PAM: Parallel Augmented Maps

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

Daniel Ferizovic
Karlsruhe Institute of Technology
dani93.f@gmail.com

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

March 28, 2018

Abstract

Ordered (key-value) maps are an important and widely-used data type for large-scale data processing frameworks. Beyond simple search, insertion and deletion, more advanced operations such as range extraction, filtering, and bulk updates form a critical part of these frameworks.

We describe an interface for ordered maps that is augmented to support fast range queries and sums, and introduce a parallel and concurrent library called PAM (Parallel Augmented Maps) that implements the interface. The interface includes a wide variety of functions on augmented maps ranging from basic insertion and deletion to more interesting functions such as union, intersection, filtering, extracting ranges, splitting, and range-sums. We describe algorithms for these functions that are efficient both in theory and practice.

As examples of the use of the interface and the performance of PAM we apply the library to four applications: simple range sums, interval trees, 2D range trees, and ranked word index searching. The interface greatly simplifies the implementation of these data structures over direct implementations. Sequentially the code achieves performance that matches or exceeds existing libraries designed specially for a single application, and in parallel our implementation gets speedups ranging from 40 to 90 on 72 cores with 2-way hyperthreading.

1 Introduction

The *map* data type (also called key-value store, dictionary, table, or associative array) is one of the most important data types in modern large-scale data analysis, as is indicated by systems such as F1 [60], Flurry [5], RocksDB [57], Oracle NoSQL [50], LevelDB [41]. As such, there has been significant interest in developing high-performance parallel and concurrent algorithms and implementations of maps (e.g., see Section 2). Beyond simple insertion, deletion, and search, this work has considered “bulk” functions over ordered maps, such as unions [20, 33, 12], bulk-insertion and bulk-deletion [26, 6, 24], and range extraction [14, 55, 7].

One particularly useful function is to take a “sum” over a range of keys, where *sum* means with respect to any associative combine function (e.g., addition, maximum, or union). As an example of such a *range sum* consider a database of sales receipts keeping the value of each sale ordered by the time of sale. When analyzing the data for certain trends, it is likely useful to quickly query the sum or maximum of sales during a

Application	In Theory (Asymptotic bound)			In Practice (Running Time in seconds)							
	Construct		Query	Construct				Query			
	Work	Span		Size	Seq.	Par.	Spd.	Size	Seq.	Par.	Spd.
Range Sum	$O(n \log n)$	$O(\log n)$	$O(\log n)$	10^{10}	1844.38	28.24	65.3	10^8	271.09	3.04	89.2
Interval Tree	$O(n \log n)$	$O(\log n)$	$O(\log n)$	10^8	14.35	0.23	63.2	10^8	53.35	0.58	92.7
2d Range Tree	$O(n \log n)$	$O(\log^3 n)$	$O(\log^2 n)$	10^8	197.47	3.10	63.7	10^6	48.13	0.55	87.5
Inverted Index	$O(n \log n)$	$O(\log^2 n)$	*	1.96×10^9	1038	12.6	82.3	10^5	368	4.74	77.6

Table 1: The asymptotic cost and experimental results of the applications using PAM. **Seq.** = sequential, **Par.** = Parallel (on 72 cores with 144 hyperthreads), **Spd.** = Speedup. "Work" and "Span" are used to evaluate the theoretical bound of parallel algorithms (see Section 4). *: Depends on the query.

period of time. Although such sums can be implemented naively by scanning and summing the values within the key range, these queries can be answered much more efficiently using augmented trees [25, 18]. For example, the sum of any range on a map of size n can be answered in $O(\log n)$ time. This bound is achieved by using a balanced binary tree and augmenting each node with the sum of the subtree.

Such a data structure can also implement a significantly more general form of queries efficiently. In the sales receipts example they can be used for reporting the sales above a threshold in $O(k \log(n/k + 1))$ time (k is the output size) if the augmentation is the maximum of sales, or in $O(k + \log n)$ time [44] with a more complicated augmentation. More generally they can be used for interval queries, k-dimensional range queries, inverted indices (all described later in the paper), segment intersection, windowing queries, point location, rectangle intersection, range overlaps, and many others.

Although there are dozens of implementations of efficient range sums, there has been very little work on parallel or concurrent implementations—we know of none for the general case, and only two for specific applications [2, 34]. In this paper we present a general library called PAM (Parallel Augmented Maps) for supporting in-memory parallel and concurrent maps with range sums. PAM is implemented in C++. We use *augmented value* to refer to the abstract "sum" on a map (defined in Section 3). When creating a map type the user specifies two augmenting functions chosen based on their application: a *base* function g which gives the augmented value of a single element, and a *combine* function f which combines multiple augmented values, giving the augmented value of the map. The library can then make use of the functions to keep "partial sums" (augmented values of sub-maps) in a tree that asymptotically improve the performance of certain queries.

Augmented maps in PAM support standard functions on ordered maps (which maintain the partial sums), as well as additional function specific to augmented maps (see Figure 1 for a partial list). The standard functions include simple functions such as insertion, and bulk functions such as union. The functions specific to augmented maps include efficient range-sums, and filtering based on augmented values. PAM uses theoretically efficient parallel algorithms for all bulk functions, and is implemented based on using the "join" function to support parallelism on balanced trees [12]. We extend the approach of using joins to handle augmented values, and also give algorithms based on "join" for some other operations such as filtering, multi-insert, and mapReduce.

PAM uses functional data structures and hence the maps are fully persistent—updates will not modify an existing map but will create a new version [22]. Persistence is useful in various applications, including the range tree and inverted index applications described in this paper. It is also useful in supporting a form of concurrency based on snapshot isolation. In particular each concurrent process can atomically read a snapshot of a map, and can manipulate and modify their "local" copy without affecting the view of other users, or being affected by any other concurrent modification to the shared copy[1]. However PAM does not directly support traditional concurrent updates to a shared map. Instead concurrent updates need to be batched and applied in bulk in parallel.

We present examples of four use cases for PAM along with experimental performance numbers. Firstly we consider the simple case of maintaining the sum of the values in a map using integer addition. For this case we report both sequential and parallel times for a wide variety of operations (union, search, multi-insert, range-sum, insertion, deletion, filter, build). We also present performance comparisons to other implementations of maps that do not support augmented values. Secondly we use augmented maps to implement interval trees. An interval tree maintains a set of intervals (e.g. the intervals of times in which users are logged into a site, or the intervals of time for FTP connections) and can quickly answer queries such as if a particular point is covered by any interval (e.g. is there any user logged in at a given time). Thirdly we implement 2d range trees. Such trees maintain a set of points in 2 dimensions and allow one to count or list all entries within a given rectangular range (e.g. how many users are between 20 and 25 years old and have salaries between \$50K and \$90K). Such counting queries can be answered in $O(\log^2 n)$ time. We present performance comparison to the sequential range-tree structure available in CGAL [47]. Finally we implement a weighted inverted index that supports and/or queries, which can quickly return the top k matches. The theoretical cost and practical performance of these four applications are shown in Table 1.

The main contributions of this paper are:

1. An interface for *augmented maps* (Section 3).
2. Efficient parallel algorithms and an implementation for the interface as part of the PAM library (Section 4).
3. Four example applications of the interface (Section 5).
4. Experimental analysis of the examples (Section 6).

2 Related Work

Many researchers have studied concurrent algorithms and implementations of maps based on balanced search trees focusing on insertion, deletion and search [36, 37, 26, 13, 6, 24, 10, 15, 21, 40, 46]. Motivated by applications in data analysis recently researchers have considered mixing atomic range scanning with concurrent updates (insertion and deletion) [14, 55, 7]. None of them, however, has considered sub-linear time range sums.

There has also been significant work on parallel algorithms and implementations of bulk operations on ordered maps and sets [53, 52, 11, 26, 20, 33, 24, 12, 3]. Union and intersection, for example, are available as part of the multicore version of the C++ Standard Template Library [26]. Again none of this work has considered fast range sums. There has been some work on parallel data structures for specific applications of range sums such as range trees [34].

