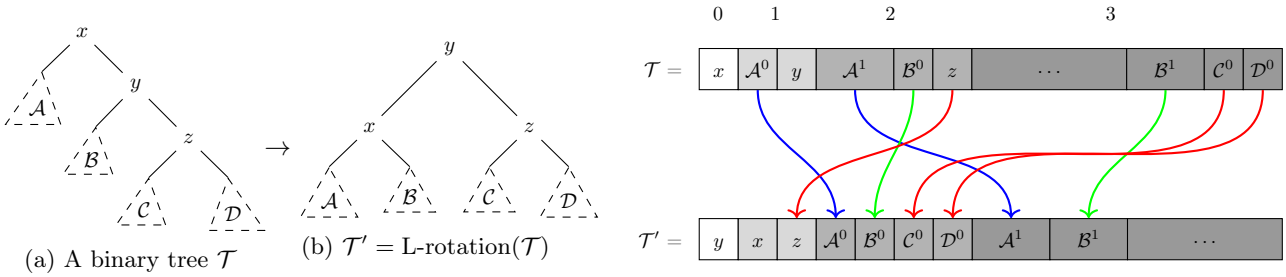
RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Parallélisation de transformations structurelles sur les Tarbres

Résumé : Les arbres sont partout, et pourtant les opérations internes des arbres restent encore moins optimisées que celles des tableaux. Notre travail s'inscrit dans la continuité de la recherche sur les algorithmes d'arbres indépendants du cache. Dans cet article, nous nous proposons d'analyser les propriétés relatives au parallélisme des Tarbres–une représentation implicite des arbres AVL. Notre première contribution se compose d'un ensemble d'opérations bas-niveau nécessaire à une maintenance efficace des Tarbres en place. Ces opérations ouvrent la porte à de nouvelles optimisations ainsi qu'à une parallélisation de leur fonctionnement interne. Notre deuxième contribution est une nouvelle implémentation qui tire profit de ces opportunités, suivie d'une comparaison avec l'expressivité du modèle polyédrique, telle qu'elle est aujourd'hui. Finalement, nous présentons nos résultats expérimentaux qui mettent en avant les performances des Tarbres.

Mots-clés : HPC, arbres indexés, AVL, modèle mémoire, indépendant du cache, bibliothèque, parallélisme



(a) A binary tree $\mathcal{T}$ (b) $\mathcal{T}' = \text{L-rotation}(\mathcal{T})$ (c) L-rotation on a Tarbre

Figure 1: Left (L) rotation applied on to the unbalanced tree T, and the associated transformation on the Tarbre representation.

1 Introduction

Trees, especially balanced trees, are at the root of efficient data-structures which need to be frequently queried such as sets, maps or dictionaries. Hence, they are often found in traditional databases as T-Trees [LC86] (a kind of balanced tree built on AVL trees [AVL62,GBY91]) or B-Trees [GBY91]. The growing need of analyzing large amount of data gathered from the internet and stored in gigantic databases requires harnessing the computing power of high performance parallel machines at their fullest. Improving the processing of trees is part of this endeavor and, a first step in this direction has been made by Blelloch et al. [BFS16], [SB19,SFB18] who investigated the benefits of bulk operations to increase parallelism.

Inspired by the community of *cache-oblivious* algorithm design, we aim to provide new implementation techniques for tree algorithms which leverage state-of-the-art super-scalar hardware and takes cache performance into account, but still falls into the framework of state-of-the-art compilers and their optimization schemes. The Tarbre memory layout uses an array-backed representation which eschew the need of the usual forests of pointers. Efficient use of this memory representation requires rethinking the classical operations on trees (insertion, deletion) in terms of adequate composition of new *structural* low-level operations on the new array-based layout; each of them exposing nice opportunities regarding locality and parallelism. We also show that the optimizations we manually propose exploit some kind of regularity of the operations and structural properties of Tarbres that resembles the polyhedral model framework. Throughout this article, we use *AVL trees* to demonstrate these low-level operations, notably via our Tarbre library implementation, but posit that the optimizations we exhibit directly apply to more sophisticated cache-oblivious algorithms.

Our experiments are performed on micro-benchmarks as well as two representative scenario usages. They demonstrate that the performance of the sequential version of Tarbres is similar or better than the explicit pointer representations, and the optimized version of Tarbres show great acceleration on parallel machines.

In Section 2 we recall the main characteristics of AVL trees as well as previous work on cache-oblivious algorithms we are building on. In Section 3 we propose a decomposition of classical operations on search trees performed directly on the Tarbre data-structure. This decomposition into *low-level* primitives which already have good locality, also enables further parallelization, as described in Section 4. Finally, Section 5 shows experimental evidence of the pertinence of the approach.

2 Inspiration and related work

2.1 Cache oblivious layouts

As the computing machines evolve toward the use of more and more complex memory hierarchies, data-structures design and complexity have also made a shift toward more precise complexity computations which capture the memory access patterns between several memory levels and their data transfer sizes [AV88].

Modern literature on data-structures algorithms classically distinguish *cache-oblivious* algorithms from *cache-aware* algorithms. In *cache-aware* algorithms, the complexity is optimized regarding a given size of cache lines whereas *cache-oblivious* [FLPR99] ones are optimized up to an unknown block size.

Since Frigo et al.'s seminal article [FLPR99], a huge number of algorithms have been studied, from matrix transpositions to FFT or sorting, the most recent and impressive descendant of these works being the *Piecewise Geometric Model index* data-structure [FV20], which relies on clever dynamic adaptation of the data-structure according to the history of queries.

Many of these work rely on variants of *search trees*. A fascinating line of research propose to encode variants of B-trees into static arrays [BDF00,BFJ02] and shows promising experimental evidence for practical efficiency. These algorithms are often quite sophisticated but still rely on in-place structural transformations on the structure of the search tree. While they have been adapted to concurrent settings [BFGK05], the internal operations on these data-structures have so far been only *sequential*. Our goal is to show how to parallelize structural modifications on such data-structures. In particular, we propose to:

- Use an implicit (pointer-free) implementation of trees under the form of arrays, where the navigation between nodes are uniquely done through arithmetic computations.
- Choose the layout used to store the trees based on previously cited experiments.
- Adapt the algorithms in order to ease parallelization through algorithmic decomposition using the structural properties of the underlying trees and careful study of data dependencies.

In the rest of this section, we present AVL trees as well as the memory layout we chose to implicitly represent them, and discuss these choices.

2.2 AVL Trees

AVL trees [AVL62,GBY91] are a classical tree representation combining the advantages of binary search trees with a self-balancing behavior which ensures performances do not degrade when data is inserted or deleted. We now recall the key definitions and algorithms.

An AVL tree is a binary search tree such that both of its children are AVL trees and that the absolute difference of theirs heights (called depth in the rest of the paper) is strictly less than one. Those trees support the same operations as standard binary search trees (*insert*, *delete* and *find*). However, in order to keep them balanced when inserting or removing an element, they also provide a mechanism called *rotation*. A left rotation is illustrated in Figure 1a and Figure 1b, and we recall the insertion algorithm in AVL in Figure 2. An insertion needs at most one rotation to preserve the balance while a deletion may require as much as $\mathcal{O}(depth)$ rotations.

