PaC-trees: Supporting Parallel and Compressed Purely-Functional Collections

Laxman Dhulipala
University of Maryland
laxman@umd.edu

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

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

Abstract

Many modern programming languages are shifting toward a functional style for collection interfaces such as sets, maps, and sequences. Functional interfaces offer many advantages, including being safe for parallelism and providing simple and lightweight snapshots. However, existing high-performance functional interfaces such as PAM, which are based on balanced purely-functional trees, incur large space overheads for large-scale data analysis due to storing every element in a separate node in a tree.

This paper presents PaC-trees, a purely-functional data structure supporting functional interfaces for sets, maps, and sequences that provides a significant reduction in space over existing approaches. A PaC-tree is a balanced binary search tree which blocks the leaves and compresses the blocks using arrays. We provide novel techniques for compressing and uncompressing the blocks which yield practical parallel functional algorithms for a broad set of operations on PaC-trees such as union, intersection, filter, reduction, and range queries which are both theoretically and practically efficient.

Using PaC-trees we designed CPAM, a C++ library that implements the full functionality of PAM, while offering significant extra functionality for compression. CPAM consistently matches or outperforms PAM on a set of microbenchmarks on sets, maps, and sequences while using about a quarter of the space. On applications including inverted indices, 2D range queries, and 1D interval queries, CPAM is competitive with or faster than PAM, while using 2.1–7.8x less space. For static and streaming graph processing, CPAM offers 1.6x faster batch updates while using 1.3–2.6x less space than the state-of-the-art graph processing system Aspen.

ACM Reference Format:
Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-trees: Supporting Parallel and Compressed Purely-Functional Collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3519939.3523733

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523733

1 Introduction

Almost all modern programming languages include extensive support for collections, such as sets, maps, and sequences either as libraries or built-in data types. Support for such collections has become the cornerstone of large-scale data processing, as exemplified by systems such as Apache Spark [47]. Among the interfaces for collections, there has been a trend towards a functional style, shying away from mutation (e.g., Spark is functional). Functional interfaces have several advantages over mutating ones, including being safe for parallelism, allowing safe composition, permitting flexible implementations (e.g., using copies when helpful), and supporting snapshots. Supporting snapshots is particularly useful in scenarios in which a stream of updates is being made to a collection which is concurrently being analyzed [19, 21, 32, 35].

Recent work [45] has developed a purely functional library, PAM, for representing sequences, ordered sets, ordered maps, and augmented maps (defined in [45]) using balanced trees, called *P-trees*. P-trees use path copying to perform updates, supporting functional updates at a reasonably low cost (e.g., $O(\log n)$ per point update). However they come at a cost of high space usage—every element requires a node in the tree. This is particularly problematic for large-scale data analysis, since in large-systems memory is often the dominating cost.

In this paper we present ***Parallel Compressed trees (PaC-trees)***: a purely-functional data structure for supporting a similar functionality as P-trees but with significant reduction in space—up to an order of magnitude (see Fig. 1). Our approach is based on blocking the leaves and compressing the blocks using arrays (see Fig. 4). We present innovative techniques for compressing and uncompressing the blocks without needing to re-implement the full functionality of P-trees. Importantly, in the paper we analyze the cost of all the operations as a function of the block size B as well as the collection size. This is analyzed both in terms of the work (runtime sequentially) and span (longest dependent path in parallel). The costs for a sample of the supported functions are given in Table 1. These costs can help the user decide on a block size for their particular application—a parameter that can be specified when creating a collection.

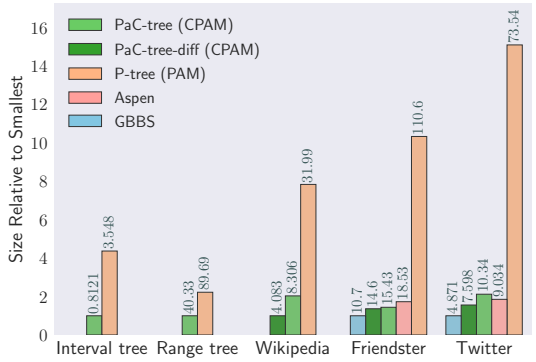

Figure 1. Relative sizes of the interval tree, range tree, inverted index (Wikipedia corpus), and graph representations (Twitter, Friendster) studied in this paper using PaC-trees from CPAM (using $B = 128$) and other systems. Lower is better. The numbers shown on top of the bars are the sizes of each representation in GiB. PaC-tree-diff compresses integer keys using difference encoding. The C-trees from Aspen [21] also support difference encoding. GBBS is the *static* compressed graph representation from the Graph Based Benchmark Suite [22] which uses difference encoding, and serves as a baseline for the tree-based graph representations.

	Primitive	Work	Span
Sequence	*Build*	$O(n)$	$O(\log n)$
	Map	$O(n)$	$O(\log n)$
	Filter	$O(n)$	$O(\log n)$
	Reduce	$O(n)$	$O(\log n)$
	Take	$O(\log n + B)$	$O(\log n)$
	n-th	$O(\log n + B)$	$O(\log n)$
	FindFirst	$O(k)$	$O(\log n)$
	Append†	$O(\log n + B)$	$O(\log n)$
	Reverse†	$O(n)$	$O(\log n)$
Set and Map	*Build*	$O(n \log n)$	$O(\log n)$
	Next/Previous	$O(\log n + B)$	$O(\log n)$
	Rank	$O(\log n + B)$	$O(\log n)$
	Range	$O(\log n + B)$	$O(\log n)$
	Insert	$O(\log n + B)$	$O(\log n)$
	Union	$O(m \log \frac{n}{m} + \min(mB, n))$	$O(\log n \log m)$
	Intersect	$O(m \log \frac{n}{m} + \min(mB, n))$	$O(\log n \log m)$
	Difference	$O(m \log \frac{n}{m} + \min(mB, n))$	$O(\log n \log m)$

Table 1. Primitives from the Sequence, Set, and Map interfaces in CPAM, including the work and span bounds. Note that primitives marked with † are specific to Sequences, and Set and Map primitives cannot be applied to Sequences. m, n are defined to be the size of the smaller and larger sets, respectively. B is the block size (the size of a blocked leaf in a PaC-tree). We assume a parallelizable encoding for the span bounds.

Using PaC-trees we have implemented CPAM: a C++ library which implements the full functionality of PAM, along with significant extra functionality involving compression. By default CPAM supports difference (or delta) encoding [36] within the blocked leaves. In such an encoding, each element is encoded based on the value of the previous element in the collection. This can greatly reduce space when elements that are close in the ordering of the collection are related.

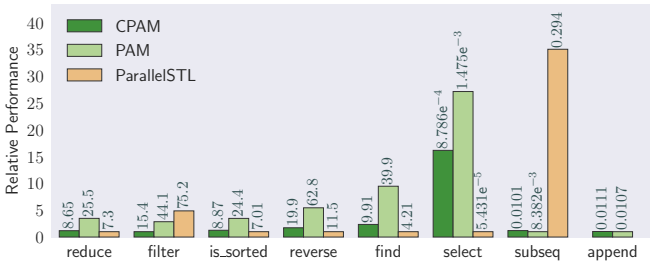

Figure 2. Relative performance of sequence primitives in CPAM (using $B = 128$), PAM, and ParallelSTL [29] on a 72-core machine with 2-way hyper-threading enabled. The numbers shown on top of the bars are the parallel (144-thread) running times in milliseconds. Lower is better. All benchmarks are run on sequences of length 10^8 containing 8-byte elements. For append, ParallelSTL takes 17.7 milliseconds on average (1594x larger than append in CPAM). CPAM and PAM represent sequences using purely-functional trees, whereas ParallelSTL uses arrays (hence static).

For example, if a graph is numbered so that neighboring vertices have similar indices, then the neighbors in a neighbor list will have small differences. These small numbers can then be encoded in a handful of bits each [42]. Similarly in an inverted index where each word points to a sequence of documents it appears in, if the documents are sorted, the differences between adjacent document identifiers can be small. This is especially true for common words, which take up the bulk of the space. In the paper we bound the extra space needed (due to the index using the tree structure) for PaC-trees compared to a static representation of the data (i.e., an array) directly using difference encoding (see Theorem 4.2).

In our default blocked representation, the first element of a block is represented uncompressed, and the rest of the elements are compressed relative to the previous element. In addition to delta-encoding, CPAM also supplies an interface for the user to define their own form of compression for each block. For example, they can quantize values, or use other variable length codes when keys are known to be small. CPAM uses a reference counting garbage collector to manage the memory for both the internal nodes and the compressed leaf nodes, which can be of variable size due to compression.

CPAM supports augmentation in which each tree node maintains an aggregate of the values of its subtree (see more details in Section 3). The aggregation function is declared as part of the type of the tree. Augmentation is useful in many applications, and indeed we use it in all of the applications we describe later. PaC-trees store an augmented value per internal node, and one for each block at the leaves. Storing one value per block significantly reduces space relative to P-trees in PAM, which store a value for every element.

To demonstrate the effectiveness of PaC-trees, and their implementation in CPAM, we measure performance and space usage on (1) a collection of microbenchmarks that directly use some of the functions supported by the library, and (2) a handful of real-world applications.

