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.

CCS Concepts • **Theory of computation → Sorting and searching**; **Shared memory algorithms**; • **Computing methodologies → Shared memory algorithms**.

Keywords balanced tree, augmented map, parallel, concurrent, library, PAM, ordered set, ordered map

ACM Reference Format:
Yihan Sun and Guy Blelloch. 2019. Implementing Parallel and Concurrent Tree Structures. In *24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '19), February 16–20, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293883.3302576

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02.
https://doi.org/10.1145/3293883.3302576

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 a 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. 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.

```
Split(T, k) =
  if T = ∅ then (∅,False,∅)
  else if k = k(T) then (l(T), True, r(T))
    else if k < m then let (L′, b, R′)=Split(l(T), k)
      in (L′,b,Join(R′, e(T), r(T)))
    else let (L′, b, R′)=Split(r(T), k)
      in (Join(l(T), e(T), L′), b, r(T))
Union(T1, T2) =
  if T1 = ∅ then T2
  else if T2 = ∅ then T1
  else let ⟨L1, v′, R2⟩ = Split(T1, k(T2))
          and L = Union(L1, l(T2)) || R = Union(R1, r(T2))
      in Join(L, e(T2), R)
Insert(T, k, v) =
  if T = ∅ then Singleton(k, v)
  else if k < k(T) then Join(Insert(l(T), k, v), e(T), r(T))
    else if k > k(T) then Join(L, e(T), Insert(r(T), k, v))
    else T
MapReduce(T, g, f, I) =
  if T = ∅ then I
  else let L = MapReduce(l(T), g, f, I)
          || R = MapReduce(r(T), g, f, I)
      in f(L, f(g(e(T)), R))
Build'(S, i, j) =
  if i = j then ∅
  else if i + 1 = j then Singleton(S[i])
  else let m = (i + j)/2
          and L = Build'(S, i, m) || R = Build'(S, m + 1, j)
      in Join(L, S[m], R)
Build(S) =
  Build'(RemoveDuplicates(Sort(S)), 0, |S|)
```

Figure 1. Example of algorithms using JOIN.

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

	n	m	T_1	T_{144}	Speedup
Union	10^8	10^8	12.517	0.2369	52.8
Find	10^8	10^8	113.941	1.1923	95.6
Insert	10^8	–	205.970	–	–
Build	$\mathbf{10^{10}}$	–	**1844.38**	**28.24**	**65.3**
Range	10^8	10^8	44.995	0.8033	56.0

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}).

is an abstract data type (ADT) based on ordered maps. The purpose of augmented maps is to support quick range-based abstract sums on ordered maps. For an ordered map with key type K, ordering on K defined by $<_K$ and value type V, we associate a map-reduce operation on it to define the abstract sum (of type A).

- The map operation. We define a *base function* $g : K \times V \mapsto A$ that maps an entry to an augmented value.
- A reduce monoid (A, f, I).
  - The set A that defines the *augmented value type*.
  - The associative function $f : A \times A \mapsto A$ that combines two augmented values. We call f the *combine function* of the augmented map.
  - The identity $I \in A$ of function f.

We also define the augmented value of such an augmented map $M = \{e_1, e_2, \ldots, e_n\}$ as $A(M) = f(g(e_1), g(e_2), \ldots, g(e_n))$. Here we extend binary operation f to multiple operands as $f(a_1, a_2, \ldots, a_n) = f(a_1, f(a_2, \ldots, a_n))$.

To support such an ADT, one efficient implementation is to use augmented trees. In fact, the above interface also defines a framework for a class of augmented trees. When a tree structure is used to maintain the ordered map, we store the abstract sum (augmented value) of the corresponding subtree in each tree node.

The Join-based algorithms also easily support augmentation. The update of an augmented value only occur when a node is involved in rotations, and thus only occur in Join. As a result, the augmentation can be dealt with by just Join, and all the other algorithms can be oblivious to the augmentation.

Augmented map is designed to support efficient range-based partial sum queries. For example, the AugLeft(M, k) function returns the augmented value for all entries in an augmented map M up to a key k. Using augmented trees, this function can be computed in $O(\log n)$ calls to the base and combine functions. More functions on the augmented interface and the algorithms can be found in [19, 20].