AVL trees are usually given a *pointer-based* representation, shown below in C: each node contains the data, pointers to its children and its depth.

```c
struct tree {
  int data, depth;
  struct tree *left, *right;
};
```

Given this representation, a rotation is an $\mathcal{O}(1)$ operation. Additionally, both insertions and deletions impact the balance ratio of $\mathcal{O}(\lg n) = \mathcal{O}(depth)$ nodes, resulting in a complexity for both operations of $\mathcal{O}(depth)$. The self-balancing nature of AVL ensures that the *find* operation is also $\mathcal{O}(depth)$. On the other hand, operations which iterates over all the values are relatively costly as they need to follow many indirections.

2.3 AVL implicit representation with BFS

Trees can also be represented *implicitly*: embedded in an array without any pointers, following an order specified in advance. Figure 4 depicts two such layouts.

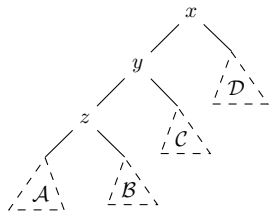
Brodal et al. [BFJ02] gives an extensive comparison of four such "implicit" layouts and the associated tree operations (insertion, computation of the children of a node at position i, search, range queries–all elements with keys within a given interval–, deletion) and their relative memory transfer complexities. It also provides experimental evaluation of these operations, that we quickly summarize here:

- All static layouts perform better than their equivalent pointer versions for trees that do not entirely fit in a cache line.

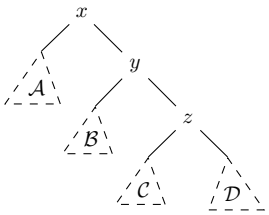
```python
def avl_insert(node, val):
  if tree is None:
    return(newNode(val));
  if (val < node.val)
    node.left = avl_insert(node.left, val);
  else if (val > node.val)
    node.right = avl_insert(node.right, val);
  else
    return node;

  node.depth = # Update depth
    1 + max(depth(node.left),depth(node.right))
  balance(node) # Rebalance the tree
  return node
```

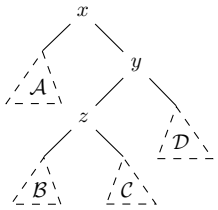
Figure 2: Insertion in an AVL



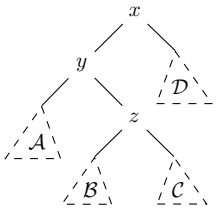
(a) Needs an R-rotation



(b) Needs an L-rotation

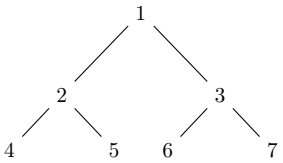


(c) Needs an RL-rotation

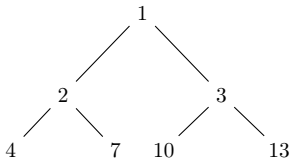


(d) Needs an LR-rotation

Figure 3: Unbalanced trees



(a) Breadth First



(b) van Emde Boas

Figure 4: Breadth First and van Emde Boas numbering

- In implicit layouts, the so-called *van Emde Boas layout*, which is a clever combination of BFS and DFS, performs better than all other layouts, particularly at very large sizes (when the size of the trees does not fit in memory and partially reside on disk). The *Breadth First layout* behaves similarly or better at smaller size, and only slightly worse on very large trees.

In the rest of this article, we will focus on the *Breadth First layout*, as it is much simpler than the *van Emde Boas layout* and provides similar performances in most cases. We believe all the techniques we present apply similarly to the *van Emde Boas layout* (albeit with more complex index computations).

3 Tarbres

We can now define the search trees under study.

Definition 1. Tarbres are AVL trees which have been laid out in an array according to a breadth-first numbering. In the rest of this article, we denote them in calligraphic letters: $\mathcal{T}$.

As stated before, our goal is to demonstrate the potentiality of parallelism of Tarbre operations. In this section, we will show how classical operations on AVL trees can be expressed in terms of low-level operations operating on Tarbres, for which we provide sequential algorithms and a complexity analysis.

3.1 Layout and tree operations

Before delving into more details, we present here for reference a simplified version which captures the gist of the internal structure that we use to represent Tarbres in memory:

```
struct tarbre {
  int * elems; // Array of elements
  size_t * depths; // Array of depths
  size_t nb_elems; // Number of values
  size_t len; // Real size of the support
}
```

Our tree is stored in a memory support consisting of two arrays, `elems` (for the values) and `depths` (for the depth of the subtree rooted in this element). `nb_elems` contains the real number of elements in the tree while `len` is the length of the underlying support. In the rest of the article, we will focus only on layout changes and movements of elements, and thus forget the `depths` field; however it should be kept in mind that all operations should also modify this field.

To describe the behavior and operations of Tarbres, we make use of some nice properties and notations coming from the breath-first numbering, described in Proposition 1 and Definition 2.

Proposition 1. In Tarbres:
- The two children indices of node i are indexed by $2i$ and $2i+1$.
- Unless i is the root, its parent has the index $\frac{i}{2}$.
- Each level k of the tree (defined by the distance to the root) is stored in the sub-array with indices belonging to the range $[2^k, 2^{k+1}-1]$ (thus of size 2^k).

Definition 2. Notations and access functions for a given Tarbre $\mathcal{T}$:
- `size(`$\mathcal{T}$`)` denotes the size of the Tarbre, *i.e.*, the number of allocated cells for $\mathcal{T}$. This quantity is always greater or equal to the number of nodes of the underlying tree.
- `depth(`$\mathcal{T}$`)` denotes the depth of the corresponding tree.
- The adjacent cells of the Tarbre $\mathcal{T}$ corresponding to level k will be denoted by $\mathcal{T}^k$ and called k^{th} *layer* of $\mathcal{T}$.
- All layers of the subtree whose root is indexed by i can be computed using the function `layers(`$\mathcal{T}$`,`i`)`.
- $i_{\mathcal{T},\mathcal{A}}$ denotes the root index of the subtree $\mathcal{A}$ in the tarbre $\mathcal{T}$. $\mathcal{T}$ is omitted whenever possible.
- The function `copy(`$\mathcal{T}$`, src, dest, nb)` makes a copy of nb (adjacent) cells from src to dest in the Tarbre. $\mathcal{T}$ is omitted whenever possible. dest should be of size equal or greater than src; the function fills the remaining cells with placeholders if required.

Most tree data-structures are read much more often than they are modified. As such an implicit layout offers many advantages: read-only operations benefits from the cache-friendly compact structure to offer great practical speed. The challenge is naturally to still provide reasonably fast modifications.

Tarbres support read operations such as `find`, `range` queries (returns all values for keys in a given range of values), or iterations (`iter`, `map`, `fold`, ...). The code of the `find` operation is shown in Figure 5. Range and iteration operations have similarly simple code.

The implementation of `insert` is the same as traditional AVL trees and is given in Figure 2. It relies on a balancing operation implemented through rotations. Traditionally, a tree-rotation is a cheap operation

```
def find(T, val)
"""Dichotomic search. Returns True is `val' is
   found, False otherwise"""
  node = 1
  while (node < T.len && T[node].depth) > 0):
    if (T[node].val > val):
      node = 2*node
    elif (T[node].val < val):
      node = 2*node + 1
    else
      return True
  return False
```

Figure 5: Pseudo-code for the `find` operation in a Tarbre.

which: 1. moves around two pointers and 2. updates the information about the depths and the balance ratio of the nodes. However, when trees are internally represented as arrays, rotations incurs several large-scale memory copies that can potentially affect the whole array.