There are many theoretical results on efficient sequential data-structures and algorithms for range-type queries using augmented trees in the context of specific applications such as interval queries, k-dimensional range sums, or segment intersection queries (see e.g. [43]). Several of these approaches have been implemented as part of systems [35, 47]. Our work is motivated by this work and our goal is to parallelize many of

---

[1] Throughout the paper we use *parallel* to indicate using multiple processors to work on a single bulk function, such as multi-insertion or filtering, and we use *concurrent* to indicate independent "users" (or processes) asynchronously accessing the same structure at the same time.

(Partial) Interface AugMap $\mathbb{AM}(K, V, A, <, g, f, I)$:

empty or $\emptyset$: M
size : $M \to \mathbb{Z}$
single : $K \times V \to M$
find : $M \times K \to V \cup \{\square\}$
insert : $M \times K \times V \times (V \times V \to V) \to M$
union : $M \times M \times (V \times V \to V) \to M$
filter : $(K \times V \to$ **bool**$) \times M \to M$
upTo : $M \times K \to M$
range : $M \times K \times K \to M$
mapReduce : $(K \times V \to B) \times (B \times B \to B) \times B \times M \to B$
build : $(K \times V)$ **seq.** $\times (V \times V \to V) \to M$
augVal : $M \to A$
augLeft : $M \times K \to A$
augRange : $M \times K \times K \to A$
augFilter : $(A \to$ **bool**$) \times M \to M$
augProject : $(A \to B) \times (B \times B \to B) \times M \times K \times K \to B$

Figure 1: The (partial) interface for an augmented map, with key type K, value type V, and augmented value type A. The augmenting monoid is (A, f, I). Other functions not listed include **delete**, **intersect**, **difference**, **split**, **join**, **downTo**, **previous**, **next**, **rank**, and **select**. In the table **seq.** means a sequence.

these ideas and put them in a framework in which it is much easier to develop efficient code. We know of no other general framework as described in Section 3.

Various forms of range queries have been considered in the context of relational databases [29, 28, 31, 16, 27]. Ho et. al. [31] specifically consider fast range sums. However the work is sequential, only applies to static data, and requires that the sum function has an inverse (e.g. works for addition, but not maximum). More generally, we do not believe that traditional (flat) relational databases are well suited for our approach since we use arbitrary data types for augmentation—e.g. our 2d range tree has augmented maps nested as their augmented values. Recently there has been interest in range queries in large clusters under systems such as Hadoop [4, 2]. Although these systems can extract ranges in work proportional to the number of elements in the range (or close to it), they do not support fast range sums. None of the "nosql" systems based on key-value stores [5, 41, 57, 50] support fast range sums.

3 Augmented Maps

Augmented maps, as defined here, are structures that associate an ordered *map* with a "sum" (the *augmented value*) over all entries in the map. It is achieved by using a base function g and a combine function f. More formally, an augmented map type $\mathbb{AM}$ is parameterized on the following:

K,	key type
$<\ : K \times K \to$ **bool**,	total ordering on the keys
V,	value type
A,	augmented value type
$g : K \times V \to A$,	the base function
$f : A \times A \to A$,	the combine function
$I : A$	identity for f

The first three parameters correspond to a standard ordered map, and the last four are for the augmentation. f must be associative ((A, f, I) is a monoid), and we use $f(a_1, a_2, \ldots, a_n)$ to mean any nesting. Then the *augmented value* of a map $m = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$ is defined as:

$$\mathcal{A}(m) = f(g(k_1, v_1), g(k_2, v_2), \ldots, g(k_n, v_n))$$

As an example, the augmented map type:

$$\mathbb{AM}(\mathbb{Z}, <_{\mathbb{Z}}, \mathbb{Z}, \mathbb{Z}, (k, v) \to v, +_{\mathbb{Z}}, 0) \tag{1}$$

defines an augmented map with integer keys and values, ordered by $<_{\mathbb{Z}}$, and for which the augmented value of any map of this type is the sum of its values.

An augmented map type supports an interface with standard functions on ordered maps as well as a collection of functions that make use of f and g. Figure 1 lists an example interface, which is the one used in this paper and supported by PAM. In the figure, the definitions above the dashed line are standard definitions for an ordered map. For example, the **range**(m, k_1, k_2) extracts the part of the map between keys k_1 and k_2, inclusive, returning a new map. The **mapReduce**(g', f', I', m) applies the function g' to each element of the map m, and then sums them with the associative function f' with identity I'. Some functions listed in Figure 1, such as UNION, INSERT and BUILD, take an addition argument h, which itself is a function. When applicable it combines values of all entries with the same key. For example the **union**(m_1, m_2, h) takes union of two maps, and if any key appears in both maps, it combines their values using h.

Most important to this paper are the definitions below the dashed line, which are functions specific to augmented maps. All of them can be computed using the functions above the dashed line. However, they can be much more efficient by maintaining the augmented values of sub-maps (partial sums) in a tree structure. Table 2 gives the asymptotic costs of the functions based on the implementation described in Section 4, which uses augmented balanced search trees. The function **augVal**(m) returns $\mathcal{A}(m)$, which is equivalent to **mapReduce**(g, f, I, m) but can run in constant instead of linear work. This is because the functions f and g are chosen ahead of time and integrated into the augmented map data type, and therefore the sum can be maintained during updates. The function **augRange**(m, k_1, k_2) is equivalent to **augVal**(**range**(m, k_1, k_2)) and **augLeft**(m, k) is equivalent to **augVal**(**upTo**(m, k)). These can also be implemented efficiently using the partial sums.

The last two functions accelerate two common queries on augmented maps, but are only applicable when their function arguments meet certain requirements. They also can be computed using the plain map functions, but can be much more efficient when applicable. The **augFilter**(h, m) function is equivalent to **filter**(h', m), where $h' : K \times V \mapsto$ **bool** satisfies $h(g(k, v)) \Leftrightarrow h'(k, v)$. It is only applicable if $h(a) \vee h(b) \Leftrightarrow h(f(a, b))$ for any a and b ($\vee$ is the logical or). In this case the **filter** function can make use of the partial sums. For example, assume the values in the map are boolean values, f is a logical-or, $g(k, v) = v$, and we want to filter the map using function $h'(k, v) = v$. In this case we can filter out a whole sub-map once we see it has `false` as a partial sum. Hence we can set $h(a) = a$ and directly use **augFilter**(m, h). The function is used in interval trees (Section 5.1). The **augProject**(g', f', m, k_1, k_2) function is equivalent to g'(**augRange**(m, k_1, k_2)). It requires, however, that $(B, f', g'(I))$ is a monoid and that $f'(g'(a), g'(b)) = g'(f(a, b))$. This function is useful when the augmented values are themselves maps or other large data structures. It allows projecting the augmented values down onto another type by g' (e.g. project augmented values with complicated structures to values like their sizes) then summing them by f', and is much more efficient when applicable. For example in range trees where each augmented value is itself an augmented map, it greatly improves performance for queries.

4 Data Structure and Algorithms

In this section we outline a data structure and associated algorithms used in PAM that can be used to efficiently implement augmented maps. Our data structure is based on abstracting the balancing criteria of a class of trees (e.g. AVL or Red-Black trees) in terms of a single JOIN function [12], which joins two maps with a key between them (defined more formally below). Prior work describes parallel algorithms for UNION, DIFFERENCE, and INTERSECT for standard un-augmented sets and maps using just JOIN [12]. Here we extend the methodology to handle augmentation, and also describe some other functions based on join. Because the balancing criteria is fully abstracted in JOIN, similar algorithm can be applied to AVL trees [1] red-black trees [8], weight-balanced trees and treaps [59]. We implemented all of them in PAM. By default, we use weight-balanced trees [48] in PAM, because it does not require extra balancing criteria in each node (the node size is already stored), but users can change to any specific balancing scheme using C++ templates.