For the microbenchmarks, we compare the performance of CPAM to PAM, and for sequences to the Intel implementation of the C++17 parallel STL library [29] (ParallelSTL). ParallelSTL is a highly optimized library supporting only sequences based on arrays. A summary of the results for sequences is given Fig. 2, and details including performance of ordered maps, and augmented maps are given in Section 9. Compared to PAM, CPAM achieves significantly better performance due to the reduced memory footprint, and hence reduced number of cache misses, while only requiring about 1/4-th as much space even without compression. Compared to ParallelSTL, CPAM has similar performance on operations that visit the whole sequence, like `reduce`, but is significantly slower on `nth` since it requires $O(\log n + B)$ work as opposed to $O(1)$ for a random array access for ParallelSTL. On `append` CPAM is significantly faster since it requires $O(\log n + B)$ work to join two trees instead of $O(n)$ required by ParallelSTL to copy the input arrays into the output array.

We consider four applications: graphs, inverted indices, 2D range queries and 1D interval queries. For inverted indices, 2D range query and 1D interval query, CPAM achieves competitive performance to PAM while using 2.1x–7.8x less space. For graph processing, we compare to an existing system Aspen [21] that represents graphs using trees. CPAM uses 1.3–2.6x less space compared to Aspen, and is almost always faster than Aspen in all tested graph algorithms.

The main contributions of this paper are:

- A new functional data structure, PaC-trees, and associated parallel algorithms that support compression for sequences, sets, maps and augmented maps.
- Theoretical bounds on the costs (work and span) and the space of the data structure and associated algorithms.
- An implementation of PaC-trees as a library, CPAM, supporting the full functionality of PAM in addition to supporting default and user defined compression schemes.[1]
- An experimental evaluation of the ideas and implementation on microbenchmarks and non-trivial applications.

2 Related Work

Our work extends P-trees and their C++ implementation in PAM [45]. Our key contribution is the ability to compress the trees achieving up to an order-of-magnitude reduction in space. This is achieved while being able to present cost bounds both in terms of time and space. These bounds are a function of a block size the user can select.

B-trees [6] and their variants block not just the leaves but all nodes of a tree, such that internal nodes can have a high fan-out. They are widely used in practice, especially for disk based data structures since nodes are on the scale of a page on disk and can be retrieved efficiently. However they are less relevant in the context of purely functional in-memory trees. In particular, path copying requires that an update copy all nodes on the path from the root to the leaf. If the nodes are large (e.g. 128+ elements each, as in our leaves) this copying would be very expensive both in terms of space and time. Various work has suggested blocking the leaves of a binary tree to represent sequences [1, 8, 15, 26, 33]. The idea is to reduce the cost of operations such as append or subsequence relative to array representations. As far as we know, these ideas have never been applied to ordered sets or ordered maps.[2] We also do not know of work that then compresses within the blocks.

Aspen [21] is a system for graph processing, based on purely functional trees and uses compression for the neighbor lists. At a high-level, our goals are shared with Aspen (e.g., non-mutating updates), but Aspen has several limitations. Importantly it is only designed for graphs, supporting only a small part of the functionality of CPAM. The tree representation in Aspen is also very different. It randomly selects elements from the collection to be *heads*. It then attaches a block of nodes to each head corresponding to the keys between the head and the next head, and puts the heads into a binary tree. PaC-trees do not require randomization, and have stronger theoretical bounds for primitive operations such as `union` than the bounds provided by C-trees in Aspen. We use CPAM to implement the full functionality of Aspen and compare to Aspen in Section 9.4.

Fig. 3 compares P-trees from PAM, functional B-trees, C-trees from Aspen, and PaC-trees. The comparison illustrates how they differ when inserting a new key.

Like CPAM, the Apache Spark [47] system supports a functional interface for collections. However it has several significant differences. First, it only supports unordered sets. Second, although it has a shared-memory parallel implementation, it is primarily designed for a distributed setting. This means its shared-memory implementation is not ideal.[3]

There is extensive research on concurrent tree data structures [3, 4, 16, 18, 24, 34, 37]. This work is mostly orthogonal to our work. Such trees support a fraction of the functionality of CPAM, typically just supporting linearizable inserts, deletes, updates and finds. Some recent works support range queries [5, 25], or arbitrary queries on a snapshot [46]. On the other hand concurrent trees support asynchronous updates, which PaC-trees do not—such updates are inherently non-functional. To support multiple concurrent updates, PaC-trees would require batching the update and applying them as a batch in parallel (fairly comparing concurrent and batched structures like PaC-tree seems challenging for this reason). We expect the use cases would be quite different.

[1] We have made CPAM publicly available: https://github.com/ParAlg/CPAM.

[2] We note that the design of the chunked sequence datatype [1] could in principle be extended to support sets, maps, and augmented maps, although the implementation is specialized for ephemeral sequences.

[3] Their shared-memory implementation is between 3.2–4.9x slower than CPAM for a map, reduce, and group-by style example taken from their user guide. For primitives such as map and reduce, their implementation performs up to 2 orders of magnitude worse than CPAM (see the full version).

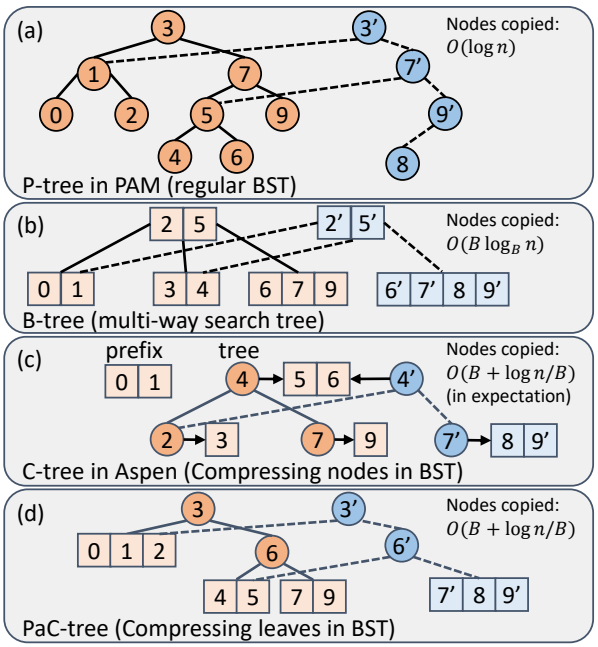

Figure 3. An illustration of (a) P-tree in PAM [11, 45] (regular BST), (b) B-tree (multi-way search tree), (c) C-tree [21] in Aspen (compressing all nodes in a BST) and (d) our PaC-tree (compressing all leaves in a BST) in CPAM. The orange nodes show a tree with keys 0-7 and 9. We then consider inserting a key 8. Blue nodes are what we need to create (copy or new) due to path-copying. Round nodes are tree nodes each storing a single key, and square nodes are organized in blocks of size $O(B)$ (expected for C-trees). Letting n be the tree size, an insertion needs to copy $O(\log n)$ nodes in P-tree, $O(B \log_B n)$ in B-tree, and $O(B + \log(n/B))$ in C-tree (in expectation) on a PaC-tree.

Blandford and Blelloch developed tree structures for ordered sets that support compression [9]. They present space bounds that are similar to ours, in terms of relating the space of a difference encoded sequence to the space of the data structure. However they support a small fraction of the functionality described in our work.

Functional trees using path-copying date back to at least the early 1990s [2], and in the sequential setting have been studied by Kaplan and Tarjan [31] and Okasaki [40].

3 Preliminaries

Binary search trees. A *binary search tree* (BST) is either an empty node, denoted as *nil*, or a node consisting of a *left* BST T_L, a key k (or with an associated value), and a *right* BST T_R, denoted $\mathsf{node}(T_L, k, T_R)$, where k is larger than all keys in T_L and smaller than all keys in T_R. We use $lc(T)$ and $rc(T)$ to extract the left and right subtrees of T, respectively, and use $k(T)$ to denote the key stored at T's root. The *size* of a BST T, or $|T|$, is the number of nodes in T. The *weight* of a BST T, or $w(T)$, is $1 + |T|$. The *height* of a BST T, or $h(T)$, is 0 for *nil*, and $\max(h(lc(T)), h(rc(T))) + 1$ otherwise. A tree node is a leaf if it has no children, and a regular node otherwise. The *left (right) spine* of a binary tree is the path of nodes from the root to a *nil* node, always following the left (right) tree.

A *weight-balanced tree*, or BB[α] trees [39] is a BST where for every $T = \mathsf{node}(T_L, v, T_R)$, $\alpha \leq \frac{w(T_L)}{w(T)} \leq 1 - \alpha$. We omit the parameter α with clear context. A weight-balanced tree T has height at most $\log_{\frac{1}{1-\alpha}} w(T)$.