To demonstrate this, let us take the example of an L-rotation illustrated in Figure 1. Figure 1a shows an unbalanced tree. It is balanced via an L-rotation to obtain the tree in Figure 1b. The Tarbre representation using an array is shown in Figure 1c. Each section of the array is identified by its content, which corresponds to a layer of a given subtree. As stated in Definition 2, the k^{th} layer of a tree $\mathcal{A}$ is noted $\mathcal{A}^k$. For instance $\mathcal{A}^0$ is the root of $\mathcal{A}$, while $\mathcal{B}^1$ is the children of $\mathcal{B}$ at depth 1. By looking carefully at the representation of the Tarbre before and after the rotation, we can distinguish several categories of memory movements. The subtree $\mathcal{A}$ is moved down one depth lower. This is directly transcribed in Figure 1c by the blue arrows, which move the specified memory sections to the right (i.e., "downwards" in the tree). The subtree $\mathcal{B}$ stays at the same depth, but moves to another position, as represented by the green arrows. The subtree rooted in z (and containing $\mathcal{C}$ and $\mathcal{D}$) moves upward, which corresponds to a left move in the array representation, represented by the red arrows.

As illustrated in this example, rotations are implemented by a set of memory movements on subtrees. A subtree is represented by several memory ranges, exactly one per depth level of the subtree. We can also note that memory ranges of the moved subtrees overlap, even among ranges belonging to a single subtree (see subtree $\mathcal{A}$ for instance). Figure 1 might thus give the impression that we need to copy the whole Tarbre in a new array while doing a rotation. This is not the case. Rotations can be implemented in-place efficiently thanks to a careful decomposition of rotations into lower-level operations which we present in the next section.

3.2 Low-level operations on Tarbres

Rotations are an instance of more general structural transformations on Tarbres. Many such transformations can be decomposed as compositions of low-level operations. As presented in Section 2, the breadth-first ordering of Tarbres provides a convenient index scheme which allows to view the array as a collection of layers. We can manipulate these layers thanks to low-level classes of operations, *shifts* and *pulls*.

Shifts (Figure 6) move a subtree from one position to another in the tree, as long as these positions do not overlap. In Figure 1, the movement of $\mathcal{B}$ represented by **green** arrows is a shift. Values which are moved overwrite previous values if any. The shift operation can be performed on subtrees rooted at any place in the Tarbre. The pseudo-code for the shift operation is described in Figure 7 and proceeds simply by doing memory copies of each layer. Note that since the subtrees do not overlap, these memory copies can occur in any order, which will be important for parallelism. The cells "emptied" by the move of $\mathcal{A}$ are not filled with special values: this operation will be followed by other operations that will fill the layers with relevant values.

Pull-downs (Figure 8) take a subtree and graft it at the place of its left or right child. In Figure 1, the movement of $\mathcal{A}$ represented by **blue** arrows is a left pull-down. Unlike the previous operation, this operation is not destructive, but instead "frees" a subtree, which can later be filled through one of the other operations.

The pseudo-code for the left pull-down operation is described in Figure 9. It copies each layer to the layer directly underneath. This time, the copies must be done from bottom to top to avoid overlaps. Note that the cells only occupy half of their new layers: the other subtree is left empty.

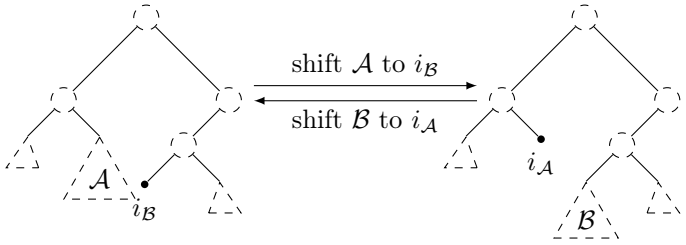


Figure 6: Subtree shifts

```
def shift(𝒯, i_𝒜, i_ℬ):
  𝒜 = layers(𝒯, i_𝒜)
  ℬ = layers(𝒯, i_ℬ)
  for k = 0 to depth(𝒜):
    # copy each layer from 𝒜 to ℬ
    copy(𝒜^k,ℬ^k,2^k)
```

Figure 7: Pseudo-code for the `shift` operation.
Shifts the subtree of $\mathcal{T}$ indexed by $i_{\mathcal{A}}$ at the position denoted by the index $i_{\mathcal{B}}$. The considered subtrees should not overlap.

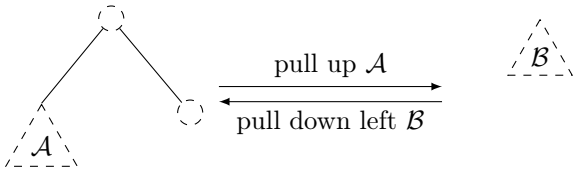


Figure 8: Subtree pull-ups and pull-downs

```
def pull_down_left(𝒯,i_𝒜):
  𝒜 = layers(𝒯, i_𝒜)
  for k = depth(𝒜) to 0: # from bottom to top
    copy(𝒜^k,A^{k+1},2^k)
```

Figure 9: Pseudo-code for the `pull_down_left` operation.
Pulls down the subtree of $\mathcal{T}$ indexed by $i_{\mathcal{A}}$ to the left.

Rotation		Right Figure 3a	Left Figure 3b	Right-left Figure 3c	Left-right Figure 3d
Steps	1.	pull-down-right$(\mathcal{T}, i_{\mathcal{D}})$	pull-down-left$(\mathcal{T}, i_{\mathcal{A}})$	pull-down-left$(\mathcal{T}, i_{\mathcal{A}})$	pull down-right$(\mathcal{T}, i_{\mathcal{D}})$
	2.	shift$(\mathcal{T}, i_{\mathcal{C}}, i_{\mathcal{C}}+1)$	shift$(\mathcal{T}, i_{\mathcal{B}}, i_{\mathcal{B}}-1)$	shift$(\mathcal{T}, i_{\mathcal{B}}, i_{\mathcal{T}.\text{left.right}})$	shift$(\mathcal{T}, i_{\mathcal{C}}, i_{\mathcal{T}.\text{right.left}})$
	3.	pull-up$(\mathcal{T}, i_z)$	pull-up$(\mathcal{T}, i_z)$	pull-up$(\mathcal{T}, i_{\mathcal{C}})$	pull-up$(\mathcal{T}, i_{\mathcal{B}})$
	4.	move-values$(\mathcal{T}, i_x, i_y, i_z)$	move-values$(\mathcal{T}, i_x, i_y, i_z)$	move-values$(\mathcal{T}, i_x, i_y, i_z)$	move-values$(\mathcal{T}, i_x, i_y, i_z)$

Figure 10: Tarbres rotations decomposed as low level operations. The function `move-values` reorders the three indexed elements to keep them sorted

```
def compress(T):
  # A DFS of T iterate in sorted order
  sorted_values = DFS(T)
  # Inserting monotone values in a tarbre
  T = tree_of_sorted_values(sorted_values)
```

Figure 11: Pseudo-code for Tarbre compression

Pull-ups (Figure 8) take a subtree and graft it in place of its parent. In Figure 1, the movement of z, $\mathcal{C}$ and $\mathcal{B}$ represented by **red** arrows is a pull-up. This is a destructive operation in the sense that the parent and one of its children subtree are overwritten. This operation cannot be performed on the root. The code for this operation is not presented, as it is similar to the pull-down operation, but in reverse.

