Implementing Parallel and Concurrent Tree Structures

Yihan Sun  
Carnegie Mellon University  
yihans@cs.cmu.edu

Guy Blelloch  
Carnegie Mellon University  
guyb@cs.cmu.edu

Abstract
As one of the most important data structures used in algorithm design and programming, balanced search trees are widely used in real-world applications for organizing data. Answering the challenges thrown up by modern large-volume and ever-changing data, it is important to consider parallelism, concurrency, and persistence. This tutorial will introduce techniques for supporting functionalities on trees, including various parallel algorithms, concurrency, multi-versioning, etc. In particular, this tutorial will focus on an algorithmic framework for parallel balanced binary trees, which works for multiple balancing schemes, including AVL trees, red-black trees, weight-based trees, and treaps. This framework allows for theoretically-efficient algorithms. The corresponding implementation is available as a library, which demonstrates good performance both sequentially and in parallel in various use scenarios.

This tutorial will focus on the following topics: 1) the algorithms and techniques used in the PAM library; 2) the interface of the library and a hands-on introduction to the download/installation of the library; 3) examples of applying the library to various applications and 4) introduction about other useful techniques for parallel tree structures and performance comparisons with PAM.

1 Introduction
Recently, the advent and development of shared-memory multi-core machines has improved the computation ability to process large-scale data in parallel and in memory. As such, it is of great interest to have simple and efficient parallel data structures to easily organize and process data. One of the most important data structures for organizing data is the balanced search tree (BST) structure, which are useful in maintaining abstract data types such as ordered maps and sets. This tutorial will introduce techniques to support parallelism and concurrency in balanced search trees for ordered sets/maps operations, show examples of applying the tree structures in various applications, as well as discuss some state-of-the-art tree structures.

We focus on writing simple and efficient parallel algorithms for trees. This tutorial introduces an algorithmic framework for parallel balanced binary trees [6, 20], which bases all tree algorithms on a single primitive Join. This framework is extendable to at least four balancing schemes: AVL trees, red-black trees, weight-balance trees, and treaps, and all algorithms except Join are generic across balancing schemes. Based on this Join-based framework, this tutorial will address techniques including many algorithms, concurrency, augmentation, persistence (meaning to yield a new version on updating) and multi-versioning. We show efficient parallel solutions to bulk operations on trees, such as Union, Filter, MapReduce, etc.. From a theoretical standpoint, all algorithms on trees are work-efficient with poly-logarithmic parallel depth.

This parallel tree framework is integrated into a C++ library called PAM (Parallel Augmented Maps) [18]. This tutorial will also discuss how to use the framework and the PAM library to solve real-world problems. The tree structure is extendable to a variety of applications in different domains, which is achieved by using an abstract data type (ADT) called the augmented map [20]. Designed as a general-purpose library for parallel tree structures, this library can be directly applied to many applications including 2D range/segment/rectangle search, inverted index searching, HTAP database systems, multi-version concurrency control, graph processing systems, and so on. Making use of the library, each of the applications only needs about one hundred lines of high-level code to get highly-optimized implementations.
The advantage of the Join-based framework and the library lies in its generality, and its simplicity and efficiency in both theory and practice.

1. Multiple real-world problems can be solved directly based on the augmented map abstraction and the PAM library. This reduces the incremental effort for users to adapt this tree structure simply as a black box to their own applications. Because of the functionalities (e.g., concurrency, persistence, multi-versioning, garbage collection) supported, users can deal with the problems on a higher level of abstraction without worrying about the details in the implementation.

2. Multiple algorithms and balancing schemes can be dealt with using generic methodology. The Join function captures all that is required for rebalancing. As a result, all algorithms except Join are generic for multiple balancing schemes. This minimizes the coding effort to re-create the tree structure with special requirements when necessary (e.g., in another programming language). Furthermore, this allows for extendability to other balancing schemes and parallel algorithms.

This tutorial will also have a hands-on introduction to the download/installation of the library. We will show code examples on how to use the framework and the library. This tutorial will finally show comparisons among different systems and tree structures under different workloads and applications, and show analysis on the favored properties for trees under different scenarios.

The library is available at https://github.com/cmuparlay/PAM. More information can be found at https://cmuparlay.github.io/PAMWeb/.

**Experimental Settings.** All experiments shown in this tutorial are tested on a 72-core Dell R930 with 4 x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache) with 1TB memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. The code was compiled with -O2 using the g++ 5.4.1 compiler which supports the Cilk Plus extensions.