Augmentation. We implement augmentation by storing with every tree node the augmented sum of the subtree rooted at that node. This localizes application of the augmentation functions f and g to when a node is created or updated[2]. In particular when creating a node with a left child L, right child R, key k and value v the augmented value can be calculated as $f(\mathcal{A}(L), f(g(k, v), \mathcal{A}(R)))$, where $\mathcal{A}(\cdot)$ extracts the augmented value from a node. Note that it takes two applications of f since we have to combine three values, the left, middle and right. We do not store $g(k, v)$. In our algorithms, creation of new nodes is handled in JOIN, which also deals with rebalancing when needed. Therefore all the algorithms and code that do not explicitly need the augmented value are unaffected by and even oblivious of augmentation.

Parallelism. We use fork-join parallelism to implement internal parallelism for the bulk operations. In pseudocode the notation "$s_1 \mid\mid s_2$" means that the two statements s_1 and s_2 can run in parallel, and when both are finished the overall statement finishes. In most cases parallelism is over the structure of the trees—i.e. applying some function in parallel over the two children, and applying this parallelism in a nested fashion recursively (Figure 2 shows several examples). The only exception is **build**, where we use parallelism in a sort and in removing duplicates. In the PAM library fork-join parallelism is implemented with the cilkplus extensions to C++ [39]. We have a granularity set so parallelism is not used on very small trees.

Theoretical bounds. To analyze the asymptotic costs in theory we use work (W) and span (S), where work is the total number of operations (the sequential cost) and span is the length of the critical path [18]. Almost all the algorithms we describe, and implemented in PAM, are asymptotically optimal in terms of work in the comparison model, i.e., the total number of comparisons. Furthermore they achieve very good parallelism (i.e. polylogarithmic span). Table 2 list the cost of some of the functions in PAM. When the augmenting functions f and g both take constant time, the augmentation only affects all the bounds by a constant factor.

---

[2] For supporting persistence, updating a tree node, e.g., when it is involved in rotations and is set a new child, usually results in the creation of a new node. See details in the persistence part.

Function	***Work***	***Span***
Map operations		
insert, delete, find, first, last, previous, next, rank, select, upTo, downTo	$\log n$	$\log n$
Bulk operations		
join	$\log n - \log m$	$\log n - \log m$
union*, intersect*, difference*	$m \log\left(\frac{n}{m} + 1\right)$	$\log n \log m$
mapReduce	n	$\log n$
filter	n	$\log^2 n$
range, split, join2	$\log n$	$\log n$
build	$n \log n$	$\log n$
Augmented operations		
augVal	1	1
augRange, augProject	$\log n$	$\log n$
augFilter (output size k)	$k \log(n/k + 1)$	$\log^2 n$

Table 2: The core functions in PAM and their asymptotic costs (all big-O). The cost is given under the assumption that the base function g, the combine function f and the functions as parameters (e.g., for AUGPROJECT) take constant time to return. For the functions noted with *, the efficient algorithms with bounds shown in the table are introduced and proved in [12]. For functions with two input maps (e.g., UNION), n is the size of the larger input, and m of the smaller.

Furthermore experiments show that this constant factor is reasonably small, typically around 10% for simple functions such as summing the values or taking the maximum.

Join, Split, Join2 and Union. As mentioned, we adopt and extend the methodology introduced in [12], which builds all map functions using JOIN(L, k, R). The JOIN function takes two ordered maps L and R and a key-value pair (k, v) that sits between them (i.e. $\max(L) < k < \min(R)$) and returns the composition of L, (k, v) and R. Using JOIN it is easy to implement two other useful functions: SPLIT and JOIN2. The function $\langle L, v, R\rangle = \text{SPLIT}(A, k)$ splits the map A into the elements less than k (placed in L) and those greater (placed in R). If k appears in A its associated value is returned as v, otherwise v is set to be an empty value noted as $\Box$. The function $T = \text{JOIN2}(T_L, T_R)$ works similar to JOIN, but without the middle element. These are useful in the other algorithms. Algorithmic details can be found in [12].

The first function in Figure 2 defines an algorithm for UNION based on SPLIT and JOIN. We add a feature that it can accept a function h as parameter. If one key appears in both sets, the values are combined using h. In the pseudocode we use $\langle L, k, v, R\rangle = T$ to extract the left child, key, value and right child from the tree node T, respectively. The pseudocode is written in a functional (non side-effecting) style. This matches our implementation, as discussed in ***persistence*** below.

Insert and Delete. Instead of the classic implementations of INSERT and DELETE, which are specific to the balancing scheme, we define versions based purely on JOIN, and hence independent of the balancing scheme.

```
$\textsc{Union}(T_1, T_2, h) =$
  if $T_1 = \emptyset$ then $T_2$
  else if $T_2 = \emptyset$ then $T_1$
  else let $\langle L_2, k, v, R_2 \rangle = T_2$
          and $\langle L_1, v', R_2 \rangle = \textsc{Split}(T_1, k)$
          and $L = \textsc{Union}(L_1, L_2) \mid\mid R = \textsc{Union}(R_1, R_2)$
       in if $v' \neq \Box$ then $\textsc{Join}(L, k, h(v, v'), R)$
          else $\textsc{Join}(L, k, v, R)$
$\textsc{Insert}(T, k, v, h) =$
  if $T = \emptyset$ then $\textsc{Singleton}(k, v)$
  else let $\langle L, k', v', R \rangle = T$ in
    if $k < k'$ then $\textsc{Join}(\textsc{Insert}(L, k, v), k', v', R)$
    else if $k > k'$ then $\textsc{Join}(L, k', v', \textsc{Insert}(R, k, v))$
    else $\textsc{Join}(L, k, h(v', v), R)$
$\textsc{MapReduce}(T, g', f', I') =$
  if $T = \emptyset$ then $I'$
  else let $\langle L, k, v, R \rangle = T$
          and $L' = \textsc{MapReduce}(L, g', f', I') \mid\mid$
             $R' = \textsc{MapReduce}(R, g', f', I')$
       in $f'(L', g'(k, v), R')$
$\textsc{AugLeft}(T, k') =$
  if $T = \emptyset$ then $I$
  else let $\langle L, k, v, R \rangle = T$
       in if $k' < k$ then $\textsc{AugLeft}(L, k')$
          else $f(\mathcal{A}(L), g(k, v), \textsc{AugLeft}(R, k'))$
$\textsc{Filter}(T, h) =$
  if $T = \emptyset$ then $\emptyset$
  else let $\langle L, k, v, R \rangle = T$
          and $L' = \textsc{Filter}(L, h) \mid\mid R' = \textsc{Filter}(R, h)$
       in if $h(k, v)$ then $\textsc{Join}(L', k, v, R')$ else $\textsc{Join2}(L', R')$
$\textsc{AugFilter}(T, h) =$
  if $(T = \emptyset)$ OR $(\neg h(\mathcal{A}(T)))$ then $\emptyset$
  else let $\langle L, k, v, R \rangle = T$
          and $L' = \textsc{AugFilter}(L, h) \mid\mid$
               $R' = \textsc{AugFilter}(R, h)$
       in if $h(g(k, v))$ then $\textsc{Join}(L', k, v, R')$ else $\textsc{Join2}(L', R')$
$\textsc{Build'}(S, i, j) =$
  if $i = j$ then $\emptyset$
  else if $i + 1 = j$ then $\textsc{Singleton}(S[i])$
  else let $m = (i + j)/2$
          and $L = \textsc{Build'}(S, i, m) \mid\mid R = \textsc{Build'}(S, m + 1, j)$
       in $\textsc{Join}(L, S[m], R)$
$\textsc{Build}(S) =$
  $\textsc{Build'}(\textsc{RemoveDuplicates}(\textsc{Sort}(S)), 0, |S|)$
```

Figure 2: Pseudocode for some of the functions on augmented maps. UNION is from [12], the rest are new. For an associated binary function f, $f(a, b, c)$ means $f(a, f(b, c))$.