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.

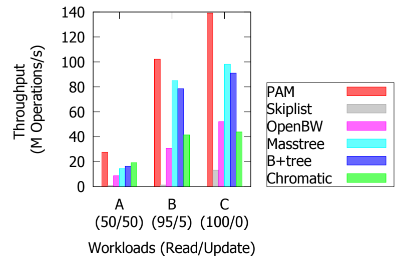

Figure 2. Comparing state-of-the-art concurrent data structures with PAM using batching.

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 5×10^7 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.

5 Applications

The augmented map abstraction along with the PAM library can be used in many different applications. Not only does

	Build, s			Q-small, μs			Q-large, ms		
	Seq.	Par.	Spd.	Seq.	Par.	Spd.	Seq.	Par.	Spd.
PAM[19]	244	7.3	33	11	0.13	85	213	2.0	108
PAM*[19]	201	3.2	64	17	0.21	81	45	0.7	65
Boost[1]	315	-	-	25	0.51	50	1174	22.4	52
CGAL[13]	526	-	-	154	-	-	111	-	-

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.

it give efficient implementation for each of the applications, the solutions are also concise. For example, previous work has shown examples of 1D interval stabbing query, 2D range query, 2D segment query, 2D rectangle query, inverted index searching using PAM. All of them only need around 100 lines of C++ code for a parallel version. A toy example of the implementation of 1D interval stabbing query is show in Figure 3, which is almost all the C++ code we need. Using the augmented tree implementation in PAM, this code effectively leads to a standard interval tree [9] structure. We note that typically implementing such an interval tree, even a sequential version would need hundreds of lines of code, while using PAM, we only need around 20 lines. We show the result of 2D range query using PAM (a range tree and a sweepline algorithm) comparing with two existing sequential libraries: CGAL [13] and Boost [1]. Sequentially, PAM outperforms both libraries. The self-speedup of PAM is 50-108.

```
struct interval_map {
  using interval = pair<point, point>;
  struct entry {
    using K = point;
    using V = point;
    using A = 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++.

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.

References

[1] 2015. Boost C++ Libraries. http://www.boost.org/. (2015).
[2] Stephen Adams. 1992. *Implementing Sets Effciently in a Functional Language.* Technical Report CSTR 92-10. University of Southampton.
[3] Kunal Agrawal, Jeremy T Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. 2014. Provably good scheduling for parallel programs that use data structures through implicit batching. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures.* ACM, 84–95.
[4] Naama Ben-David, Guy Blelloch, Yihan Sun, and Yuanhao Wei. 2018. Efficient Single Writer Concurrency. In *arXiv preprint arXiv:1803.08617.*
[5] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (1983).
[6] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA).* 253–264.
[7] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-blocking Trees. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).*
[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. ACM Symposium on Cloud Computing (SoCC).*
[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition).* MIT Press.
[10] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA).* 355–364.
[11] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proc. of the 7th ACM European Conference on Computer Systems.* ACM, 183–196.
[12] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD).*
[13] Gabriele Neyer. 2017. dD Range and Segment Trees. In *CGAL User and Reference Manual* (4.10 ed.). CGAL Editorial Board. http://doc.cgal.org/4.10/Manual/packages.html
[14] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.
[15] Colin Reid, Philip A Bernstein, Ming Wu, and Xinhao Yuan. 2011. Optimistic concurrency control by melding trees. *Proceedings of the VLDB Endowment* 4, 11 (2011).
[16] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endowment* 4, 11 (2011), 795–806.
[17] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting binary search trees. *Journal of the ACM (JACM)* 32, 3 (1985), 652–686.
[18] Yihan Sun, Guy Blelloch, and Daniel Ferizovic. 2018. The PAM library. (2018). https://github.com/cmuparlay/PAM
[19] Yihan Sun and Guy E Blelloch. 2019. Parallel Range and Segment Queries with Augmented Maps. *Algorithm Engineering and Experiments (ALENEX)* (2019).
[20] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: Parallel Augmented Maps. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP).*
[21] Robert Endre Tarjan. 1983. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
[22] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a Bw-tree takes more than just buzz words. In *International Conference on Management of Data (SIGMOD).*