Parallelism. Our implementation of PaC-trees is based on nested fork-join parallelism [20, 27, 30]. We analyze our algorithms use work-span model based on binary-forking [12]. The *work* W of a parallel algorithm is the total number of operations, while the *span* is the critical path length of its computational DAG. We use $s_1 \parallel s_2$ to indicate that statements s_1 and s_2 can run in parallel. Almost all algorithms use divide-and-conquer to enable parallelism. Any computation with W work and S span will run in time $T < \frac{W}{P} + S$ on P processors assuming shared memory and a greedy scheduler [14, 17]. We use $\log n$ to denote $\log_2(n+1)$ in the cost bounds.

Encoding schemes. We use ***Difference Encoding*** (DE) to encode integer keys. Given a sorted set of keys, K, the difference encoding scheme stores the differences between consecutive keys using an integer code, such as byte or γ codes. We only consider byte codes in this paper since they are cheap to encode and decode and do not waste much space compared to using γ codes [42].

Functional data structures. PaC-trees are purely functional data structures. In functional data structures values are immutable, so updates must be made by copying parts of the structure. For search trees, only the path to the update location needs to be copied. Hence for balanced trees of size n, single point updates such as inserts and deletes involve copying $O(\log n)$ nodes (Fig. 3(a)). This also applies to multi-point updates. For example, if a `filter` ends up removing a single element, only $O(\log n)$ nodes need to be copied. Functional trees can also easily support multiversioning with low time and space overhead [7, 44]. Because the data are immutable, any operation accesses the tree in an isolated version. Updates can be applied in *batches* in parallel and yield a new version. This enables all read-only queries to be performed at the same time without being affected by ongoing (concurrent) updates. In addition to multiversioning, functional data structures also allow for multiple histories.

Join-based algorithms. PaC-trees are implemented using the *join-based* approach [11, 13, 28, 43–45] first implemented in PAM [45]. In the framework, a variety of tree algorithms are implemented based on two primitives, `join` and `expose`.[4] Given a balancing scheme $\mathcal{S}$, the $\mathsf{join}(T_L, e, T_R)$ function returns a balanced tree T satisfying $\mathcal{S}$ which has the same in-order values as $\mathsf{node}(T_L, e, T_R)$. In other words, it concatenates T_L and T_R by an entry e in the middle while preserving

[4] PAM did not explicitly use expose as a primitive, but only conceptually treated it as a primitive.

the balancing invariants (see Fig. 7 as an example of joining two PaC-trees). The expose(T_L) function returns a triple (T_L, e, T_R), where $e \in T$ is an entry, T_L and T_R are two binary trees such that both T_L and T_R satisfy $\mathcal{S}$, are balanced with each other under $\mathcal{S}$, and T_L (T_R) contains all keys in T that go before (after) e in T's in-order value. It has been shown that on weight-balance trees with $\alpha \leq 1 - 1/\sqrt{2}$, a join operation can be done in $O(\log \frac{n}{m})$ work [11], where $n = \max(|T_L|, |T_R|)$ and $m = \min(|T_L|, |T_R|)$.

Based on join and expose, many parallel tree algorithms can be expressed in a simple and elegant recursive style (see Fig. 5 and Fig. 6 for examples). We adopt the join-based approach in our implementation, and in particular carefully designed join and expose functions for PaC-trees. This greatly simplifies the implementation and correctness arguments of our algorithms. We give more details in Sections 5 and 6.

Augmentation. An *augmented tree* is a search tree where each node maintains an aggregated value (called ***augmented values***) of all entries in its subtree. Typical examples would be a weighted sum, minimum or maximum of values, where we can obtain the augmented value in a node by combining augmented values of the children and itself. This generalizes to all associative operations. PaC-trees support generic user-defined augmentation for any associative operations. An example of PaC-tree with augmentation is shown in Fig. 4.

4 PaC-Trees

In this paper, we propose PaC-trees to support purely functional collections, which support *parallelism, determinism, compression, augmentation, strong theoretical bounds, and multi-versioning*. PaC-trees are purely functional. The base data structure of a PaC-tree is a weight-balanced BST. The internal nodes remain binary so they are cheap to copy. The leaves in a PaC-tree are organized in blocks of size B to $2B$ for some parameter B. An illustration is shown in Fig. 3. If the blocks grow too large, they are split, and if they become too small they are merged with a neighboring node.

Definition 4.1 (PaC-tree)**.** A PaC-tree $PaC(\alpha, B, C)$, parameterized by the balancing factor α, block size B, and encoding scheme C satisfies the following invariants:

- (**Weight Balance**) For any tree node $v \in T$, $\alpha \leq \frac{w(v_*)}{w(v)} \leq 1 - \alpha$, where $\alpha \leq 1 - \frac{1}{\sqrt{2}}$ is a constant, and v_* is either $lc(v)$ or $rc(v)$. Unless mentioned otherwise, we use $\alpha = 0.29$.
- (**Blocked Leaves**) If $|T| \geq B$, each leaf $u \in T$ maintains B to $2B$ entries in an array (called a ***block***) using the encoding scheme C. Unless mentioned otherwise, we assume C is empty, which means the entries are blocked without additional compression of the entries.

When the context is clear, we omit α, B and C in the definition and simply call it a ***PaC-tree***. We call a leaf node containing multiple entries in a PaC-tree a ***flat node***, and a node containing a single entry a ***regular node***. We say a

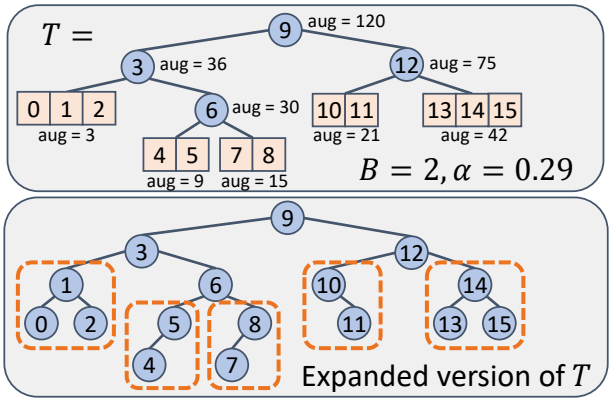

Figure 4. (a). An illustration of a PaC-tree with keys $\{0, 1, \ldots, 15\}$, and augmentation as sum of keys. All nodes are weight-balanced. All leaves are blocked as arrays of size B to $2B$. (b) The expanded version of the PaC-tree in (a).

PaC-tree (or a subtree) T is a ***simplex*** tree if $|T| < B$, and thus T only contains regular nodes. We say a PaC-tree (or a subtree) T is a ***complex*** tree if T contains both regular nodes and flat nodes. We define the ***expanded*** version of a PaC-tree T (or a flat node v) to be a regular binary tree (without flat nodes), where all flat nodes in T (or v itself) are fully expanded as perfectly-balanced binary trees. In Fig. 4, we show an example of an expanded tree.

We now present the space bound of a PaC-tree. For integer keys, we can use difference encoding to bound the space.

Theorem 4.2. *The total space of a PaC-tree* $\text{PaC}(\alpha, B, C_{DE})$ *maintaining a set E of integer keys is $s(E) + O(|E|/B + B)$, where C_{DE} is difference encoding, and $s(E)$ is the size needed for E using difference encoding.*

Proof. The space needed for a PaC-tree includes the regular nodes and the leaf nodes. First of all, when $|E| < B$, all entries are maintained in a simplex tree, taking $O(B)$ space. When $|E| \geq B$, there are $O(|E|/B)$ regular nodes, each taking $O(1)$ space for meta-data (pointers, size, etc.). The total space used by regular nodes is $O(|E|/B)$. All the leaf nodes are organized in blocks. Let A be an array that stores all keys in E using difference encoding. Comparing the total size of all the blocks and A, the only extra space is the first element of each block (which cannot be compressed). There are $O(|E|/B)$ such blocks, and thus the extra space used is $O(|E|/B)$. □

We note that this bound is deterministic, as opposed to the bound for C-trees (which only holds in expectation). Furthermore, using known facts about difference encoding yields the following result, showing that PaC-trees yield a compact parallel representation of ordered sets [9].

Corollary 4.3. *Given any set from $U = \{0, \ldots, m-1\}$ with $|S| = n$, the total space of a PaC-tree* $\text{PaC}(\alpha, B, C_{DE})$ *maintaining S is $O(n \log \frac{n+m}{n})$ bits for $B = \Omega(\log n)$.*

5 Algorithms

We now describe join-based algorithms on PaC-trees. To enable a general ordered map interface, we implement PaC-trees based on the PAM interface. PAM supports dozens of operations on sequences, sets, maps, and augmented maps, and it would require significant work to re-implement them all. Instead, we carefully redesigned join and expose such that *all the other algorithms can remain the same as in PAM*. In particular, none of the other algorithms have to deal with the blocked leaves or compression, which greatly simplifies the algorithm design and correctness arguments. We found that the overhead of this approach is not large, but for many frequently-used operations, we design special base cases for dealing with compressed nodes. These base cases can improve the performance by up to 6x (see Section 7). Some of the theoretical results also require special base cases (see **??**).