Like the UNION function, INSERT also takes an addition function h as input, such that if the key to be inserted is found in the map, the values will be combined by h. The algorithm for insert is given in Figure 2, and the algorithm for deletion is similar. The algorithms run in $O(\log n)$ work (and span since sequential). One might expect that abstracting insertion using JOIN instead of specializing for a particular balance criteria has significant overhead. Our experiments show this is not the case—and even though we maintain the reference counter for persistence, we are only 17% slower sequentially than the highly-optimized C++ STL library (see section 6).

Build. To construct an augmented map from a sequence of key-value pairs we first sort the sequence by the keys, then remove the duplicates (which are contiguous in sorted order), and finally use a balanced divide-and-conquer with JOIN. The algorithm is given in Figure 2. The work is then $O(W_{\text{sort}(n)} + W_{\text{remove}(n)} + n)$ and the span is $O(S_{\text{sort}(n)} + S_{\text{remove}(n)} + \log n)$. For work-efficient sort and remove-duplicates with $O(\log n)$ span this gives the bounds in Table 2.

Reporting Augmented Values. As an example, we give the algorithm of AUGLEFT(T, k') in Figure 2, which returns the augmented value of all entries with keys less than k'. It compares the root of T with k', and if k' is smaller, it calls AUGLEFT on its left subtree. Otherwise the whole left subtree and the root should be counted. Thus we directly extract the augmented value of the left subtree, convert the entry in the root to an augmented value by g, and recursively call AUGLEFT on its right subtree. The three results are combined using f as the final answer. This function visits at most $O(\log n)$ nodes, so it costs $O(\log n)$ work and span assuming f, g and I return in constant time. The AUGRANGE function, which reports the augmented value of all entries in a range, can be implemented similarly with the same asymptotical bound.

Filter and AugFilter. The filter and augFilter function both select all entries in the map satisfying condition h. For a (non-empty) tree T, FILTER recursively filters its left and right branches in parallel, and combines the two results with JOIN or JOIN2 depending on whether h is satisfied for the entry at the root. It takes linear work and $O(\log^2 n)$ span for a balanced tree. The **augFilter**(h, m) function has the same effect as **filter**(h', m), where $h' : K \times V \mapsto$ **bool** satisfies $h(g(k, v)) \Leftrightarrow h'(k, v)$ and is only applicable if $h(a) \vee h(b) \Leftrightarrow h(f(a, b))$. This can asymptotically improve efficiency since if $h(\mathcal{A}(T))$ is false, then we know that h will not hold for any entries in the tree T[3], so the search can be pruned (see Figure 1). For k output entries, this function takes $O(k \log(n/k + 1))$ work, which is asymptotically smaller than n and is significantly more efficient when k is small. Its span is $O(\log^2 n)$.

Other Functions. We implement many other functions in PAM, including all in Figure 1. In MAPREDUCE(g', f', I', T), for example, it is recursively applied on T's two subtrees in parallel, and g' is applied to the root. The three values are then combined using f'. The function AUGPROJECT(g', f', m, k_1, k_2) on the top level adopts a similar method as AUGRANGE to get related entries or subtrees in range $[k_1, k_2]$, projects g' to their augmented values and combine results by f'.

Persistence. The PAM library uses functional data structures, and hence does not modify existing tree nodes but instead creates new ones [49]. This is not necessary for implementing augmented maps, but is helpful in the parallel and concurrent implementation. Furthermore the fact that functional data structures are persistent (no existing data is modified) has many applications in developing efficient data structures [22]. In this paper three of our four applications (maintaining inverted indices, interval trees and range trees) use persistence in a critical way. The JOIN function copies nodes along the join path, leaving the two original

---

[3] Similar methodology can be applied if there exists a function h'' to decide if all entries in a subtree will be selected just by reading the augmented value.

trees unchanged. All of our code is built on JOIN in the functional style, returning new trees rather than modifying old ones. Such functional data structures mean that parts of trees are shared, and that old trees for which there are no longer any pointers need to be garbage collected. We use a reference counting garbage collector. When the reference count is one we use a standard reuse optimization—reusing the current node instead of collecting it and allocating a new one [32].

Concurrency. In PAM any number of users can concurrently access and update their local copy (snapshot) of any map. This is relatively easy to implement based on functional data structures (persistent) since each process only makes copies of new data instead of modifying shared data. The one tricky part is maintaining the reference counts and implementing the memory allocator and garbage collector to be safe for concurrency. This is all implemented in a lock-free fashion. A compare-and-swap is used for updating reference counts, and a combination of local pools and a global pool are used for memory allocation. Updates to the shared instance of a map can be made atomically by swapping in a new pointer (e.g., with a compare-and-swap). This means that updates are sequentialized. However they can be accumulated and applied when needed in bulk using the parallel multi-insert or multi-delete.

5 Applications

In this section we describe applications that can be implemented using the PAM interface. Our first application is given in Equation 1, which is a map storing integer keys and values, and keeping track of sum over values. In this section we give three more involved applications of augmented maps: interval trees, range trees and word indices (also called inverted indices). We note that although we use trees as the implementation, the abstraction of the applications to augmented maps is independent of representations.

5.1 Interval Trees

We give an example of how to use our interface for interval trees [23, 25, 18, 43, 35, 19]. This data structure maintains a set of intervals on the real line, each defined by a left and right endpoint. Various queries can be answered, such as a stabbing query which given a point reports whether it is in an interval.

There are various versions of interval trees. Here we discuss the version as described in [18]. In this version each interval is stored in a tree node, sorted by the left endpoint (key). A point x is covered by in an interval in the tree if the maximum right endpoint for all intervals with keys less than x is greater than x (i.e. an interval starts to the left and finishes to the right of x). By storing at each tree node the maximum right endpoint among all intervals in its subtree, the stabbing query can be answered in $O(\log n)$ time. An example is shown in Figure 4.

In our framework this can easily be implemented by using the left endpoints as keys, the right endpoints as values, and using $\max$ as the combining function. The definition is:

$I = \mathbb{AM}(\mathbb{R}, <_{\mathbb{R}}, \mathbb{R}, \mathbb{R}, (k, v) \mapsto v, \max_{\mathbb{R}}, -\infty)$

Figure 3 shows the C++ code of the interval tree structure using PAM. The entry with augmentation is defined in `entry` starting from line 3, containing the key type `key_t`, value type `val_t`, comparison function `comp`, augmented value type (`aug_t`), the base function g (`base`), the combine function f (`combine`), and the identity of f (`identity`). An augmented map (line 16) is then declared as the interval tree structure with `entry`. The constructor on line 18 builds an interval tree from an array of n intervals by directly calling the augmented-map constructor in PAM ($O(n \log n)$ work). The function `stab(p)` returns if p is inside any interval using `amap::aug_left(m,p)`. As defined in Section 3 and 4, this function returns the augmented sum, which is the $\max$ on values, of all entries with keys less than p. As mentioned we need only to compare it with p. The function `report_all(p)` returns all intervals containing p, which are those with keys less than p but values larger than p. We first get the sub-map in `m` with keys less then p (`amap::upTo(m,p)`), and filter all with values larger than p. Note that $h(a) = (a > p)$ and the combine function $f(a, b) = \max(a, b)$ satisfy $h(a) \vee h(b) \Leftrightarrow h(f(a, b))$. This means that to get all nodes with values $> p$, if the maximum value of a subtree is less than p, the whole subtree can be discarded. Thus we can apply `amap::aug_filter` ($O(k \log(n/k + 1))$ work for k results), which is more efficient than a plain filter.

```
struct interval_map {
  using interval = pair<point, point>;
  struct entry {
    using key_t = point;
    using val_t = point;
    using aug_t = point;
    static bool comp(key_t a, key_t b)
      { return a < b;}
    static aug_t identity()
      { return 0;}
    static aug_t base(key_t k, val_t v)
      { return v;}
    static aug_t combine(aug_t a, aug_t 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);}

  amap report_all(point p) {
    amap t = amap::up_to(m,p);
    auto h = [] (P a) -> bool {return a>p;}
    return amap::augFilter(t,h);}};
```

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

5.2 Range Trees

Given a set of n points $\{p_i = (x_i, y_i)\}$ in the plane, where $x_i \in X, y_i \in Y$, each point with weight $w_i \in W$, a *2D range sum query* asks for the sum of weights of points within a rectangle defined by a horizontal range (x_L, x_R) and vertical range (y_L, y_R). A *2D range query* reports all points in the query window. In this section, we describe how to adapt 2D range trees to the PAM framework to efficiently support these queries.