These low-level operations can be applied on all kinds of breadth-first arrays, and do not preserve the balancing property of AVLs by themselves. We can now combine them to implement rotations.

3.3 Rotations as sequences of low-level operations

Intuitively, we can see that pull-downs free a subtree, shifts consume then free a subtree, and pull-ups consume a subtree. In order to use these operations to implement structural transformations such as AVL rotations which do not erase any data, we must compose sequences of operations similarly to a sliding puzzle[1]. Such sequences are presented for each rotation (from Figure 3) in Figure 10. Each sequence is composed of three large-scale movements on subtrees, one pull down, one shift, one pull up, along with trivial movements on individual values.

The left rotation, which was already presented in Figure 1, can thus be synthetically described as the following sequence: pull-down $\mathcal{A}$ to the left, shift $\mathcal{B}$ to the left, pull-up the subtree rooted in z and place correctly the values x, y and z. Other rotations are described similarly in Figure 10. For each shift, we note that the subtrees are clearly disjoint in the concrete operations used for the rotations.

3.4 Layout Density and Compression

When inserting an element in a Tarbre (cf Figure 2), there is a temporary state in which the Tarbre is possibly not balanced, which in our layout means that we might have to allocate a new layer. However, unless the considered tree is perfectly balanced, the Tarbre is filled with placeholders waiting to be assigned a value and the allocation is useless. This property avoids intermediary allocations, but might result in a lot of wasted space as layers are allocated which would not be required by a better balancing.

Concretely, we want to avoid Tarbres with low density (the number of relevant values divided by the size of the underlying array). Thanks to Fibonacci trees[2], we can evaluate how low this density can be. The Fibonacci tree of depth n is composed of two subtrees, the Fibonacci trees of depth $n-1$ and $n-2$ while the trees 0 and 1 are the trees with zero or one element, respectively. A Fibonacci tree is "the most imbalanced AVL": the depth difference of two sibling subtrees is always exactly one.

Proposition 2. Let us consider Fibonacci Tarbres, where a Fibonacci tree uses the Tarbre representation. If F_n is the n^{th} Fibonacci number, the Fibonacci Tarbre of depth n has density $(F_n - 1)/2^n = \mathcal{O}((\phi/2)^n)$.

Proof. The Fibonacci tree of depth n has $F_n - 1$ elements. Its Tarbre support is of size 2^n, which gives us a density of $F_n/2^n$. Furthermore, we have $F_n \sim_{n\to\infty} \phi^n/\sqrt{5}$, which concludes. □

In practice, this means that density can be arbitrarily low for large trees. For depth $n = 15$, we can already obtain densities as low as 0.018. To solve this issue we use a simple compression algorithm, depicted in Figure 11, in which sorted values are extracted from the Tarbre and used to build a perfectly balanced Tarbre. Building a Tarbre from sorted data is linear through a simple partitioning algorithm.

[1] https://en.wikipedia.org/wiki/Sliding_puzzle
[2] https://en.wikipedia.org/wiki/Fibonacci_number#Computer_science

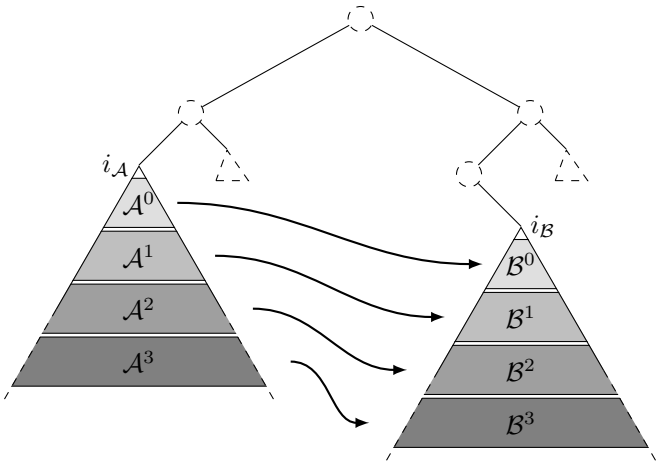


Figure 12: Shift memory movements

Proposition 3. The compress algorithm has time complexity $\mathcal{O}(n)$ where n is the size of the considered sub-Tarbre.

This compression algorithm is triggered by insertion and deletion operations when the density of the considered Tarbre falls below a chosen threshold. We determine the best threshold experimentally in Section 5.

3.5 Algorithmic complexity

From the pseudo-codes of the low-level operations shifts and pulls, we immediately obtain an algorithmic complexity of $\sum_{k=0}^{d-2} 2^k = O(2^d)$ copies, where d is the depth of the Tarbre; thus:

Proposition 4. Rotations, thus insertions (and deletions) in Tarbres are of complexity $\mathcal{O}(2^d)$ where d is the depth of the considered sub-Tarbre, ie $\mathcal{O}(n)$ if n is the size of the underlying sub-arrays in memory.

The Tarbre layout induces a clearly worse complexity than classical AVLs for which all the operations have complexity $\mathcal{O}(depth)$ (cf Section 2.2). However, the code of the low-level operations expose some parallelism that we will exploit in order to reduce the practical complexity of Tarbre operations.

4 Parallelism

So far, we have defined Tarbre operations in terms of simple low-level operations that copy pieces of the tree from one area in memory to the other. While such transformations already offer good locality, performances can be further enhanced by parallelizing the structural transformations implemented in the previous section.

4.1 Shifts

The pseudo-code implementation of the `shift` operation (Figure 7) exposes a sequence of independent memory copies. The implementation in C described in Figure 13 makes use of this independence and Figure 12 schematizes its action on the layers of a Tarbre. The source region does not overlap with the destination region and the source pointer and region pointer are guaranteed not to alias. The C99 signature of `memcpy`, which uses restrict pointers, helps the compiler detect and exploit that property. The C code also makes it clear that each layer can be moved independently of the others, allowing us to directly parallelize the `for` loop thanks to an OpenMP `#pragma`.

Another room for improvement comes from the fact the data could be moved in small chunks whose size should depend on the features of the processor such as its cache size and the size of the registers used by its vector unit. This is especially helpful for big layers since `memcpy` is always single-threaded. Splitting the data in chunks allows even more parallelism since the copy can be distributed evenly on all processors.

```c
void shift (int* t, size_t len, int idxA, int idxB) {
   int start_lvl = _greatest_bit_pos(idxA + 1);
   int end_lvl   = _greatest_bit_pos(len - 1);

  #pragma omp parallel for
  for (int i = 0 ; i < end_lvl - start_lvl + 1 ; ++i) {
    int depth = 1 << i;
    int size  = depth * (sizeof *t);
    int src   = depth * (idxA + 1) - 1;
    int dst   = depth * (idxB + 1) - 1;
    memcpy(t + dst, t + src, size);
  }
}
```

Figure 13: C implementation of the shift operation.
The two first lines compute the range of levels to be copied from $\mathcal{A}$. Adjacent cells belonging to the same level are copied independently of the other levels.

```c
void naive_pull_down_left(int* t, size_t len, int idx) {
  int start_lvl = _greatest_bit_pos(len - 1);
  int end_lvl   = _greatest_bit_pos(idx + 1);

  for (int i = start_lvl ; end_lvl <= i ; --i) {
    int depth = 1 << (i - end_lvl);
    int size  = depth * (sizeof *t);
    int dest  = depth * (2 * (idx + 1)) - 1;
    int src   = depth * (idx + 1) - 1;

    memcpy(t + dest, t + src, size);
  }
}
```

Figure 14: Naive C implementation of Pull-down Left
The layers to be copied start respectively at index $log_2(len-1), \ldots, log_2(idx)+1, log_2(idx)$. Each layer is moved to the index just below.