At a high-level, when exposing a flat node, the node is automatically expanded (using unfold), and similarly when join obtains a complex tree of size B to $2B$, it is flattened (fold). An illustration of unfold and fold is shown in Fig. 7. We start with the join and expose algorithms. We then present the union algorithm as an example to illustrate join-based algorithms, and give the code for other functions in Fig. 6 and the full version of this paper. We focus on union as it is the core sub-routine used in applications such as inserting or deleting batches of vertices and edges in graphs, combining inner trees when constructing range trees, and updating sets of documents in an inverted index, among others.

Expose. This function returns the left subtree, root data and the right subtree of a node T. For a regular node, this function just reads the child pointers and the root. For a flat node, this function first unfolds the tree into a perfectly balanced tree and then reads the corresponding data.

Join. Recall that the join function takes two trees T_L and T_R, and a key k (or a key-value) as input, and returns a balanced tree concatenating entries in T_L, k and T_R in order (see Fig. 7). In other words, when trees are used for ordered sets or maps, k should be larger than all keys in T_L and smaller than all keys in T_R. Pseudocode for join is shown in Fig. 5.

The algorithm first compares the weights of T_L and T_R. When balanced, they are directly connected by k. The other two cases are symmetric so WLOG we assume $|T_L| > |T_R|$. In this case, the algorithm must attach T_R in the right spine of T_L, which will be handled by $\texttt{join\_right}(T_L, k, T_R)$. This algorithm first checks if T_L and T_R are balanced and connects them if so. Otherwise, it recursively calls join_right on $rc(T_L)$ and T_R, getting T'. If we re-attach T' as T_L's right child, we will get a "correct" output tree (modulo balance). We then use a single or double rotation to rebalance if necessary. It is known that either a single or double rotation can rebalance a weight-balanced tree in this situation [11]. This guarantees the *weight balance* invariant of PaC-trees.

To also guarantee the *blocked leaves* invariant, we add two conditions when calling node to create a new node with its left and right subtrees. Whenever a node with size B to $2B$ is created, we fold the tree into a flat node. Whenever a node with size $2B$ to $4B$ is created, we extract the median of the tree as the root to re-distribute its two subtrees, such that both subtrees are flat nodes with (almost) the same size.

Lemma 5.1. *The* join *function maintains the invariants of PaC-trees.*

Split. For a PaC-tree T and key k, $\texttt{split}(T, k)$ returns a triple (T_L, b, T_R), where T_L (T_R) is a tree containing all keys in T that are less (greater) than k, and b the entry of key k if $k \in T$ (see Fig. 7). We first use $\texttt{expose}(T)$ to get its left (right) subtrees $lc(T)$ ($rc(T)$) and root key $k(T)$, and compare k with $k(T)$. If $k = k(T)$, we simply return $(lc(T), k, rc(T))$. Otherwise WLOG we assume k is smaller. In that case, the entire right subtree $rc(T)$ and the root $k(T)$ belong to T_R. We then split $lc(T)$ by k, getting (L_L, b, L_R). By definition, all keys smaller than k should be in L_L, and all keys larger than k can be obtained by $\texttt{join}(L_R, k(T), rc(T))$.

Union. Using join and split, we can implement set algorithms on two PaC-trees, such as union, intersection and difference. We describe union as an example (the other two are similar). This algorithm uses divide-and-conquer. At each level of recursion, T_1 is split by the root of T_2, breaking T_1 into two subsets with all keys smaller (larger) than $k(T_2)$, denoted as L_1 (R_1). Then two recursive calls to union are made in parallel. One unions $L(T_2)$ with L_1 (all keys smaller than $k(T_2)$), returning T_L, and the other one unions $R(T_2)$ with R_1 (all keys larger than $k(T_2)$), returning T_R. Finally the algorithm combines the results with $\texttt{join}(T_L, k(T_2), T_R)$.

Other algorithms. We show the pseudocode of other parallel algorithms in Fig. 6 and more in the full version of the paper. We omit the details as they are self-explanatory and all of them are exactly the same as in PAM, just by plugging in the new version of join and expose functions for PaC-tree. Almost all of them use divide-and-conquer to enable parallelism. We refer the reader to [45] for more details.

Importantly, all of our PaC-tree algorithms are theoretically efficient. We present the work-span bound in Table 1 and give a proof for union as an example in Section 6. Note that Lemma 5.1 ensures the correctness of the other algorithms, as their return values are always obtained by a join.

Theorem 5.2. *All* join*-based algorithms on PaC-tree maintains the invariants of PaC-trees.*

6 Theoretical Guarantees

In the following section we show work and span bounds for operations on PaC-trees. We assume the encoding scheme

```
1  fold(T) {
2    flatten T into array A
3    (encoding if needed)
4    return A; }
5  unfold(A) {
6    /* return a perfectly balanced tree
7    from sorted array A */ }
8  expose(T) {
9    if (isflat(T)) {
10     T' = unfold(T);
11     return (lc(T'), k(T'), rc(T')); }
12   else return (lc(T), k(T), rc(T)); }
13 join(T_L, k, T_R) {
14   if (heavy(T_L, T_R))
15     return join_right(T_L, k, T_R);
16   if (heavy(T_R, T_L))
17     return join_left(T_L, k, T_R);
18   return node(T_L, k, T_R); }
19 /* join_left is symmetric */
20 join_right(T_L, k, T_R) {
21   (l, k', c)=expose(T_L);
22   if (balance(|T_L|, |T_R|)
23     return node(T_L, k, T_R));
24   T' = join_right(c, k, T_R);
25   (l_1, k_1, r_1) = expose(T');
26   if (balance(|l|, |T'|))
27     return node(l, k', T');
28   if ((balanced(|l|, |l_1|)) and
29         (balanced(|l| + |l_1|, r_1)))
30     return rotateleft(node(l, k', T'));
31   else return rotateleft(node(l, k',
32          rotateright(T'))); }
33 join2(T_L, T_R) {
34   if (T_L = nil) return T_R;
35   (T'_L, m, _) = split(T_L, last(T_L));
36   return join(T'_L, m, T_R);  }
37 node(l, k, r) {
38   /* create node x with left subtree l,
39   root key k and right subtree r */
40   if (|x| > 4B) return x;
41   if (B ≤ |x| ≤ 2B) return fold(x);
42   else { // 2B < |x| ≤ 4B
43     /* redistribute x's both subtrees to
44     be flat nodes with |x|/2 entries */
45     return x;}}
46 split(T, k) {
47   if (|T| = 0) return (nil, nil, nil);
48   (L, m, R) = expose(T);
49   if (k == k(m)) return (L, m, R);
50   if (k < k(m)) {
51     (L_L, b, L_R) = split(L, k);
52     return (L_L, b, join(L_R, m, R));
53   } else {
54     (R_L, b, R_R) = split(R, k);
55     return (join(L, m, R_L), b, R_R); } }
```

Figure 5. Primitives on PaC-trees. All codes are functional (e.g. rotates copy nodes).

```
1  from_sorted(A,n) {
2    if (n = 0) return nil;
3    if (n = 1) return node(nil,A[0],nil);
4    L = from_sorted(A,n/2) ||
5      R = from_sorted(A+n/2,n-n/2);
6    return node(L,A[n/2],R); }
7  build(A,n) {
8    parallel_sort(A,n);
9    return from_sorted(A,n); }
10 union(T_1, T_2) {
11   if (T_1 == nil) return T_2;
12   if (T_2 == nil) return T_1;
13   (L_2, k_2, R_2) = expose(T_2);
14   (L_1, b, R_1) = split(T_1, k_2);
15   T_L = union(L_1, L_2) ||
16         T_R = union(R_1, R_2);
17   return join(T_L, k_2, T_R); }
18 // keep a key in T only when it satisfies f
19 filter(T, f) {
20   if (T == nil) return nil;
21   (L, k, R) = expose(T);
22   T_L = filter(L, f) ||
23         T_R = filter(R, f);
24   if (f(k))
25     return join(T_L, k, T_R);
26   else return join2(T_L, T_R); }
```

Figure 6. Examples of parallel algorithms on PaC-trees. "||" indicates calls that are made in parallel.