The standard 2D range tree [9, 43, 58] is a two-level tree (or map) structure. The outer level stores all the points ordered by the x-coordinates. Each tree node stores an inner tree with all points in its subtree but ordered by the y-coordinates. Then a range query can be done by two nested queries on x- and y-coordinates respectively. Sequential construction time and query time is $O(n \log n)$ and $O(\log^2 n)$ respectively (the query time can be reduced to $O(\log n)$ with reasonably complicated approaches).

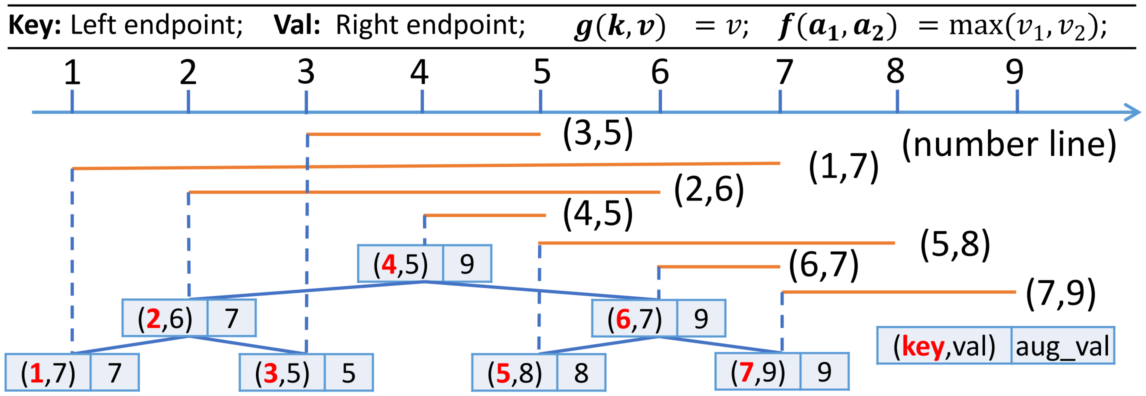


Figure 4: An example of an interval tree.

In our interface the outer tree (R_O) is represented as an augmented map in which keys are points (sorted by x-coordinates) and values are weights. The augmented value, which is the inner tree, is another augmented map (R_I) storing all points in its subtree using the points as the key (sorted by y-coordinates) and the weights as the value. The inner map is augmented by the sum of the weights for efficiently answering range sums. Union is used as the combine function for the outer map. The range tree layout is illustrated in in Figure 5, and the definition in our framework is:

$$R_I = \mathbb{AM}\,(\,P, <_Y, W, W, (k,v) \to v, \quad +_W, 0_W\,)$$
$$R_O = \mathbb{AM}\,(\,P, <_X, W, R_I, R_I.\text{singleton}, \cup, \quad \emptyset\,)$$

Here $P = X \times Y$ is the point type. W is the weight type. $+_W$ and 0_W are the addition function on W and its identity respectively.

It is worth noting that because of the persistence supported by the PAM library, the combine function UNION does not affect the inner trees in its two children, but builds a new version of R_I containing all the elements in its subtree. This is important in guaranteeing the correctness of the algorithm.

To answer the query, we conduct two nested range searches: (x_L, x_R) on the outer tree, and (y_L, y_R) on the related inner trees [43, 58]. It can be implemented using the augmented map functions as:

```
QUERY($r_O, x_L, x_R, y_L, y_R$) =
  let $g'(r_I)$ = AUGRANGE($r_I, y_L, y_R$)
  in AUGPROJECT($g', +_W, r_O, x_L, x_R$)
```

The **augProject** on R_O is the top-level searching of x-coordinates in the outer tree, and g' projects the inner trees to the weight sum of the corresponding y-range. f' (i.e., $+_W$) combines the weight of all results of g' to give the sum of weights in the rectangle. When f' is an addition, g' returns the range sum, and f is a UNION, the condition $f'(g'(a), g'(b)) = g'(a) + g'(b) = g'(a \cup b) = g'(f(a,b))$ holds, so AUGPROJECT is applicable. Combining the two steps together, the query time is $O(\log^2 n)$. We can also answer range queries that report all point inside a rectangle in time $O(k + \log^2 n)$, where k is the output size.

5.3 Ranked Queries on Inverted Indices

Our last application of augmented maps is building and searching a weighted inverted index of the kind used by search engines [66, 56] (also called an inverted file or posting file). For a given corpus, the index stores a mapping from words to second-level mappings. Each second-level mapping, maps each document that the term appears in to a weight, corresponding to the importance of the word in the document and the importance

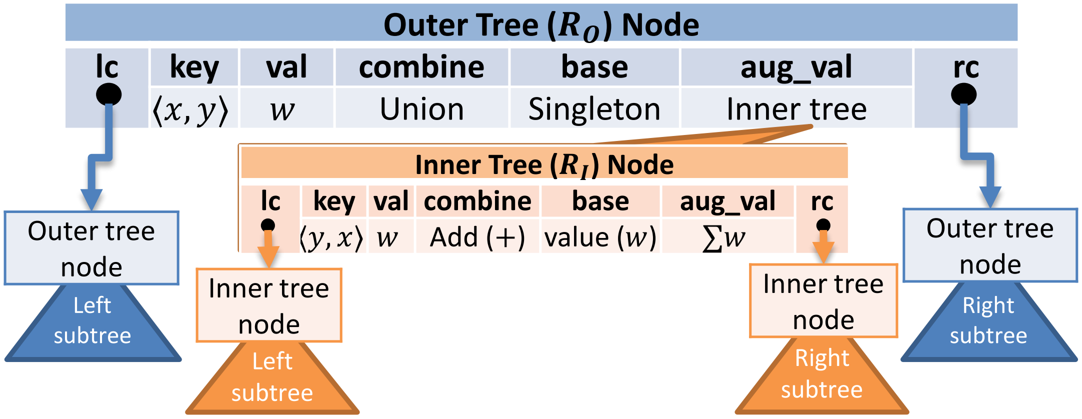


Figure 5: The range tree data structure under PAM framework. In the illustration we omit some attributes such as size, reference counter and the `identity` function. Note that the functions `combine`, `base` and `identity` of both the outer tree and inner tree are static functions, so these functions shown in this figure actually do not take any real spaces.

of the document itself. Using such a representation, conjunctions and disjunctions on terms in the index can be found by taking the intersection and union, respectively, of the corresponding maps. Weights are combined when taking unions and intersections. It is often useful to only report the k results with highest weight, as a search engine would list on its first page.

This can be represented rather directly in our interface. The inner map, maps document-ids (D) to weights (W) and uses maximum as the augmenting function f. The outer map maps terms (T) to inner maps, and has no augmentation. This corresponds to the maps:

$$M_I = \mathbb{AM}\,(\,D, <_D, W,\ W, (k, v) \to v, \max_W, 0\,)$$

$$M_O = \ \mathbb{M}\,(\,T, <_T\ M_I,)$$

We use $\mathbb{M}(K, <_K, V)$ to represent a plain map with key type K, total ordering defined by $<_K$ and value type V. In the implementation, we use the feature of PAM that allows passing a combining function with UNION and INTERSECT (see Section 4), for combining weights. The AUGFILTER function can be used to select the k best results after taking unions and intersections over terms. Note that an important feature is that the UNION function can take time that is much less that the size of the output (e.g., see Section 4). Therefore using augmentation can significantly reduce the cost of finding the top k relative to naively checking all the output to pick out the k best. The C++ code for our implementation is under 50 lines.