4.2 Pull-ups and pull-downs

Pulls are operations which move the content of a subtree either upwards or downwards. As before we use `pull-down-left` as our leading example but all this section also applies to other pulls. Contrary to shifts, the memory movements made by the naive implementation of pulls (such as Figure 9) are *not* independent, which prevents direct parallelization.

Fortunately, we can do better thanks to clever code transformations. We first showcase a naive C implementation of the pull-down left operation in Figure 14 which corresponds to the pseudo-code in Figure 9. This naive implementation proceeds by iterating from the bottom of the subtree to the top, and moving each layer downward and to the left. As we noted before, the memory copies made by each loop of the naive pull operation conflicts with each other. Therefore, this version would only be parallelized through chunking, as previously described in the `shift` implementation.

However, we can transform this loop to be much more amenable to parallelization thanks to a transformation similar to loop skewing [Laf10]. The idea is presented in Figure 15 and implemented in Figure 16. The double arrows in Figure 15 present the iteration axes in regard to the tree structure. We iterate along the leftmost branch of the subtree (variable j). For each node in this leftmost branch, we copy each layers of the right subtree downwards along the branch (variable i). For instance, in the case presented Figure 15, the $(n-2)$-th step of the iteration operates on the node b and its right subtree $\mathcal{B}$. We then move each layer of $\mathcal{B}$ downward to the root $i_{\mathcal{C}}$. This "frees" the subtree rooted in $i_{\mathcal{B}}$. We then look at a and its right subtree

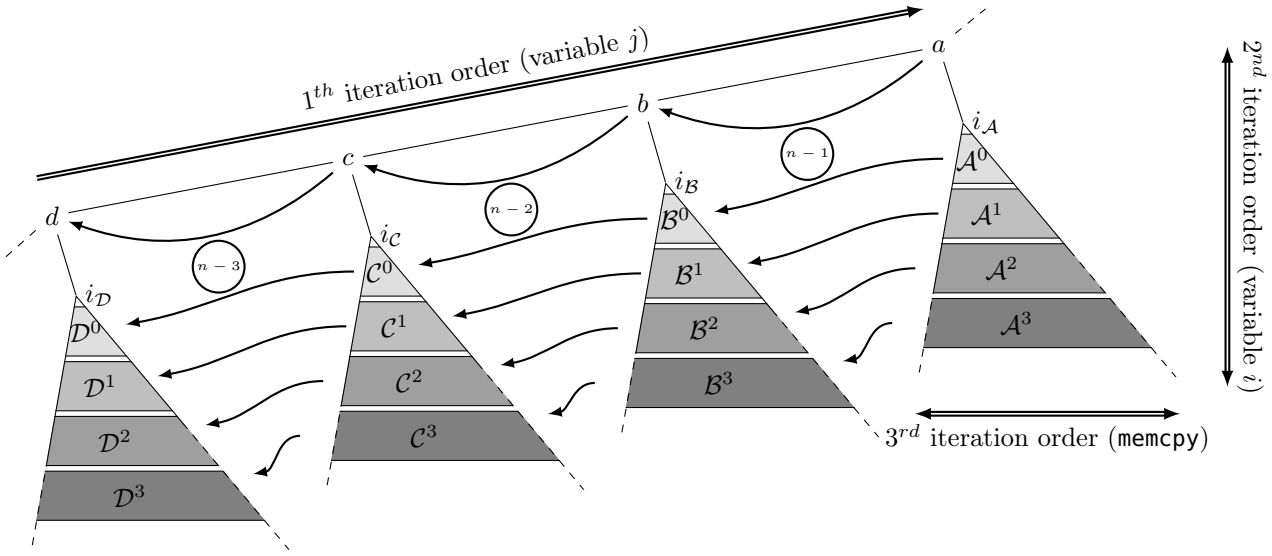


Figure 15: Optimized `pull_down_left` memory movements

```c
void pull_down_left(int* t, size_t len, int idx) {
  int start_lvl = _greatest_bit_pos(idx + 1);
  int end_lvl   = _greatest_bit_pos(len - 1);

  for (int j = end_lvl ; j >= start_lvl ; --j) {
    int cur_root = (idx + 1) * (1 << (j - start_lvl));
    t[2 * cur_root - 1] = t[cur_root - 1];

    #pragma omp parallel for
    for (int i = 0 ; i < end_lvl - (j + 1) + 1 ; ++i) {
      int depth = 1 << i;
      int size  = depth * (sizeof *t);
      int src   = depth * (2 * cur_root + 1) - 1;
      int dest  = depth * (4 * cur_root + 1) - 1;

      memcpy(t + dest, t + src, size);
    }
  }
}
```

Figure 16: Optimized C implementation of Pull-down left
Layers as described in Figure 15. For each layer, elements are copied by chunks of a predefined size in order to maximize the performance of `memcpy`.

$\mathcal{A}$, and copy its layers to the root $i_{\mathcal{B}}$.

The internal iteration now moves layers from one subtree to another with non-overlapping roots. This exactly corresponds to a shift, and we can apply the same optimizations and move the layers in parallel in a chunked manner. The loop nesting also opens up opportunities to *pipeline* the operations between layers. Indeed, if we look at the example in Figure 15, if layer $\mathcal{B}^1$ has been moved, regardless the status of the other layers, we can start moving $\mathcal{A}^1$ in its place. Unfortunately, while this optimization should provide significant gains, it is so far not automatically done by compilers as we will now see. We did not implement it due to the complexity of implementing it by hand.

These transformations, which are unlocked by skewing the iteration, directly apply to other pull operations, by iterating either top-bottom or bottom-up along the left- or right-most branch of the subtree.

4.3 Polyhedral Interpretation

The optimizations on the low-level operations `shift`, `pull-up` and `pull-down` resemble the kind of transformations that can be automatically performed by *polyhedral optimizing compilers*. The polyhedral model [Fea92a, Fea92b, Fea11] is a powerful algebraic framework that is at the core of many advances in optimization and code generation of compute-intensive kernels (operating on arrays). One reason of its success is the practicality of the operations that are expressed in terms of algebraic computations on affine sets. From a given program *with regular control* (`for` loops, in a nutshell) using arrays accessed with affine patterns, a set of dependencies is computed which enables *rescheduling*. This rescheduling often takes the form of a base change, for instance transforming an iteration on $\langle i, j\rangle$ into an iteration on $\langle i + j, j\rangle$. Then the compiler can automatically parallelize parts or the full nested loops and apply pipelining, chunking and/or vectorization during code generation.

We now study the previously presented optimizations' characteristics under the light of the polyhedral model.