**Preliminaries.** We call each element (key-value pairs) in the tree an entry, noted as $e = (k, v)$. We assume keys type $K$ and value type $V$. We define the entry type $E = K \times V$. We use $l(x)$ or $r(x)$ to extract the left or right subtree of a tree node $x$. We use $k(x), v(x)$ and $e(x)$ to extract the key, value and entry of a tree node or the root of a (sub)tree.

## 2 Algorithms

In this tutorial, we will show the Join-based algorithmic framework. The Join($T_L$, $e$, $T_R$) operation works on two balanced binary trees $T_L$ and $T_R$, and an entry $e$. It will return a new balanced binary tree in which the in-order traversal is the in-order traversal of $T_L$, then $e$, then the in-order traversal of $T_R$. In particular, when the trees are search trees, the key of $e$ should be larger than all keys in $T_L$, and smaller than all keys in $T_R$.

```plaintext
1  Split($T$, $k$) =
2    if $T = \emptyset$ then ($\emptyset$, False, $\emptyset$)
3    else if $k = k(T)$ then ($l(T)$, True, $r(T)$)
4    else if $k < m$ then let ($L', b, R'$) = Split($l(T)$, $k$)
5      in ($L', b$, Join($R', e(l(T), r(T))$))
6    else let ($L', b, R'$) = Split($r(T)$, $k$)
7      in (Join($l(T)$, $e(T)$), $L'$, $b$, $r(T)$)
8  Union($T_1$, $T_2$) =
9    if $T_1 = \emptyset$ then $T_2$
10   else if $T_2 = \emptyset$ then $T_1$
11   else let ($L_1, v', R_1$) = Split($T_1$, $k(T_2)$)
12     and $L = \text{Union}(L_1, (T_2) | R = \text{Union}(R_1, r(T_2)))$
13     in Join($L$, $e(T_2)$, $R$)
14  Insert($T$, $k$, $v$) =
15    if $T = \emptyset$ then Singleton($k$, $v$)
16   else if $k < k(T)$ then Join(Insert($l(T)$, $k$, $v$), $e(T)$, $r(T)$)
17   else if $k > k(T)$ then Join($L$, $e(T)$, Insert($r(T)$, $k$, $v$))
18   else $T$
19  MapReduce($T$, $g$, $f$, $I$) =
20    if $T = \emptyset$ then $I$
21   else let $L = \text{MapReduce}(l(T), g, f, I)$
22     and $R = \text{MapReduce}(r(T), g, f, I)$
23     in $f(L, f(g(e(T)), R))$
24  Build($S$, $i$, $j$) =
25    if $i = j$ then $\emptyset$
26   else if $i + 1 = j$ then Singleton($S[i]$)
27   else let $m = (i + j)/2$
28     and $L = \text{Build}(S, i, m)$ | $R = \text{Build}(S, m + 1, j)$
29     in Join($L$, $S[m], R$)
30  Build($S$) =
31    Build REMOVE Duplicates(Sort($S$), 0, $|S|$)
```

**Figure 1.** Example of algorithms using Join.

than all keys in $T_R$. In the sequential setting, this function was first defined by Tarjan [21], and later extended to other balancing schemes [2, 17]. Blelloch et al. describe the Join algorithms for AVL trees, red-black trees, weight-balanced trees and treaps, respectively, and use them as primitives for parallel tree algorithms. The Join function will deal with all rotation and rebalancing issues for the other algorithms. As a result, all the other algorithms are identical across multiple balancing schemes, most of them also parallel.

Figure 1 shows several examples. We show some performance numbers in Table 1. The tree scales up to $10^{10}$ tree nodes, and get 50-100x self-speedup. More experimental evaluations can be found in [6, 20] along with other operations such as Filter, Intersection, and MultiInsertion.

## 3 Augmentation

PAM provides an interface for abstract augmentation of trees. At a higher level, Sun et al. define the augmented map, which
Table 1. Timings in seconds for various functions in PAM. Here \( T_{144} \) means on all 72 cores with hyperthreads (i.e., 144 threads), and \( T_1 \) means the same algorithm running on one thread. “Speedup” means the speedup (i.e., \( T_1/T_{144} \)).