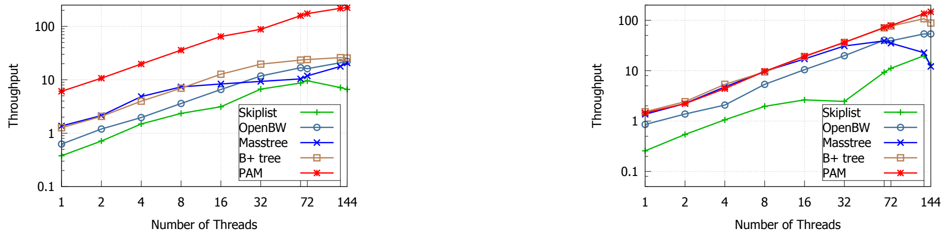
6 Experiments

For the experiments we use a 72-core Dell R930 with 4 x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache), and 1Tbyte memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. Our code was compiled using g++ 5.4.1, which supports the Cilk Plus extensions. Of these we only use `cilk_spawn` and `cilk_sync` for fork-join, and `cilk_for` as a parallel loop. We compile with `-O2`. We use `numactl -i all` in all experiments with more than one thread. It evenly spreads the memory pages across the processors in a round-robin fashion.

We ran experiments that measure performance of our four applications: the augmented sum (or max), the interval tree, the 2D range tree and the word index searching.

(a). 5×10^7 insertions, throughput (M/s), $p = 144$. Compare to some concurrent data structures.

(b). 10^7 concurrent reads, throughput (M/s), $p = 144$. Compare to some concurrent data structures.



(c). $n = 10^8$, running time. PAM on different input sizes.

(d). $n = 10^8$, speedup. The interval tree using PAM.

(e). $n = 10^8$, sequential building time. The range tree. Compare to CGAL.

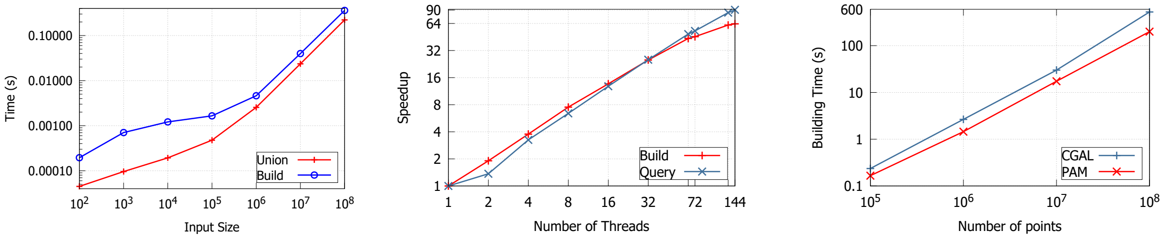


Figure 6: (a), (b) The performance (throughput, millions of elements per second) of PAM comparing with some concurrent data structures. In (a) we use our MULTIINSERT, which is not as general as the concurrent insertions in other implementations. (c) The running time of UNION and BUILD using PAM on different input sizes. (d) The speedup on interval tree construction and query. (e) The running time on range tree construction.

6.1 The Augmented Sum

Times for set functions such as union using the same algorithm in PAM without augmentation have been summarized in [12]. In this section we summarize times for a simple augmentation, which just adds values as the augmented value (see Equation 1). We test the performance of multiple functions on this structure. We also compare PAM with some sequential and parallel libraries, as well as some concurrent data structures. None of the other implementations support augmentation. We use 64-bit integer keys and values. The results on running time are summarized in Table 3. Our times include the cost of any necessary garbage collection (GC). We also present space usage in Table 4.

We test versions both with and without augmentation. For general map functions like UNION or INSERT, maintaining the augmented value in each node costs overhead, but it seems to be minimal in running time (within 10%). This is likely because the time is dominated by the number of cache misses, which is hardly affected by maintaining the augmented value. The overhead of space in maintaining the augmented value is 20% in each tree node (extra 8 bytes for the augmented value). For the functions related to the range sum, the augmentation is necessary for theoretical efficiency, and greatly improves the performance. For example, the AUGRANGE function using a plain (non-augmented) tree structure would require scanning all entries in the range, so the running time is proportional to the number of related entries. It costs 0.44s to process 10^4 parallel AUGRANGE queries. With augmentation, AUGRANGE has performance that is close to a simple FIND function, which is only 3.04s for 10^8 queries. Another example to show the advantage of augmentation is the AUGFILTER function. Here we use MAX instead of taking the sum as the combine function, and set the filter function as selecting all entries with values that are larger than some threshold θ. We set the parameter m as the output size, which can be adjusted by choosing appropriate θ. Such an algorithm has theoretical work of $O(m\log(n/m+1))$, and is significantly more efficient than a plain implementation (linear work) when $m \ll n$. We give two examples of tests on $m = 10^5$ and 10^6. The change of output size does not affect the running time of the non-augmented version, which is about 2.6s sequentially and 0.05s in parallel. When making use of the augmentation, we get a 3x improvement when $m = 10^6$ and about 14x improvement when $m = 10^5$.

For sequential performance we compare to the C++ Standard Template Library (STL) [45], which supports `set_union` on sets based on red-black trees and sorted vectors (arrays). We denote the two versions as *Union-Tree* and *Union-Array*. In Union-Tree results are inserted into a new tree, so it is also persistent. When the two sets have the same size, the array implementation is faster because of its flat structure and better cache performance. If one map is much smaller, PAM performs better than Union-Array because of better theoretical bound ($O(m\log(n/m+1))$ vs. $O(n+m)$). It outperforms Union-Tree because it supports persistence more efficiently, i.e., sharing nodes instead of making a copy of all output entries. Also, our JOIN-based INSERT achieves performance close to (about 17% slower) the well-optimized STL tree insertion even though PAM needs to maintain the reference counts.

In parallel, the speedup on the aggregate functions such as UNION and BUILD is above 50. Generally, the speedup is correlated to the ratio of reads to writes. With all (or mostly) reads to the tree structure, the speedup is often more than 72 (number of cores) with hyperthreads (e.g., FIND, AUGLEFT and AUGRANGE). With mostly writes (e.g., building a new tree as output) it is 40-50 (e.g., FILTER, RANGE, UNION and AUGFILTER). The BUILD function is relatively special because the parallelism is mainly from the parallel sorting. We also give the performance of the MULTIINSERT function in the Multicore STL (MCSTL) [61] for reference. On our server MCSTL does not scale to 144 threads, and we show the best time it has (on 8-16 threads). On the functions we test, PAM outperforms MCSTL both sequentially and in parallel.

PAM is scalable to very large data, and still achieve very good speedup. On our machine, PAM can process up to 10^{10} elements (highlighted in Table 3). It takes more than half an hour to build the tree sequentially, but only needs 28 seconds in parallel, achieving a 65-fold speedup. For AUGRANGE the speedup is about 90.

Also, using path-copying to implement persistence improves space-efficiency. For the persistent UNION on two maps of size 10^8 and 10^5, we save about 49% of tree nodes because most nodes in the larger tree are re-used in the output tree. When the two trees are of the same size and the keys of both trees are extracted from the similar distribution, there is little savings.

We present the parallel running times of UNION and BUILD on different input sizes in Figure 6 (c). For UNION we set one tree of size 10^8 and vary the other tree size. When the tree size is small, the parallel running time does not shrink proportional to size (especially for BUILD), but is still reasonably small. This seems to be caused by insufficient parallelism on small sizes. When the input size is larger than 10^6, the algorithms scales very well.