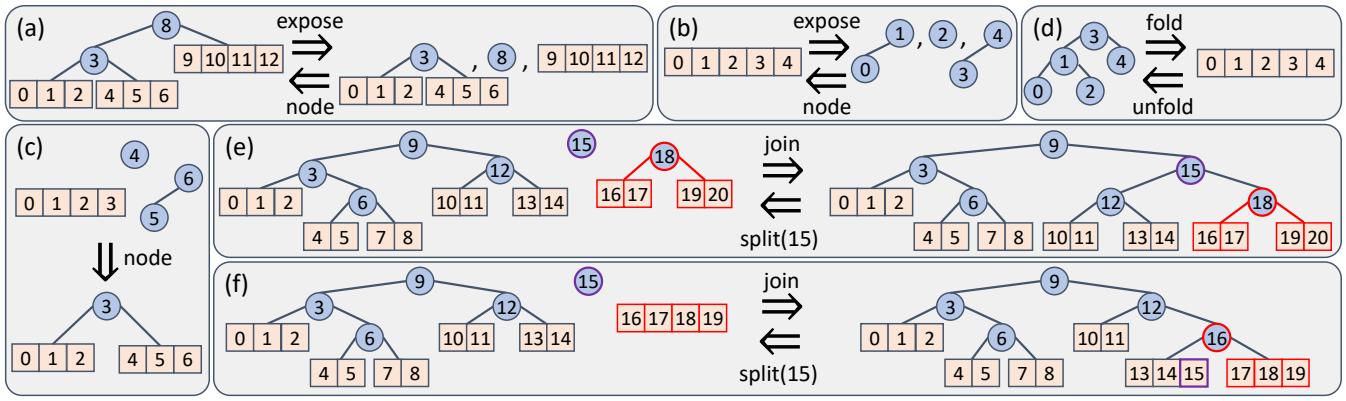

Figure 7. Illustration of primitives on PaC-trees. For Figures (a)–(d), $B = 3$. For Figures (e)–(f), $B = 2$. Fig. (a): the `expose` function on a regular node and the `node` function to obtain a regular node when the output tree size is larger than $4B$. Fig. (b): the `expose` function on a flat node and the `node` function to obtain a flat node when the output tree weight is between B and $2B$. Fig. (c): the `node` function to obtain a flat node when the output size is between $2B$ and $4B$. Fig. (d): `fold` and `unfold` functions. Fig. (e): `join` function on two regular nodes and its corresponding `split` function. Fig. (f): `join` function on a regular node and a flat node and its corresponding `split` function.

is empty, which means that to flatten or expand a block of size n costs $O(n)$ work and $O(\log n)$ span. If the encoding scheme is not parallelizable (e.g., for difference encoding), the span bound of the algorithms will be affected. We present more details in the full version of the paper.

We start with the cost of the `join` and `split` algorithms.

Theorem 6.1. *Consider a* join *algorithm on two PaC-trees T_L, T_R and an key k. Let $n = \max(|T_L|, |T_R|)$ and $m = \min(|T_L|, |T_R|)$. If both T_L and T_R are complex trees, the algorithm takes $O\left(\log \frac{n}{m}\right)$ work and span. If both T_L and T_R are simplex trees, the algorithm takes $O(B)$ work and $O(\log B)$ span. Otherwise, the algorithm takes $O(B + n/B)$ work and $O(\log n)$ span.*

Theorem 6.2. *Consider a* split *algorithm on a PaC-tree T. If T is a complex tree, the work and span of* split *are $O(\log \frac{|T|}{B} + B)$ and $O(\log |T|)$, respectively. If T is an simplex tree, the work and span of* split *is $O(\log |T|)$.*

Due to page limit, we provide the proofs of Theorems 6.1 and 6.2 in the full version of this paper. Based on these results, we now analyze the cost of the set operations.

Theorem 6.3. *Consider the* union *algorithm (and the closely related* intersection *and* difference *algorithms in the Appendix) in Fig. 5 on two PaC-trees of sizes m and $n \geq m$. The work and span for these algorithms are $O\left(m \log \frac{n}{m} + mB\right)$ and $O(\log n \log m)$ respectively.*

To prove the theorem, we first present some definitions and lemmas. First, note that all the work can be asymptotically bounded by the three categories below:

(1). ***split work***: all work done by split (Line 14),
(2). ***join work***: all work done by join (Line 17) or join2 in intersection and difference,
(3). ***expose work***: all work done by expose (Line 13).

One observation is that the split work is identical among the three set algorithms. This is because the three algorithms behave the same on the way down the recursion when doing splits, and only differ in what they do at the base case and on the way up the recursion when building the output tree (see the other two set algorithms in **??**).

We use op to denote the set operation (one of union, intersection or difference). In these algorithms, the tree T_1 is split by the keys in T_2. We call T_1 the ***decomposed tree*** and T_2 the ***pivot tree***, denoted as T_d and T_p respectively. Let $m = \min(|T_p|, |T_d|)$ and $n = \max(|T_p|, |T_d|)$.

Lemma 6.4. *For each function call to* op *on trees $P \subseteq T_p$ and $D \subseteq T_d$, the work done by* join *(or* join2*) is asymptotically bounded by the work done by* split.

We present the proof in the full version of this paper. Next, we prove the bounds for split work and expose work, respectively.

Lemma 6.5. *The expose work is $O(\min(mB, n))$.*

Proof. expose costs $\Theta(B)$ when the subtree is a flat node, and $O(1)$ otherwise. At most $O(m)$ nodes in T_p will split T_d, so the total cost is $O(mB)$. The cost is also no more than $O(n)$ since each node is involved in at most one expose, after which the flat node will be fulled expanded. In summary the cost is $O(\min(mB, n))$. □

Lemma 6.6. *The total* split *work is $O\left(m \log \frac{n}{m} + mB\right)$.*

Proof. The total split work can be viewed as two parts: the total work to done by split functions to traverse and split non-flat nodes, and the work to expose and split the flat nodes. Note that here "non-flat nodes" include both regular nodes in complex trees, and all the nodes in expanded trees.

First of all, the total work to traverse and split all non-flat nodes can be asymptotically bounded by the split work when both T_p and T_d are considered to be fully expanded. This cost is $O\left(m \log \frac{n}{m}\right)$ from the result for P-trees [11].

We then consider all work done by split functions on flat nodes. The only extra cost is the cost of unfold. Every node in T_p will be used at most once to split T_d, which involves at most one unfold function with cost $O(B)$. There can be at most $O(m)$ nodes in T_p used to split T_d. Thus the total unfold work in split is $O(mB)$.

Therefore in total the split work is $O\left(m \log \frac{n}{m} + mB\right)$. □

We can now prove Theorem 6.3.

Proof. (Theorem 6.3) Combining Lemmas 6.4 to 6.6 proves the work bound in Theorem 6.3. For the span, note that the algorithms need $O(\log |T_p|/B)$ rounds to reach a flat node, where the flat node will be expanded, taking $O(\log B)$ span. Then the algorithm keeps recursing until a *nil* node is reached, which takes $O(\log B)$ rounds. In each of the recursive calls, we need $O(\log |T_d|)$ span to deal with split and join. In total the span is $O(\log m \log n)$. □

Note that the $O(mB)$ term can be expensive when m is large. In fact, we can show a tighter bound using a more efficient (but more complicated) base case. We show the bound in Theorem 6.7, and defer the algorithm and proof to the full version. In our implementation, we use the version in Fig. 5, which has good performance in practice.

Theorem 6.7. *There exist algorithms for* union, intersection *and* difference *on two PaC-trees of sizes m and n ($n \geq m$) with work $O\left(m \log \frac{n}{m} + \min(n, mB)\right)$ and span $O(\log n \log m)$.*

7 Implementation

In this section, we describe CPAM, our implementation of PaC-trees. CPAM is built in C++, based on the PAM framework [45]. Our implementation of sequence and map primitives are mostly unchanged. Most of the changes are to introduce flat nodes, to handle folding and unfolding in join, to express the recursive functions using the expose primitive, and in some cases to add optimized base cases.

Optimized Base Cases. We first implemented union as in Fig. 5, which recursively calls expose to access the left and right subtrees. Although simple and theoretically efficient, in practice unfolding flat nodes into expanded trees and recursing on these trees requires additional memory allocations, and potentially more cache-misses. We therefore designed

a new sequential base-case for `union` when $|T_1| + |T_2| < \kappa$, where κ is a configurable *base-case granularity*. Our base-case works by writing both T_L and T_R into a pre-allocated array A of size κ and merging them in-place to perform the union. It then constructs a PaC-tree from the result in A. Compared to the original version of `union` that only uses `expose`, using the special base-case with $\kappa = 4B$ is 4.4x faster, and using $\kappa = 8B$ is 6.7x faster ($B = 128$). We observed similar improvements for some other commonly-used primitives such as `filter`, `map_reduce`, `multi_insert`, `multi_delete`, and `intersection`. We use $\kappa = 8B$ in our experiments. We use a parallel granularity of $4B$, which is the threshold for forking parallel tasks in algorithms such as `filter` and `union`.

Persistence and Memory Management. CPAM uses a reference counting garbage collector for memory management. CPAM provides functional ordered maps, and thus by default does not modify the input trees. However, in certain cases an application may wish to modify a tree in-place to save memory, e.g., when updates and queries are separated. Although one could deal with in-place and functional updates separately, this is not attractive. Instead, we designed a simple approach to handle both cases using the same code, which we describe in Appendix **??** due to space constraints.

Compression on Blocks. CPAM makes it easy to apply user-specified encoding schemes. Our data structure is templated over a type representing a block encoding scheme (no encoding by default). To add a new encoding scheme, users provide a structure with methods that calculate the encoded size for a block, encode the elements into a buffer, and decode elements from an encoded buffer. This design allows users to specify encoding schemes based on the underlying data type or application, such as text compression. For example, it is easy to add new types of difference coding, e.g., using γ-coding, which would obtain better space usage at the expense of worse running time [42].

8 Applications

In this section we describe four applications that we implement using CPAM. Our inverted index, and range and interval tree applications are based on the implementations from PAM [45]. Our graph processing application is based on Aspen [21]. We focus on the key features of the applications in the context of PaC-trees here.