4.3.1 Shifts (cf Figure 7 and Figure 13)

when copying $\mathcal{A}$ into $\mathcal{B}$, each layer $\mathcal{A}^k$ is independent (there is no read/write dependency) with all layers of $\mathcal{B}$ (globally, without looking at the iteration number k). This information is enough to ensure that the parallel `for` in Figure 13 is correct and could be used further to enable vectorization.

However, there are some difficulties to prove that the moves are independent:

- `idxA` and `idxB` are "sufficiently disjoint" when calling the `shift` operation. This could be added as assertions (since it comes from the calling context depicted in Figure 10).
- Even with this information, we still need to prove that t + src + size never overlaps with t + dest; which also means being able to make arithmetical proofs which captures the exponential behavior of 2^i.

Although rather simple by hand, integer arithmetic with exponentials is not easily automated and the independence of the loops over i is not captured by the polyhedral model. In particular, we tried to use PLUTO [BBK+08, BHRS08], a state-of-the-art tool using the polyhedral model on a modified program where the memory copy has been expanded to a sequence of individual cell copies. Without any surprise, the tool fails to parallelize the loops as the number of iterations of the inner loop depends on the iteration variable of the outer loop in a non-linear fashion.

4.3.2 Pull-down-left (cf Figure 9, Figure 14 and Figure 16)

the skewing process consists in making a base change from $\langle i\rangle$ to $\langle j, i\rangle$, the component j being the direction "along the left-most branch", and i the depth direction. After this base change, the instance of $\texttt{memcopy}_{\langle i,j\rangle}$ executed in loop number $\langle i, j\rangle$ only directly depends on $\texttt{memcopy}_{\langle i,j-1\rangle}$, which enables parallelization of the loops in direction i, and pipelining along the j direction. For the same reasons as for shifts (morally, this operation is a clever variant of shifts), even after a manual skewing, the polyhedral model tools are not capable of finding those dependencies and cannot parallelize. Finding the skewing direction automatically is even harder; however it seems natural to identify the "left-most branch direction" or "the right-most branch direction" as the new "base axis" when dealing with trees. This example also shows room for the definition of a new shape for tiling, resembling staircases with steps of size 2^i.

Through this prism, the base operations on Tarbres highlight a clear relationship between our manual parallelization effort and the philosophy of "regularity" that is at the core of the polyhedral model. Our operations exhibit regularity in two ways: a structural regularity captured by the "father-son" relation along the left-most or right-most branch; and a computation regularity through "shapes" which are not affine but of size 2^i.

Research on how to fit trees in the polyhedral model has already been conducted by Paul Feautrier and Albert Cohen [Coh99a, Coh99b, Fea98]. However, their approach takes a fairly different path, using a transducer based-representation to represent how recursive programs operate on (pointer-shaped) trees, and computation of dependencies is reduced to the problem of finding whether a given chain belongs to an algebraic language. We believe that this approach, despite its originality and expressivity, relies on a framework which is too sophisticated and costly, and it has thus far not led to an efficient code generation

algorithm. Moreover, we think that it is not versatile enough to be combined with other classical polyhedral model optimizations which we wish to leverage.

4.4 Implementation

A first prototype for Tarbres has been implemented in C++ and his available online. [3]

We also provide a reference implementation of traditional pointer-based AVLs, which is used for testing and benchmarks. For these two versions, the operations of insertion, find, delete and maps are provided. We implemented parallelization through two means:

- an OpenMP implementation using `#pragma` as described above. However, it turns out that OpenMP fails to capture the parallelism of the Pull operations. Indeed, OpenMP cannot parallelize the inner loop because the ending condition $i < \text{end\_lvl} - (j+1) + 1$ is dependent on the iteration of the outer loop j.
- We therefore also made an implementation with manual uses of pthreads with a reusable pool of threads. This implementation uses the fact that the layers in the shift operation can be moved independently and, further split the `memcpy` calls into multiple `memcpy` calls (chunking). This is because `memcpy` is single-threaded, and it is better to launch as many as `memcpy` tasks as possible.

5 Experiments

We now present and analyze our experimental results. Our goal is to 1. tune our implementation, notably compression 2. evaluate Tarbres on various types of loads and complete use-cases 3. evaluate our parallel implementations on synthetic micro-benchmarks. The experiments have been done on a machine equipped with an Intel® Xeon™ Gold 6130 CPU @ 2.10GHz, for a total of 32 cores, 377GB of RAM and 22.5MB of cache. All code is compiled with `-O3`.

5.1 Tuning and compression

In Section 3.4, we use a notion of *threshold* to trigger compression. When density is too low, locality and access performances starts to degrade and size overhead grows. We must therefore preserve the density of Tarbres to preserve good performances. This is similar to [BFJ02] which uses a density criterion to balance cache-aware search trees. To properly tune this parameter, we measure the time required to insert 2^x elements in a Tarbre, depending on this density threshold. The results are given in Figure 17 with one curve per x.

We immediately observe that the time taken explodes as soon as we use a threshold greater than 0.25, which is an immediate consequence of the fact that AVLs trees are at most unbalanced of one level: from a tree with a density slightly above 0.25, adding one element will very often add a new level, and in consequence decrease the density drastically. We also observe that for small numbers of elements (less than $2^{17} \approx 130000$), compression does not really matter. On bigger Tarbres, while the measures are fairly unstable, it seems a threshold of around 0.15 provide the best performances. We set this parameter for the remaining benchmarks.

5.2 Macro benchmarks

5.2.1 Map operations

As a first simple measurement, we consider the in-place sequential `map` operation on trees. This benchmark aims to capture the performance behavior of all iterations on whole trees, which are usual operations provided on dictionaries for instance. Our expectation is that iterations should be very favorable to Tarbres compared to AVLs, as this is where cache-friendliness should shine. To focus on this aspect and avoid mixing parallelism concerns, we only consider *sequential* implementations here. Figure 18 shows the performance results for trees of size 2^{13} to 2^{25}. The vertical axis shows the time in seconds on a log scale. As expected, Tarbres, using the implicit layout, are around 10 times faster than pointer-based AVLs. Other iteration and range operations are also similarly fast using Tarbres.

[3] https://gitlab.inria.fr/paiannet/calv/-/tree/next

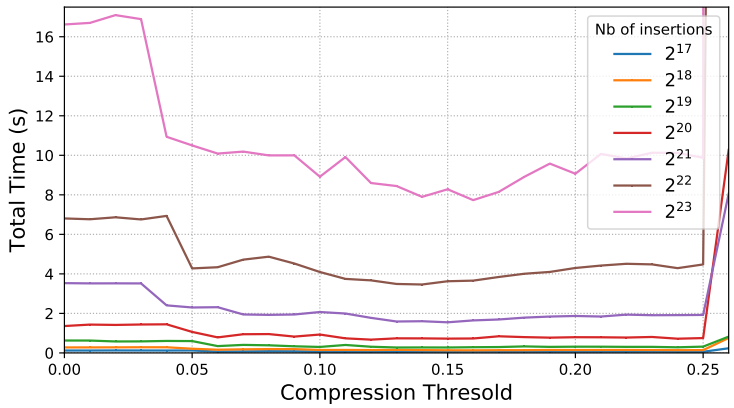


Figure 17: Performance of Tarbre creation in function of the compression threshold for several sizes.