We also compare with four comparison-based concurrent data structures: skiplist, OpenBw-tree [63], Masstree [42] and B+ tree [65]. The implementations are from [63][4]. We compare their concurrent insertions with our parallel MULTIINSERT and test on YCSB microbenchmark C (read-only). We first use 5×10^7 insertions to an empty tree to build the initial database, and then test 10^7 concurrent reads. The results are given in Figure 6(a) (insertions) and (b) (reads). For insertions, PAM largely outperforms all of them

---

[4] We do not compare to the fastest implementation (the Adaptive Radix Tree [38]) in [63] because it is not comparison-based.

sequentially and in parallel, although we note that their concurrent insertions are more general than our parallel multi-insert (e.g., they can deal with ongoing deletions at the same time). For concurrent reads, PAM performs similarly to B+ tree and Masstree with less than 72 cores, but outperforms all of them on all 144 threads. We also compare to Intel TBB [54, 62] concurrent hash map, which is a parallel implementation on *unordered* maps. On inserting $n = 10^8$ entries into a pre-allocated table of appropriate size, it takes 0.883s compared to our 0.323s (using all 144 threads).

6.2 Interval Trees

We test our interval tree (same code as in Figure 3) using the PAM library. For queries we test 10^9 stabbing queries. We give the results of our interval tree on 10^8 intervals in Table 5 and the speedup figure in Figure 6(d).

Sequentially, even on 10^8 intervals the tree construction only takes 14 seconds, and each query takes around 0.58 μs. We did not find any comparable open-source interval-tree library in C++ to compare with. The only available library is a Python interval tree implementation [30], which is sequential, and is very inefficient (about 1000 times slower sequentially). Although unfair to compare performance of C++ to python (python is optimized for ease of programming and not performance), our interval tree is much simpler than the python code—30 lines in Figure 3 vs. over 2000 lines of python. This does not include our code in PAM (about 4000 lines of code), but the point is that our library can be shared among many applications while the Python library is specific for the interval query. Also our code supports parallelism.

In parallel, on 10^8 intervals, our code can build an interval tree in about 0.23 second, achieving a 63-fold speedup. We also give the speedup of our PAM interval tree in Figure 6(d). Both construction and queries scale up to 144 threads (72 cores with hyperthreads).

6.3 Range Trees

We test our 2D range tree as described in Section 5.2. A summary of run times is presented in Table 5. We compared our sequential version with the range tree in the CGAL library [51] (see Figure 6(e)). The CGAL range tree is sequential and can only report all the points in the range. For 10^8 input points we control the output size of the query to be around 10^6 on average. Table 5 gives results of construction and query time using PAM and CGAL respectively. Note that our range tree also stores the weight and reference counting in the tree while CGAL only stores the coordinates, which means CGAL version is more space-efficient than ours. Even considering this, PAM is always more efficient than CGAL and less than half the running time both in building and querying time on 10^8 points. Also our code can answer the weight-sum in the window in a much shorter time, while CGAL can only give the full list.

We then look at the parallel performance. As shown in Table 5 it took 3 seconds (about a 64-fold speedup) to build a tree on 10^8 points. On 144 threads the PAM range tree can process 1.82 million queries on weight-sum per second, achieving a 87-fold speedup.

We also report the number of allocated tree nodes in Table 4. Because of path-copying, we save 13.8% space by the sharing of inner tree nodes.

6.4 Word Index Searching

To test the performance of the inverted index data structure described in Section 5.3, we use the publicly available Wikipedia database [64] (dumped on Oct. 1, 2016) consisting of 8.13 million documents. We removed all XML markup, treated everything other than alphanumeric characters as separators, and converted all upper case to lower case to make searches case-insensitive. This leaves 1.96 billion total words with 5.09 million unique words. We assigned a random weight to each word in each document (the values of the weights make no difference to the runtime). We measure the performance of building the index from an array of `(word, doc_id, weight)` triples, and the performance of queries that take an intersection (logical-and) followed by selecting the top 10 documents by weight.

Unfortunately we could not find a publicly available C++ version of inverted indices to compare to that support and/or queries and weights although there exist benchmarks supporting plain searching on a single word [17]. However the experiments do demonstrate speedup numbers, which are interesting in this application since it is the only one which does concurrent updates. In particular each query does its own intersection over the shared posting lists to create new lists (e.g., multiple users are searching at the same time). Timings are shown in Table 6. Our implementation can build the index for Wikipedia in 13 seconds, and can answer 100K queries with a total of close to 200 billion documents across the queries in under 5 seconds. It demonstrates that good speedup (77x) can be achieved for the concurrent updates in the query.

7 Conclusion

In this paper we introduce the *augmented map*, and describe an interface and efficient algorithms to for it. Based on the interface and algorithms we develop a library supporting the augmented map interface called PAM, which is parallel, work-efficient, and supports persistence. We also give four example applications that can be adapted to the abstraction of augmented maps, including the augmented sum, interval trees, 2D range trees and the inverted indices. We implemented all these applications with the PAM library. Experiments show that the functions in our PAM implementation are efficient both sequentially and in parallel. The code of the applications implemented with PAM outperforms some existing libraries and implementations, and also achieves good parallelism. Without any specific optimizations, the speedup is about more than 60 for both building interval trees and building range trees, and 82 for building word index trees on 72 cores. For parallel queries the speedup is always over 70.

References

[1] Georgy Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Proc. of the USSR Academy of Sciences*, 145:263–266, 1962. In Russian, English translation by Myron J. Ricci in Soviet Doklady, 3:1259-1263, 1962.

[2] Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Proc. ACM SIGMOD-SIGACT-SIGAI Symp. on Principles of Database Systems (PODS)*, pages 429–440, 2016.

[3] Yaroslav Akhremtsev and Peter Sanders. Fast parallel operations on search trees. *arXiv preprint arXiv:1510.05433*, 2015.

[4] Ahmed M. Aly, Hazem Elmeleegy, Yan Qi, and Walid Aref. Kangaroo: Workload-aware processing of range data and range queries in hadoop. In *Proc. ACM International Conference on Web Search and Data Mining (WSDM)*, pages 397–406, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3716-8.

[5] Flurry analytics, -. URL `https://developer.yahoo.com/flurry/docs/analytics/`.

[6] Antonio Barbuzzi, Pietro Michiardi, Ernst Biersack, and Gennaro Boggia. Parallel bulk insertion for large-scale analytics applications. In *Proc. International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[7] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, pages 357–369, 2017.

[8] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[9] Jon Louis Bentley. Decomposable searching problems. *Information processing letters*, 8(5):244–251, 1979.

[10] Juan Besa and Yadran Eterovic. A concurrent red-black tree. *J. Parallel Distrib. Comput.*, 73(4): 434–449, 2013.

[11] Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 16–26, 1998.

[12] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2016.

[13] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, 2010.

[14] Trevor Brown and Hillel Avni. *Range Queries in Non-blocking k-ary Search Trees*, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35476-2.

[15] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 329–342, 2014.

[16] Chee Yong Chan and Yannis E. Ioannidis. Hierarchical prefix cubes for range-sum queries. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 675–686, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-615-7.

[17] Rosetta Code. Inverted index, -. URL `https://rosettacode.org/wiki/Inverted_index`.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (second edition)*. MIT Press and McGraw-Hill, 2001.

[19] Costin S. C# based interval tree. `https://code.google.com/archive/p/intervaltree`, 2012.

[20] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *Proc. of the VLDB Endowment*, 4(4):255–266, 2011.

[21] Dana Drachsler, Martin T. Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 343–356, 2014.

[22] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proc. ACM Symp. on Theory of computing (STOC)*, pages 109–121, 1986.

[23] Herbert Edelsbrunner. Dynamic rectangle intersrction searching. Technical Report Institute for Technical Processing Report 47, Technical University of Graz, Austria, 1980.

[24] Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. In *Experimental Algorithms*, pages 111–122. Springer, 2014.

[25] Preparata Franco and Michael Ian Preparata Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 1985.

[26] Leonor Frias and Johannes Singler. Parallelization of bulk operations for STL dictionaries. In *Euro-Par 2007 Workshops: Parallel Processing, HPPC 2007, UNICORE Summit 2007, and VHPC 2007*, pages 49–58, 2007.

[27] Hong Gao and Jian-Zhong Li. Parallel data cube storage structure for range sum queries and dynamic updates. *J. Comput. Sci. Technol.*, 20(3):345–356, May 2005. ISSN 1000-9000.

[28] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, Mar 1997.

[29] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 47–57, New York, NY, USA, 1984. ACM. ISBN 0-89791-128-8.