Inverted Index. We implement a weighted inverted index, similar to those used in search engines. The inverted index maintains a top-level map from words to document lists ($B = 128$). Each document list is a map from document id to an importance score ($B = 128$). The document lists are augmented to maintain the highest importance score. The inverted index supports standard AND/OR queries over words, returning results by rank, and top-k (based on importance) queries. The document ids are compressed using difference encoding, requiring less than two bytes per document.

2D Range Tree. The two-dimensional range tree is a top-level map from x-coordinate to y-coordinate ($B = 128$). The tree is augmented so that every internal node stores all y-coordinates in its subtree (this is itself a set represented as a PaC-tree with $B = 16$). Updates can add and delete points, and queries can list of or count the points in a given rectangular range. The range tree supports count queries in $O(\log^2 n)$ time, which can be batched to run in parallel.

Interval Tree. The interval tree maintains intervals over the number line, for example, representing the time of a TCP connection, or the time a user is logged into some service. A stabbing query can report all or any intervals that cross a given point. The intervals are represented as an augmented tree from left-coordinate to right-coordinate with $B = 32$. The augmentation maintains the maximum right-coordinate in the subtree. This allows stabbing queries in time $O(k \log n)$ where k is the number of intervals requested or returned (whichever is less). Intervals can be inserted or deleted in $O(\log n)$ time and can be batched to run in parallel.

Graph Processing. Graphs are represented as a two-level structure similar to the inverted index, with a top-level augmented tree (the *vertex tree*) from vertices to edge lists ($B = 64$). Each edge list is a map from neighbor-id to an edge-weight (or empty when unweighted) called an *edge tree* ($B = 64$). The augmentation on the vertex tree maintains the total number of edges in the graph. We focus on unweighted graphs in this paper but note that our implementation also supports weights. As with inverted indices, using difference encoding allows us to store an edge using just 2–3 bytes on average including the bytes used for regular nodes.

On top of this representation, we implement graph algorithms using the Ligra interface [41], including breadth-first search, maximal independent set, and single-source betweenness centrality. Our implementations are based on the ones in Aspen and GBBS [22, 23]. We design parallel batch-updates for our representation, which are applicable in graph-streaming and batch-dynamic graph algorithms.

9 Experiments

Experimental Setup. We run experiments on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4×2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use a work-stealing scheduler for parallelism [10]. We use `numactl -i all` to balance the memory allocations across the sockets for parallel executions. Unless otherwise mentioned, all of the reported numbers are run on 72 cores with hyper-threading.

Overview of Results We show the following experimental results in this section.

- PaC-trees are competitive with PAM for microbenchmarks (Section 9.1) and applications including inverted indices

(Section 9.2) and 2D range queries and 1D interval queries (Section 9.3) while using 2.1x–7.8x less space.
- Varying the block size B for an PaC-tree trades off off performance for space efficiency (Section 9.1). For even a modest value of $B = 128$, PaC-trees use only 1% more space than a (static) compressed array.
- For graph processing and streaming, CPAM uses 1.3–2.6x less space compared to Aspen, and is almost always faster than Aspen in all tested graph algorithms (Section 9.4).

9.1 PaC-Tree Performance

We begin by studying the performance and space of PaC-trees on a set of microbenchmarks and compare with P-trees from PAM. All experiments in this section use maps and augmented maps where the keys and values are both 64-bit integers. Unless otherwise mentioned PaC-trees use $B = 128$.

Microbenchmark Performance. Table 2 shows the results on PaC-trees, PaC-trees with difference-encoding (DE), and P-trees for a representative subset of the map and sequence primitives. The speedups for both types of PaC-trees range from 28.7–101x and are largest for the version using DE due to additional work for difference encoding. In absolute running time, PaC-trees with DE are usually slower than PaC-trees due to compression and decompression costs, but the overhead is mostly within 10%.

In most of the primitives tested, PaC-trees are faster than P-trees while also using 2.5x less space. For example, PaC-trees are 1.68x faster than P-trees in `union` on two trees of sizes 10^8. We note that in this case, the union processes the entirety of both input trees, and so the more cache-friendly processing of blocks in PaC-trees results in lower time. However, if sizes of the two trees are different, the work for `union` only depends on the smaller size. In this case, since the cost of `union` using PaC-trees has an additional $O(mB)$ term compared with P-trees, PaC-trees are 5.5x slower than P-trees. However, we expect better performance for smaller block sizes ($B < 128$), which we discuss next.

Effect of Varying B on Performance. Fig. 9 shows the results of varying the block size B, on the performance of various operations. Most operations obtain speedups as B is increased up until $B = 16$. For the sequential operations, such as `find` and `range`, we see a steady increase in the running time for $B > 16$ and see a similar trend for Union-Imbal, which takes the union of trees with 10^8 and 10^5 elements. This slowdown with increasing B is due to the extra $O(mB)$ term in the work of `union`. For the smallest block size ($B = 1$), our running time matches that of P-trees on this operation.

Space Usage. For $B = 128$, PaC-trees obtain a 2.48x reduction in space usage compared to using P-trees, and a further 1.73x reduction in space usage by using difference encoding. The 10^8 pairs stored in the experiments require 1.6GB of memory to represent as a single flat array, which is also a lower bound for the space usage of a search tree structure. To understand how close PaC-trees come to this lower bound, we study the space usage of unaugmented maps using PaC-trees as a function of the block size B (Fig. 10). Using $B = 32$, PaC-trees are only 1.05x larger than the lower bound and using $B = 128$, it is just 1.01x larger than the lower bound. For $B = 128$, just 1.1% of the allocated memory is used for regular nodes and metadata in the flat nodes. These savings are obtained without using any additional encoding. Applying difference encoding improves the space by 1.77x over the unencoded trees and the array lower bound, and is only 1.03x larger than the space used to difference encode all of the keys in a single array, leaving the values uncompressed, which is a lower bound for a search tree structure using difference encoding for such input.

Using PaC-trees requires much lower space overhead for augmentation compared to P-trees (Fig. 10). For P-trees, adding 8 byte augmented values increases the size of the maps by 20%, whereas PaC-trees (both with and without difference encoding) using $B = 128$ incurs only a 1% increase in space for the augmented values. The savings comes from only storing a single augmented value per flat node, which only uses extra space proportional to n/B augmented values.

9.2 Inverted Index

Next, we study our performance on the inverted index application. We run the application on documents derived from a large Wikipedia dataset also used by PAM for a fair comparison. The dataset is processed by removing all markup, converting characters that are not alphanumeric to whitespace and making all words case insensitive [45]. The processed dataset contains 1.94 billion words over 8.13 million documents. Like PAM, our evaluation measures the performance of (1) building an index over (words, doc_id, weight) triples and (2) running queries that fetch the posting lists for two words, compute the intersection of the lists, and select the top 10 documents by weight.

Table 3 shows the results of the experiment. For building the index, our implementation achieves 76x speedup and our parallel running times are comparable with those of PAM (at most 1.1x slower). For the queries, we observe that the unencoded trees achieve essentially the same parallel time as PAM, whereas the difference encoded trees are 1.18x slower due to the higher cost of intersection operations in our difference encoded implementation. The space usage using PaC-trees is much smaller than that of PAM, being 3.84x smaller without encoding and 7.81x smaller using a custom encoder that combines difference encoding for the keys with byte-encoding for the integer values (weights).

9.3 Interval and Two-Dimensional Range Trees

We benchmark our interval and two-dimensional range trees as in PAM [43]. We build our interval tree on 10^8 intervals, and for queries run stabbing queries over 10^8 points in parallel. We observe that both building and querying the

	n	m	PaC-tree T_1	PaC-tree T_{144}	PaC-tree Spd.	PaC-tree (Diff) T_1	PaC-tree (Diff) T_{144}	PaC-tree (Diff) Spd.	P-tree (PAM) T_1	P-tree (PAM) T_{144}	P-tree (PAM) Spd.
No augmentation											
Size (GB)	10^8	—	1.61	—	—	0.926	—	—	4.00	—	—
Build	10^8	—	5.55	0.186	29.8	5.71	0.180	31.7	5.94	0.221	26.8
Union	10^8	10^8	5.33	0.088	60.5	6.29	0.089	70.6	8.97	0.168	53.3
Union	10^8	10^5	1.09	0.021	51.9	1.28	0.022	58.1	0.206	0.0038	54.2
Intersect	10^8	10^8	4.35	0.065	66.9	5.68	0.081	70.1	9.50	0.139	68.3
Difference	10^8	10^8	3.00	0.055	54.4	3.55	0.056	63.3	8.17	0.123	66.4
Map	10^8	10^8	0.859	0.037	22.9	1.14	0.023	49.5	1.32	0.091	14.5
Reduce	10^8	—	0.306	0.018	17.0	0.308	0.0092	33.4	1.60	0.034	47.0
Filter	10^8	—	0.997	0.028	35.6	1.24	0.018	68.8	1.90	0.0524	36.2
Find	10^8	10^8	103	1.17	88.0	125	1.23	101.6	105.5	1.05	100.4
Insert	10^8	10^6	0.829	—	—	1.42	—	—	0.773	—	—
Multi-Insert	10^8	10^8	18.8	0.332	56.6	19.9	0.323	61.6	9.67	0.338	28.6
Range	10^8	10^6	11.5	0.318	36.1	13.1	0.226	57.9	3.77	0.0738	45.6
With augmentation											
Size (GB)	10^8	—	1.63	—	—	0.936	—	—	4.80	—	—
Build	10^8	—	5.66	0.197	28.7	5.84	0.186	31.3	6.48	0.246	26.3
Union	10^8	10^8	5.52	0.098	56.3	6.52	0.090	72.4	10.13	0.196	51.6
AugRange	10^8	10^7	12.3	0.331	37.1	13.9	0.234	59.4	4.80	0.082	58.5
AugFilter	10^8	—	12.2	0.333	36.6	13.6	0.234	58.1	4.95	0.081	61.1

Table 2. Microbenchmark results. We fix $B = 128$ for PaC-trees. n is the tree size. For set functions and `multi-insert`, $m \leq n$ is the size of the other set (batch). For other functions, m is the number of queries tested. T_1 is the sequential running time. T_{144} is parallel running time using 72 cores (144 hyperthreads). *Diff* means difference encoding. We highlight the best parallel running time (or size) per experiment in green and underlined.