<table>
<thead>
<tr>
<th></th>
<th>( n )</th>
<th>( m )</th>
<th>( T_1 )</th>
<th>( T_{144} )</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>( 10^8 )</td>
<td>( 10^8 )</td>
<td>12.517</td>
<td>0.2369</td>
<td>52.8</td>
</tr>
<tr>
<td>Find</td>
<td>( 10^8 )</td>
<td>( 10^8 )</td>
<td>113.941</td>
<td>1.1923</td>
<td>95.6</td>
</tr>
<tr>
<td>Insert</td>
<td>( 10^5 )</td>
<td>—</td>
<td>205.970</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Build</td>
<td>( 10^{10} )</td>
<td>—</td>
<td>1844.38</td>
<td>28.24</td>
<td>65.3</td>
</tr>
<tr>
<td>Range</td>
<td>( 10^8 )</td>
<td>( 10^8 )</td>
<td>44.995</td>
<td>0.8033</td>
<td>56.0</td>
</tr>
</tbody>
</table>

Another option is to use transactions with multi-versioning, where each concurrent thread works on a (possibly isolated) version. There are two main approaches supporting multi-versioning on trees. The first one is to use version chains, in which a tree node maintains a history of versions and their values in a chain. This allows for serializability either using locking [5] or opportunistically [12]. However, the main drawback is that reading usually requires checking the visibility of each version in the chain, which can be expensive. The second approach is based on path-copying, which effectively leads to functional data structures. This avoids updating internal information of any existing tree node, and thus avoids contention caused by concurrency. In addition, any tree pointer effectively provides a snapshot to a certain version, and reading as well as writing is no more expensive than working on a single-versioned system. However, concurrent updates would result in two separate versions. To guarantee serializability, additional techniques are required, such as version melding in Hyder [15], flat-combining [10] or other batching- or combining-based approaches [3, 4, 16] that avoids concurrent writes.

PAM uses path-copying with a single writer. It is however, possible to batch updates and run them in parallel [4]. In many cases, using batching based on the bulk operations with path-copying can be more efficient than using concurrent data structures. Figure 2 shows the comparison between using PAM with batching and some state-of-the-art concurrent data structures [7, 11, 14, 22] on Yahoo! Cloud Serving Benchmark (YCSB) [8] with \( 5 \times 10^5 \) initial nodes and \( 10^7 \) transactions. For PAM, we control the batching latency within 50ms, which is approximately the typical network latency, and thus is less likely to dominate the cost. More details can be found in [4]. For all the tested workloads of mixed reads and writes, PAM with batching shows better performance than all the other concurrent data structures, especially for read-heavy workloads.

The JOIN-based algorithms can be easily extended to support persistence using path-copying. The observation is that all copying only occurs in the JOIN, so all that is needed is a JOIN operation that does path copying.

4 Persistence and Concurrency

There is a rich literature on supporting concurrency for tree structures. There have been many concurrent tree structures that support in-place concurrent update, e.g., [7, 11, 22]. However, these data structures only allow for atomic updates for a single operation.

5 Applications

The augmented map abstraction along with the PAM library can be used in many different applications. Not only does
Table 2. The running time of 2D range queries - “Seq. “, “Par. “ and “Spd.” refer to the sequential, parallel running time and the speedup. Q-small and Q-large mean the query time for small and large query windows, respectively. PAM and PAM\* mean the sweepline algorithm and the range tree using PAM, respectively.

<table>
<thead>
<tr>
<th></th>
<th>Build, s</th>
<th>Q-small, µs</th>
<th>Q-large, ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAM[19]</td>
<td>244</td>
<td>7.3</td>
<td>33</td>
</tr>
<tr>
<td>PAM*[19]</td>
<td>201</td>
<td>3.2</td>
<td>64</td>
</tr>
<tr>
<td>Boost[1]</td>
<td>315</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CGAL[13]</td>
<td>526</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

References


Acknowledgments

This work was supported in part by NSF grants CCF-1408940, CCF-1533858, and CCF-1629444. We also thank Julian Shun and Laxman Dhulipala for their useful suggestions and help.

struct interval_map {
    using interval = pair<point , point >;

    struct entry {
        using K = point ;
        using V = point ;

        static bool comp(K a, K b) { return a < b ;}
        static A I() { return 0 ;}
        static A base(K k, V v) { return v ;}
        static A combine(A a, A b) {
            return (a > b) ? a : b ;}

        using amap = aug_map<entry>;
        amap m;

        interval_map(interval× A, size_t n) {
            m = amap(A,A+n);
        }

        bool stab(point p) {
            return (amap::aug_left(m,p) > p);}
    }

    Figure 3. The definition of interval maps using PAM in C++.