[30] Chaim-Leib Halbet and Konstantin Tretyakov. Python based interval tree. `https://github.com/chaimleib/intervaltree`, 2015.

[31] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in olap data cubes. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 73–88, New York, NY, USA, 1997. ACM. ISBN 0-89791-911-4.

[32] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proc. of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 351–363, 1986.

[33] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proc. of the VLDB Endowment*, 8(3):293–304, 2014.

[34] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. Parallel multi-dimensional range query processing with r-trees on gpu. *Journal of Parallel and Distributed Computing*, 73(8):1195 – 1207, 2013. ISSN 0743-7315.

[35] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 407–418, 2000.

[36] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[37] Kim S. Larsen. AVL trees with relaxed balance. *J. Comput. Syst. Sci.*, 61(3):508–522, 2000.

[38] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.

[39] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3): 244–257, 2010.

[40] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 302–313, 2013. ISBN 978-1-4673-4909-3.

[41] LevelDB, -. URL `leveldb.org`.

[42] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proc. of the 7th ACM European Conference on Computer Systems*, pages 183–196. ACM, 2012.

[43] Mark De Berg Mark, Mark Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry*. Springer, 2000.

[44] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

[45] David R Musser, Gillmer J Derge, and Atul Saini. *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Professional, 2009.

[46] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, 2014.

[47] Gabriele Neyer. dD range and segment trees. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.10 edition, 2017. URL `http://doc.cgal.org/4.10/Manual/packages.html#PkgRangeSegmentTreesDSummary`.

[48] Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2 (1):33–43, 1973.

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

[50] Oracle. Oracle nosql, -. URL `https://www.oracle.com/database/nosql/index.html`.

[51] Mark H Overmars. Designing the computational geometry algorithms library cgal. In *Applied computational geometry towards geometric engineering*, pages 53–58. Springer, 1996.

[52] Heejin Park and Kunsoo Park. Parallel algorithms for red–black trees. *Theoretical Computer Science*, 262(1):415–435, 2001.

[53] Wolfgang J. Paul, Uzi Vishkin, and Hubert Wagener. Parallel dictionaries in 2-3 trees. In *Proc. International Colloq. on Automata, Languages and Programming (ICALP)*, pages 597–609, 1983.

[54] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4): 298–298, 2008.

[55] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2012.

[56] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets:*. Cambridge University Press, 10 2011.

[57] RocksDB, -. URL `rockdb.org`.

[58] Hanan Samet. *The design and analysis of spatial data structures*, volume 199. Addison-Wesley Reading, MA, 1990.

[59] Raimund Seidel and Celcilia R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[60] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, August 2013. ISSN 2150-8097.

[61] Johannes Singler, Peter Sanders, and Felix Putze. MCSTL: the multi-core standard template library. In *Proc. European Conf. on Parallel Processing (Euro-Par)*, pages 682–694, 2007.

[62] TBB. Intel® Threading Building Blocks 2.0 for Open Source, -. `https://www.threadingbuildingblocks.org/`.

[63] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words.

[64] Wikimedia Foundation. Wikepedia:database download. `https://en.wikipedia.org/wiki/Wikipedia:Database_download`, 2016.

[65] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proc. of the 2016 International Conference on Management of Data*, pages 1567–1581. ACM, 2016.

[66] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006. ISSN 0360-0300.

	n	**m**	$\mathbf{T_1}$	$\mathbf{T_{144}}$	**Spd.**
PAM (with augmentation)					
Union	10^8	10^8	12.517	0.2369	52.8
Union	10^8	10^5	0.257	0.0046	55.9
Find	10^8	10^8	113.941	1.1923	95.6
Insert	10^8	–	205.970	–	–
Build	10^8	–	16.089	0.3232	49.8
Build	$\mathbf{10^{10}}$	–	**1844.38**	**28.24**	**65.3**
Filter	10^8	–	4.578	0.0804	56.9
Multi-Insert	10^8	10^8	23.797	0.4528	52.6
Multi-Insert	10^8	10^5	0.407	0.0071	57.3
Range	10^8	10^8	44.995	0.8033	56.0
AugLeft	10^8	10^8	106.096	1.2133	87.4
AugRange	10^8	10^8	193.229	2.1966	88.0
AugRange	$\mathbf{10^{10}}$	$\mathbf{10^8}$	**271.09**	**3.04**	**89.2**
AugFilter	10^8	10^6	0.807	0.0163	49.7
AugFilter	10^8	10^5	0.185	0.0030	61.2
Non-augmented PAM (general map functions)					
Union	10^8	10^8	11.734	0.1967	59.7
Insert	10^8	–	186.649	–	–
build	10^8	–	15.782	0.3008	52.5
Range	10^8	10^8	42.756	0.7603	56.2
Non-augmented PAM (augmented functions)					
AugRange	10^8	10^4	21.642	0.4368	49.5
AugFilter	10^8	10^6	2.695	0.0484	55.7
AugFilter	10^8	10^5	2.598	0.0497	52.3
STL					
Union Tree	10^8	10^8	166.055	–	–
Union Tree	10^8	10^5	82.514	–	–
Union Array	10^8	10^8	1.033	–	–
Union Array	10^8	10^5	0.459	–	–
Insert	10^8	–	158.251	–	–
MCSTL					
Multi-Insert	10^8	10^8	51.71	7.972	6.48
Multi-Insert	10^8	10^5	0.20	0.027	7.36

Table 3: Timings in seconds for various functions in PAM, the C++ Standard Template Library (STL) and the library Multi-core STL (MCSTL) [61]. 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. “Spd.” means the speedup (i.e., T_1/T_{144}). For insertion we test the total time of n insertions in turn starting from an empty tree. All other libraries except PAM are not augmented.

Func.	**Type**	**Overhead for aug.**			**Saving from node-sharing**		
		node size	**aug. size**	**over-head**	**#nodes in theory**	**Actual #nodes**	**Saving ratio**
Union	$\mathbf{m = 10^8}$	48B	8B	20%	390M	386M	1.2%
	$\mathbf{m = 10^5}$	48B	8B	20%	200M	102M	49.0%
Range	**Outer**	48B	8B	20%	100M	100M	0.0%
Tree	**Inner**	40B	4B	11%	266M	229M	13.8%

Table 4: Space used by the UNION function and the range tree application. We use B for byte, M for million.

Lib.	**Func.**	**n**	**m**	$\mathbf{T_1}$	$\mathbf{T_{144}}$	**Spd.**
PAM (interval)	**Build**	10^8	-	14.35	0.227	63.2
	Query	10^8	10^8	53.35	0.576	92.7
PAM (range)	**Build**	10^8	-	197.47	3.098	63.7
	Q-Sum	10^8	10^6	48.13	0.550	87.5
	Q-All	10^8	10^3	44.40	0.687	64.6
CGAL (range)	**Build**	10^8	-	525.94	-	-
	Q-All	10^8	10^3	110.94	-	-

Table 5: The running time (seconds) of the range tree and the interval tree implemented with PAM interface on n points and m queries. Here "T_1" reports the sequential running time and "T_{144}" means on all 72 cores with hyperthreads (i.e., 144 threads). "Spd." means the speedup (i.e., T_1/T_{144}). "Q-Sum" and "Q-All" represent querying the sum of weights and querying the full list of all points in a certain range respectively. We give the result on CGAL range tree for comparisons with our range tree.

	n ($\times\mathbf{10^9}$)	**1 Core**		$\mathbf{72^*}$ **Cores**		**Speed-up**
		Time (secs)	**Melts /sec**	**Time (secs)**	**Gelts /sec**	
Build	1.96	1038	1.89	12.6	0.156	82.3
Queries	177	368	480.98	4.74	37.34	77.6

Table 6: The running time and rates for building and queering an inverted index. Here "one core" reports the sequential running time and "$\mathbf{72}^*$ cores" means on all 72 cores with hyperthreads (i.e., 144 threads). Gelts/sec calculated as $n/(\text{time} \times 10^9)$.