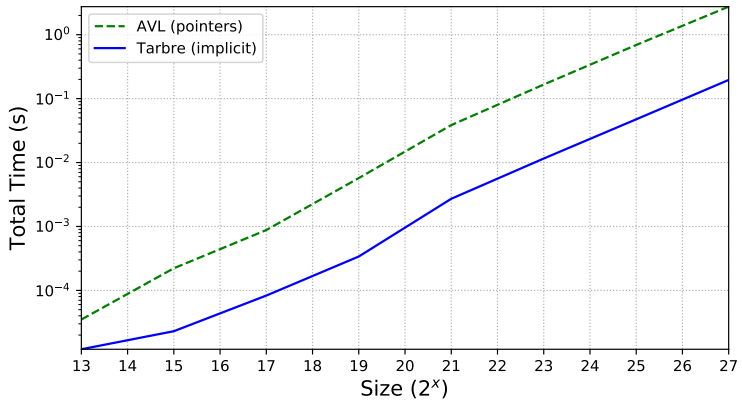


Figure 18: Performance of the in-place `map` operation on trees
Average timing of a single operation (log/log scale), for growing sizes of Tarbres (2^x is the size of the underlying array). The computation of the average is done by a program that sequentially runs 100 times the shift operation.

5.2.2 Scenario – Naming Environment

We now consider a more complex scenario to evaluate the practical performances of Tarbres: a dictionary used as naming environment in a compiler. Indeed, compilers, in particular during name resolution and type-checking, heavily rely on efficiently adding and finding names of functions and variables. To evaluate this use-case, we considered the OCaml type-checker. OCaml [LDF+21] is a functional statically-typed language known for its rich type system and efficient compiler. Typing in general and name resolution in particular is a fairly performance-sensitive operation, and the type-checker uses a pointer-based AVL as name environment. An additional challenge is that this environment is used in a *persistent* manner. Since variables respects lexical scoping in OCaml, new variables are registered in a new independent naming environment that is discarded when the scope under consideration is closed.

We instrumented the implementation of the naming environment to log all its operations, so that we can replay them with different tree implementations. Naming environments and names are represented by unique identifiers. We also logged when a scope is closed, to indicate that a particular version of the naming environment is freed. A short excerpt of the log is shown below which demonstrates the type of usage pattern found in this scenario. The `add` and `addl` operations indicate *persistent* insertions of one or multiple values, `find` des a lookup in the tree, and `free` frees the tree. Tree 0 is empty.

```
1223 <- addl(0,kind_478,layout_479) // Tree creation
1224 <- add(1223,arr_483) // Tree extension
find(1224,arr_483) // Lookup in tree 1224
find(1224,c_init_460)
...
free(1224) // Tree 1224 is freed
find(1223,create_430) // Lookup in tree 1223
```

We created the log of operations done by the naming environment when compiling the OCaml standard library and replay it in our Tarbres setting. The results are shown in Figure 19. As we have seen, the persistent usage in this scenario makes it particularly advantageous for pointer-based representation, since persistence is cheap for such implementations. Nevertheless, sequential Tarbres are slightly faster than pointer-based AVL trees on this benchmark. This shows that even on persistent workloads, the overhead of linear operations is well-compensated by the improved locality on small to medium-sized trees. We also observe that the trees stay dense (above 50%), which makes the compression irrelevant in this particular case.

	AVL (pointers)	Tarbre (implicit)
Time (s)	$1.5 \pm 0.01s$	$1.4 \pm 0.01s$
Average memory	11.5Mo	12.5Mo

Figure 19: Experimental results for name-resolution operations

5.2.3 Scenario – A key-value database

Key-value databases are big dictionaries which are generally implemented using variants of B-trees. To evaluate Tarbres in a similar context, we used the scenario described by the Influxdb team [4] to compare several key-value stores such as LevelDB (which uses Log Merge Trees) and LMDB (which uses B+-Trees). We do not attempt to compare our performance against their (highly fine-tuned) implementation, but we use their scenario to evaluate our sequential Tarbre implementation against the pointer-based one. The scenario is as follows:

1. Insert n random keys in a fresh database;
2. Delete $n/2$ random keys;
3. Compress the database;
4. Read $n/2$ random keys;
5. Insert $n/2$ random keys.

The experiment has been conducted with $n = 100$K and $n = 10$M with similar results. The results of this experiment with $n = 1$M can be seen in Figure 20. Timewise, our Tarbre layout either beats or is as good as standard AVL trees. In particular, there is a noticeable speed-up on 'delete' and 'read' operations. The speedup becomes even more noticeable as n grows. However, this speed up as a price, for $n = 1$M, standard AVL trees use 73MB where Tarbres use 206MB, this difference is mostly due to the number of placeholders the Tarbre implementation has to keep tabs on. As is common in key-value stores, the compression step is explicit and called manually. While compression is automatic for Tarbre, an explicit call before starting the read heavy section to maximize the Tarbre density helps. The total time for Tarbres is 4.01s without the manual compression step and 3.86s with manual compression. The total time for AVL trees is 4.77s.

This scenario shows that the *sequential* implementation of Tarbres is already competitive with traditional AVL trees on varied workloads. Let us now look at the gain provided by parallelization.

5.3 Parallel Micro Benchmarks

To evaluate our parallelization effort in isolation, we measure each low-level operation in turn: Shift, Pull-up et Pull-down. We compare three implementations: a *sequential* one, which is a straight C++ implementation

[4] https://www.influxdata.com/blog/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/

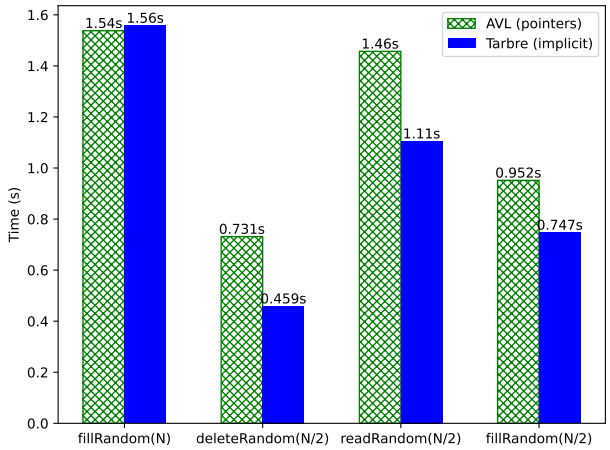


Figure 20: Performance of sequential Tarbres vs standard AVL trees on each part of the scenario with $n = 1000000 (\approx 2^{20})$

of the pseudo-code presented in Section 3.2. We also implemented two parallel implementation using the strategies described in Section 4: one using OpenMP, the other directly with pthreads.

Figure 21, Figure 22 and Figure 23 show the results for `shift`, `pull-down` and `pull-up` respectively. They report the average time of the given operation as a function of the size of the Tarbres. As expected, for all operations under study, the sequential version is clearly better than the other ones for "small" Tarbres ($\leq 2^{21}$); and its experimental complexity grows linearly with the size of the Tarbres (as proved in Proposition 4). As we already mentioned, the OpenMP version performs badly, our main hypothesis being that the allocation of threads follows a pattern that is incompatible to our parallelization scheme. In contrast, the pthread version which uses a pool of threads tailored to call `memcopy` behaves well; for big sized-arrays, the performance is an order of magnitude better than the sequential version.