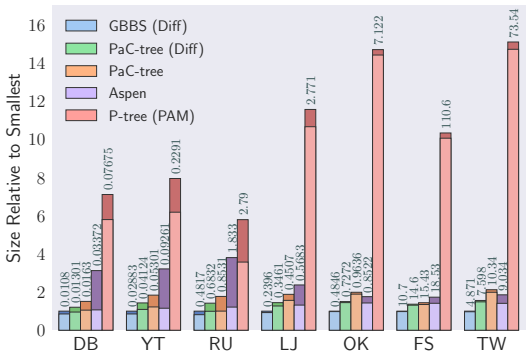

Figure 8. Relative space usage of different graph representations. GBBS (Diff) is our *static* baseline compressed graph representation. PaC-tree uses PaC-trees for vertex and edge trees, and PaC-tree (Diff) difference encodes both trees. Aspen uses P-trees for the vertex tree and C-trees with difference encoding for edge trees. P-tree (PAM) uses P-trees for the vertex and edge trees. The values on top of each bar are the memory usage in GiB.

	Library	Space	Method	n	m	T_1	T_{144}	Spd.
Inverted Index	PaC-tree	8.29	*Build*	10^8	—	746	9.73	76.6
			Query	10^8	10^8	341	4.46	76.4
	PaC-tree (D)	4.07	*Build*	10^8	—	754	9.81	76.8
			Query	10^8	10^8	367	5.32	68.9
	P-tree (PAM)	31.9	*Build*	10^8	—	575	8.86	64.9
			Query	10^8	10^8	313	4.48	69.8
Interval	PaC-tree	0.812	*Build*	10^8	—	10.9	0.179	60.8
			Query	10^8	10^8	60.8	0.525	115.8
	P-tree (PAM)	3.54	*Build*	10^8	—	11.6	0.271	42.8
			Query	10^8	10^8	54.3	0.628	86.4
Range	PaC-tree	40.3	*Build*	10^8	—	164	2.71	60.7
			Q-Sum	10^8	10^6	54.2	0.629	86.1
			Q-All	10^8	10^3	7.20	0.266	27.0
	P-tree (PAM)	89.6	*Build*	10^8	—	169	2.84	59.6
			Q-Sum	10^8	10^6	60.7	0.735	82.5
			Q-All	10^8	10^3	21.6	0.552	39.1

Table 3. Build and query times and space usage in GiB for inverted index, interval tree, and range tree applications. T_1 is the single-thread time, T_{144} is the 72-core time using hyper-threading, and Spd. is the parallel speedup. The best parallel running time (or size) is highlighted in green and underlined per experiment.

trees achieves good parallel speedup (60–115x). PaC-trees are 1.51x faster than PAM in construction, and is 1.19x faster for queries. Overall we find that PaC-trees enable better performance than PAM while using 4.37x less space.

We build our range trees on 10^8 uniformly random points in the plane between $(0, 0)$ and $(1e8, 1e8)$. We run two types of queries: the first count the number of points in the range (Q-Sum), and the second returns all points in the range. We tuned the window sizes used in our queries to match the settings evaluated by PAM (around 10^6 points returned per query). Both PaC-trees and P-trees build the data structure in a similar amount of time. PaC-trees achieve better performance than P-trees for both queries, being 1.16x faster for Q-Sum and 1.96x faster for Q-All queries, likely due to requiring fewer cache-misses when processing the tree to output the points within a given range. The range tree application using PAM has previously been compared with range trees in CGAL [38] and was shown to outperform it [43].

For space usage, PaC-trees result in 2.18x less space compared to PAM. We note that 95% of the space used in PAM is for the P-trees stored as augmented values in each node (representing the union of the y-coordinates in the subtree). The majority of our savings come from compressing the augmented trees using PaC-trees which results in a 2.53x less space for the inner trees, and 2.18x less space overall.

9.4 Graph Processing and Graph Streaming

Our last set of experiments study the performance of PaC-trees for a set of standard benchmarks from the graph processing and graph streaming literature. Our evaluation roughly

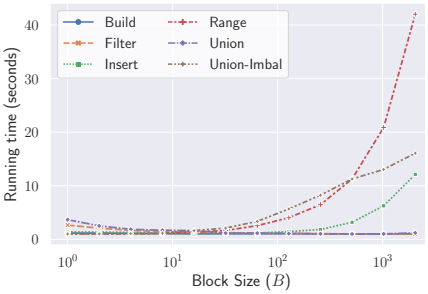

Figure 9. Primitive running times for PaC-trees vs. block size B. We use 10^8 key-value pairs (8 bytes each). Union, Intersection and Difference all work on two trees with 10^8 elements. Union-Imbal takes the union of trees with 10^8 and 10^5 elements.

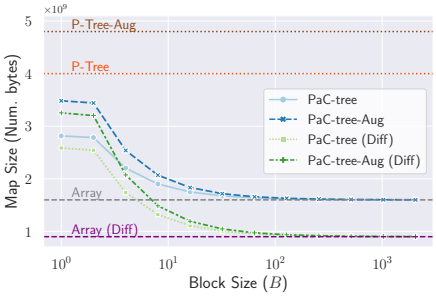

Figure 10. Size of PaC-trees (with or without DE) as a function of block size B. We use 10^8 key-value pairs (8 bytes each). For augmented maps (-Aug), augmented values are 8 bytes each. The grey line shows the number of bytes to store the 10^8 elements in an array and the purple line shows the bytes used to store the difference encoded keys in a single array using byte encoding.

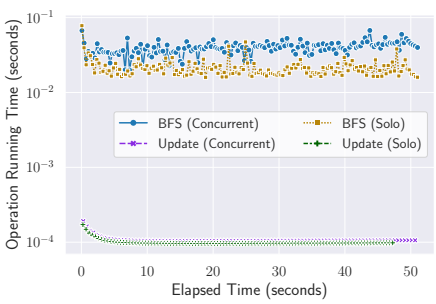

Figure 11. Performance of concurrent updates and queries. The time series plot illustrates running times when running BFS queries with batch-insertions of edges concurrently (Concurrent), and when queries and updates are run individually (Solo) on the LiveJournal graph.

Graph	Vertices	Edges	Ours	Aspen	$\frac{\text{Aspen}}{\text{Ours}}$
DBLP (DB)	425,957	2,099,732	0.0130	0.03409	2.62x
YouTube (YT)	1,138,499	5,980,886	0.0412	0.0934	2.26x
USA-Road (RU)	23,947,348	57,708,624	0.683	1.843	2.69x
LiveJournal (LJ)	4,847,571	85,702,474	0.346	0.527	1.52x
com-Orkut (CO)	3,072,627	234,370,166	0.727	0.893	1.22x
Twitter (TW)	41,652,231	2,405,026,092	7.59	9.42	1.23x
Friendster (FS)	65,608,366	3,612,134,270	14.6	19.1	1.30x

Table 4. Statistics about tested graphs and memory usage of PaC-tree and Aspen in GiB.

		Aspen		Ours				
	Graph	FS	FS Time	No-FS	FS	$\frac{\text{FS}}{\text{No-FS}}$	FS Time	$\frac{\text{Aspen}}{\text{Ours}}$
BFS	*LiveJournal*	21.7	3.82	19.8	17.5	1.13x	1.38	1.24x
	com-Orkut	15.3	2.35	14.5	12.4	1.16x	1.12	1.23x
	Twitter	138	37.8	125	112	1.11x	12.5	1.23x
MIS	*LiveJournal*	55.3	3.82	72.0	45.7	1.57x	1.38	1.21x
	com-Orkut	70.2	2.35	96.9	69.2	1.40x	1.12	1.01x
	Twitter	1022	37.8	1190	971	1.22x	12.5	1.05x
BC	*LiveJournal*	74.6	3.82	82.1	72.3	1.13x	1.38	1.03x
	com-Orkut	76.3	2.35	88.6	78.2	1.13x	1.12	0.975x
	Twitter	1150	37.8	2735	1030	2.65x	12.5	1.11x

Table 5. Parallel running times (in milliseconds) for Aspen and our implementation. We show the algorithm performance without flat snapshots (**No-FS**), with flat snapshots (**FS**), and the time to computing the flat snapshot (**FS Time**).

follows Aspen's and we compare our performance and space usage with that of Aspen and its C-tree implementation.

Graph Data and Space Usage. Most of the graphs we study are Web graphs and social networks which are low-diameter graphs that are frequently used in practice. To also test on high-diameter graphs, we ran our implementations on a road network. Complete details about our inputs are in the full version of the paper. Table 4 shows information about our graph inputs, including the number of vertices, edges, and space used.

We evaluate five graph representations including using PAM, Aspen, PaC-tree with or without difference encoding, and GBBS. Aspen uses C-trees as edge trees and leaves vertex trees uncompressed using P-trees. GBBS is a state-of-the-art static graph processing library which represents graphs as static arrays using difference encoding, which serves as our baseline of graph representation. Fig. 8 shows the relative size of each graph format. We see that the smallest format in all cases is ***PaC-tree (Diff)***, which applies PaC-trees with difference encoding for both vertex and edge trees. Using this format yields a space improvement of between 4–9.7x over just using P-trees. For the graphs with high average-degree, most of the savings come from using PaC-trees for the edge trees. Adding difference encoding to both trees yields between 1.05–1.32x space improvement. PaC-trees are also 1.3–2.6x more space-efficient than Aspen. Note that C-trees in Aspen are also difference encoded, so the main difference between the two representations is that PaC-tree (Diff) also uses PaC-trees to chunk the vertex tree, and that PaC-trees employ a deterministic strategy for chunking. PaC-trees with difference encoding achieves consistently lower space compared with Aspen, ranging between 1.3x for Friendster, our largest graph, to a maximum space improvement for 2.62x on USA-Road, our sparsest graph. The space savings come from chunking the vertex trees, which is not possible in Aspen, since the C-tree implementation is specialized for edge trees.

Graph Algorithm Performance. We study the performance of three fundamental graph kernels: breadth-first search (BFS), single-source betweenness centrality (BC), and maximal independent set (MIS). Our implementations are based on those in Aspen. We study performance using our most

space-efficient version (PaC-tree (Diff)). Following Aspen, our implementation also supports the *flat snapshot* object, which is an array storing all vertices in the current graph. The idea is that instead of accessing edges for a vertex through the vertex tree (performing tree traversal), algorithms directly access edge trees through the flat snapshot.

Table 5 shows performance results for three of our graph datasets. Across all three kernels our implementations are 1.12x faster than Aspen's implementations on average. We observe that flat snapshots can be generated 2.09–3.02x faster in CPAM due to PaC-trees requiring fewer cache-misses to traverse than P-trees when creating flat snapshot array. We note that the implementation of edgeMap and other primitives from Ligra (including constants and other tuning parameters) are exactly the same in both CPAM and Aspen. Aspen also difference encodes in its edge trees (represented using C-trees). The performance improvements that we observe are therefore a result of PaC-trees providing faster flat snapshots, and having better balance in chunk sizes compared to the randomized approach used in C-trees.

Concurrent Updates and Queries. Our last experiment concurrent updates and queries on graphs. The experiment performs n undirected edge insertions drawn from the rMAT generator (details provided in the full version). We use a batch size of 5 in the updates (10 directed edges are inserted per batch). We then spawn two parallel jobs, one performing the updates one batch after the other, and the other performing BFS queries, one after the other. Both the updates and queries are parallel (i.e., they internally make use of parallelism).

Fig. 11 shows the result of the experiment. We find that the concurrent queries are 1.85x slower on average than the queries in isolation, and that the concurrent updates are 1.07x slower on average than updates in isolation. In the concurrent setting, the average latency to make one of the update batches visible is 100 microseconds, and the updates achieve a throughput of 94,000 undirected edge updates per second. We leave further optimizations and a more in depth study of the graph setting for future work with our system.

10 Conclusion

We have presented PaC-tree, a deterministic compressed ordered map data structure and an implementation of the structure in a library CPAM. The important features of PaC-trees and its implementation in CPAM include the following.

- It is purely functional allowing for persistent snapshots while updates are being made, and safe for parallelism.
- It supports sequences, ordered sets, ordered maps, and augmented maps, with a wide variety of functions on them.
- It provides theoretical bounds on work, span, and space.
- It achieves fast sequential time and gets up to 100x speedup on 72 cores with 144 hyperthreads.
- It achieves memory usage that is close to a compressed array and up to an order of magnitude smaller than PAM.
- It is internally memory manged using reference counting.
- It is backward compatible with PAM.
- It has been used to implement the full functionality of Aspen while improving runtime and/or space.

For future work, we are interested in extending PaC-trees to support higher-fanout internal nodes, similar to B-trees, which would allow users to improve query latency at the expense of increased work when performing updates. Other future work includes applying PaC-trees to improve space utilization in databases, and to improve the performance of collection-based applications using non-volatile memory.

Acknowledgement

This work was supported by the National Science Foundation grants CCF-1901381, CCF-1910030, CCF-1919223, CCF-2103483, and CCF-2119352.

References

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2014. Theory and Practice of Chunked Sequences. In *European Symposium on Algorithms (ESA)*.

[2] Stephen Adams. 1993. Efficient sets—a balancing act. *Journal of functional programming* 3, 04 (1993).

[3] Vitaly Aksenov, Vincent Gramoli, Petr Kuznetsov, Anna Malova, and Srivatsan Ravi. 2017. A concurrency-optimal binary search tree. In *European Conference on Parallel Processing (Euro-Par)*. Springer.

[4] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *USENIX Annual Technical Conference*.

[5] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A key-value map for scalable real-time analytics. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[6] R. Bayer and E. M. McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica* 1, 3 (01 Sep 1972).

[7] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. 2021. Space and Time Bounded Multiversion Garbage Collection. In *International Symposium on Distributed Computing (DISC)*. https://doi.org/10.4230/LIPIcs.DISC.2021.12

[8] Jean-Philippe Bernardy. 2008. The Haskell Yi package. http://hackage.haskell.org/package/yi-0.6.2.3/docs/src/Data-Rope.html.

[9] Daniel K. Blandford and Guy E. Blelloch. 2004. Compact Representations of Ordered Sets. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[10] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[11] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[12] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal Parallel Algorithms in the Binary-Forking Model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[13] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2018. Parallel Write-Efficient Algorithms and Data Structures for Computational Geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[14] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999).

[15] Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An Alternative to Strings. *Softw. Pract. Exper.* 25, 12 (1995).

[16] Anastasia Braginsky and Erez Petrank. 2012. A lock-free B+ tree. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[17] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206.

[18] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[19] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *ACM European Conference on Computer Systems (EuroSys)*.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press.

[21] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[22] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021). https://doi.org/10.1145/3434393

[23] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *Intl. Workshop on Graph Data Management Experiences and Systems (GRADES)*.

[24] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[25] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent non-blocking binary search trees supporting wait-free range queries. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[26] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2008. Implicitly-threaded Parallelism in Manticore. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[27] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[28] Yan Gu, Yihan Sun, and Guy E. Blelloch. 2018. Algorithmic Building Blocks for Asymmetric Memories. In *European Symposium on Algorithms (ESA)*.

[29] Switzerland International Organization for Standardization, Geneva. 2018. ISO/IEC TS 19570:2018: Programming Languages – Technical Specification for C++ Extensions for Parallelism. https://www.iso.org/standard/70588.html.

[30] Java Fork-Join, Oracle Java Documentation [n.d.]. http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html.

[31] Haim Kaplan and Robert Endre Tarjan. 1996. Purely Functional Representations of Catenable Sorted Lists. In *ACM Symposium on Theory of Computing (STOC)*.

[32] Alfons Kemper, Thomas Neumann, Jan Finis, Florian Funke, Viktor Leis, Henrik Mühe, Tobias Mühlbauer, and Wolf Rödiger. 2013. Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. *IEEE Data Eng. Bull.* 36, 2 (2013).

[33] Edward A. Kmett. 2010. The Haskell Rope package.

[34] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (1980).

[35] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. LLAMA: Efficient graph analytics using large multiversioned arrays. In *IEEE International Conference on Data Engineering (ICDE)*.

[36] Colt McAnlis and Aleks Haekey. 2016. *Understanding Compression*. O'Reilly Media, Inc.

[37] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[38] 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

[39] Jürg Nievergelt and Edward M Reingold. 1973. Binary search trees of bounded balance. *SIAM J. on Computing* 2, 1 (1973).

[40] Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press.

[41] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[42] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *IEEE Data Compression Conference (DCC)*.

[43] Yihan Sun and Guy E Blelloch. 2019. Parallel Range, Segment and Rectangle Queries with Augmented Maps. In *Algorithm Engineering and Experiments (ALENEX)*.

[44] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment (PVLDB)* 13, 2 (2019).

[45] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[46] Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with applications to concurrent data structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[47] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016).