These benchmarks show that we were able to manually exploit the parallelism that we had in mind by proposing the low-level operations on Tarbre. We nevertheless believe that there is still room for improvement using pipelining or vectorization.

6 Conclusion

In order to address the parallelism issues of self-rebalancing trees, we studied trees represented using a flat array, which we dubbed Tarbres. We decomposed Tarbres structural operations as sequences of low-level procedures than can be individually optimized. In fact, they all expose enough regularity to use classical rescheduling techniques using parallel-for and copy pipelining. After making a strong analysis of these opportunities in the light of the polyhedral model vision, we proposed an initial implementation of Tarbres as an open-source C++ library which take advantage of this regularity to provide improved cache-locality, parallelism and chunking. Even though our implementation does not use vectorization and pipelining, experimental evaluation show that it outperform pointer-based representation on read-only and iterating operations by at least an order of magnitude, and is comparable for write operations.

We believe all the optimizations we highlighted apply to tree cache-oblivious algorithms. For instance [BFJ02] uses local structural transformations that are reminiscent of our version of in-place rotations. Similarly, many key-value databases uses a notion of global rebalancing as compression which would gain from being parallelized.

Furthermore, we also believe these optimizations could be introduced automatically by compilers. In particular, our model exhibit vectorization opportunities that could be exploited through tiling and pipelining which have been successfully applied in the context of the polyhedral model [Fea92a, Fea92b, Fea11], In the future, we plan to study further an extension the polyhedral model so that it could directly optimize our operations.

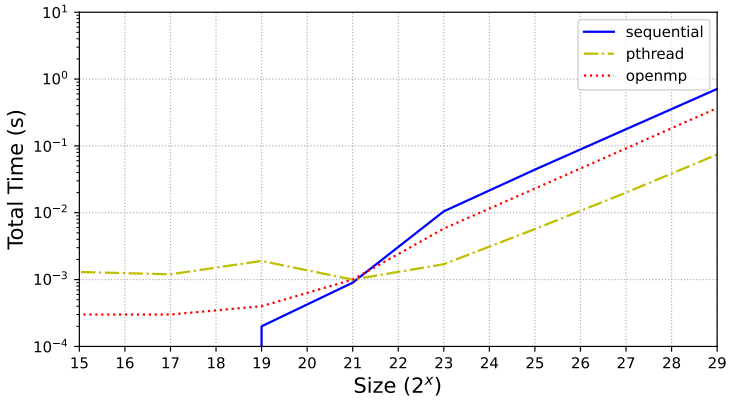


Figure 21: Performance of the `shift` operation on Tarbres

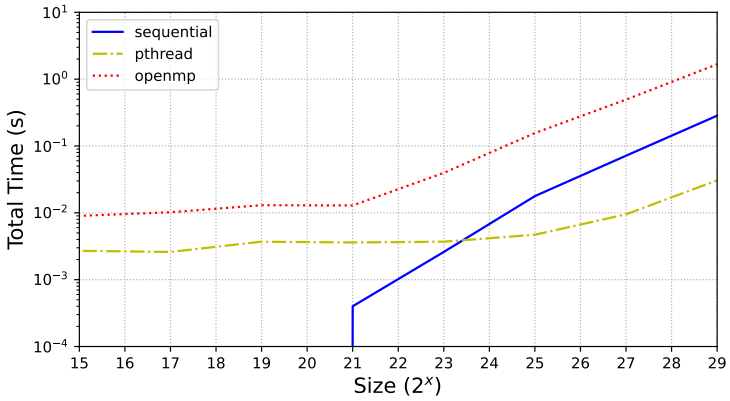


Figure 22: Performance of the `pull-down` operation on Tarbres

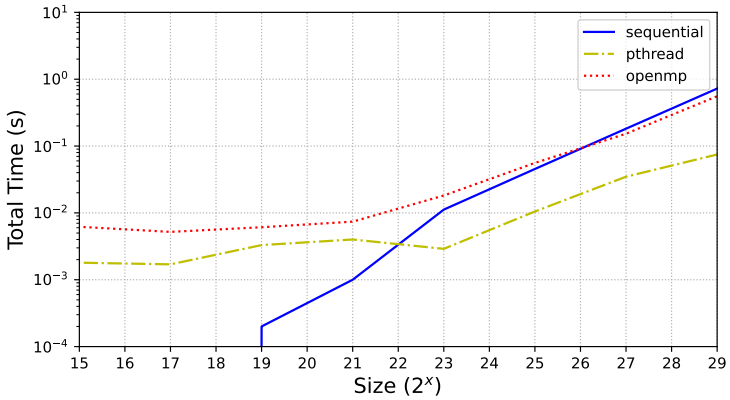


Figure 23: Performance of the `pull-up` operation on Tarbres

References

[AV88] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, September 1988.

[AVL62] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):263–266, April 1962.

[BBK+08] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)*, April 2008.

[BDF00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 399–409. IEEE Computer Society, 2000.

[BFGK05] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious b-trees. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 228–237, New York, NY, USA, 2005. Association for Computing Machinery.

[BFJ02] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'02, pages 39–48, USA, 2002. Society for Industrial and Applied Mathematics.

[BFS16] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 253–264, New York, NY, USA, 2016. Association for Computing Machinery.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[Coh99a] Albert Cohen. Analyse de flot de données pour programmes récursifs à l'aide de langages algébriques. *Technique et Science Informatiques*, 1999.

[Coh99b] Albert Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. Theses, Université de Versailles-Saint Quentin en Yvelines, December 1999.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[Fea98] Paul Feautrier. A parallelization framework for recursive tree programs. In David Pritchard and Jeff Reeve, editors, *Euro-Par'98 Parallel Processing*, pages 470–479, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[Fea11] Paul Feautrier. *Encyclopedia of Parallel Computing*, chapter Polyhedron Model, pages 1581–1592. Springer, 2011.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[FV20] Paolo Ferragina and Giorgio Vinciguerra. The PGM-Index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, April 2020.

[GBY91] Gaston H. Gonnet and Ricardo Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C*. Addison-Wesley Pub (Sd), 2nd edition, 1991.

[Laf10] Eric Laforest. Ece 1754 survey of loop transformation techniques, 2010.

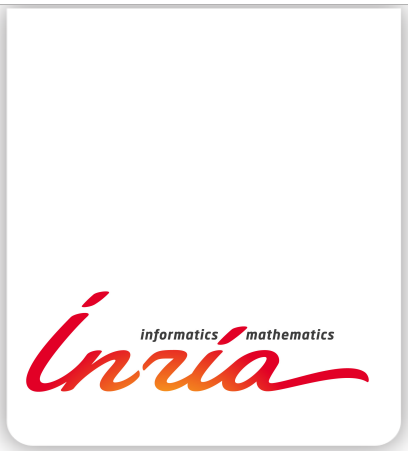
[LC86] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, page 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[LDF+21] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.12, Documentation and user's manual*. Projet Gallium, INRIA, April 2021.

[SB19] Yihan Sun and Guy Blelloch. Implementing parallel and concurrent tree structures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP'19, page 447–450, New York, NY, USA, 2019. Association for Computing Machinery.

[SFB18] Yihan Sun, Daniel Ferizovic, and Guy E. Belloch. Pam: Parallel augmented maps. *SIGPLAN Not.*, 53(1):290–304, February 2018.

Contents



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399