Parallel-batched Interpolation Search Tree

Vitaly Aksenov[1], Ilya Kokorin[1,2], and Alena Martsenyuk

[1] ITMO University, Russia
[2] vk.com, Russia

Abstract. A sorted set (or map) is one of the most used data types in computer science. In addition to standard set operations, like `Insert`, `Remove`, and `Contains`, it can provide set-set operations such as `Union`, `Intersection`, and `Difference`. Each of these set-set operations is equivalent to some batched operation: the data structure should be able to execute `Insert`, `Remove`, and `Contains` on a batch of keys. It is obvious that we want these "large" operations to be parallelized. These sets are usually implemented with the trees of logarithmic height, such as 2-3 trees, treaps, AVL trees, red-black trees, etc. Until now, little attention was devoted to parallelizing data structures that work asymptotically better under several restrictions on the stored data. In this work, we parallelize Interpolation Search Tree which is expected to serve requests from a *smooth* distribution in doubly-logarithmic time. Our data structure of size n performs a batch of m operations in $O(m \log \log n)$ work and poly-log span.

Keywords: Parallel Programming · Data Structures · Parallel-Batched Data Structures.

1 Introduction

A *Sorted set* is one of the most ubiquitous *Abstract Data Types* in Computer Science, supporting `Insert`, `Remove`, and `Contains` operations among many others. The sorted set can be implemented using different data structures: to name a few, skip-lists [21], red-black trees [11], splay trees [22], or B-trees [10, 9].

Since nowadays most of the processors have multiple cores, we are interested in parallelizing these data structures. There are two ways to do that: write a concurrent version of a data structure or allow one to execute a batch of operations in parallel. The first approach is typically very hard to implement correctly and efficiently due to problems with synchronization. Thus, in this work we are interested in the second approach: *parallel-batched data structures*.

Several parallel-batched data structures implementing a sorted set are presented: for example, 2-3 trees [18], red-black trees [17], treaps [6], (a, b) trees [2], AVL-trees [15], and generic joinable binary search trees [5, 23].

Although many parallel-batched trees were presented, we definitely lack implementations that can execute separate queries in $o(\log n)$ time under some assumptions. However, there exist at least one sequential data structures with this property — *Interpolation Search Tree*, or *IST*.

Despite the fact that concurrent IST is already presented [8, 20] we still lack its parallel-batched version: it differs much from the concurrent version since it

allows many processes to execute scalar requests simultaneously, while we use the multiprocessing to parallelize large non-scalar requests.

The work is structured as follows: in Section 2 we describe the important preliminaries; in Section 3 we present the original Interpolation Search Tree; in Sections 4, 5, and 6 we present the parallel-batched contains, insert and remove algorithms; in Section 7 we present a parallelizable method to keep the IST balanced; in Section 8 we present a theoretical analysis; in Section 9 we discuss the implementation and present experimental results; we conclude in Section 10. The full version of the paper appears at [3].

2 Preliminaries

2.1 Parallel-batched data structures

Definition 1. Consider a data structure D storing a set of keys and an operation Op. If Op involves only one key (e.g., it checks whether a single key exists in the set, or inserts a single key into the set) it is called a *scalar operation*. Otherwise, i.e., if Op involves multiple keys, it is called a *batched operation*.

A data structure D that supports at least one *batched operation* is called a *batched data structure*.

We want the following batched operations from a sorted set:

- `Set.ContainsBatched(keys[])` — the operation takes an array of keys of size m and returns an array `Result` of size m. For each i, `Result[i]` is `true` if `keys[i]` exist in the set, and `false` otherwise.
- `Set.InsertBatched(keys[])` — the operation takes an array of keys of size m. If `keys[i]` does not exist in the set, the operation adds it to the set.
- `Set.RemoveBatched(keys[])` — the operation takes an array of keys of size m. If `keys[i]` exists in the set, the operation removes it from the set.

Note, that: 1) `InsertBatched` calculates the union of two sets; 2) `RemoveBatched` calculates the difference of two sets; and 3) `ContainsBatched` calculates the intersection of two sets.

We can employ parallel programming techniques (e.g., *fork-join parallelism* [7, 14]) to execute batched operations faster.

Definition 2. A batched data structure D that uses parallel programming to speed up batched operation execution is called a *parallel-batched data structure*.

2.2 Time complexity model

In our work, we assume the standard *work-span* complexity model [1] for *fork-join* computations. We model each computation as a *directed acyclic graph*, where nodes represent operations and edges represent dependencies between them. This graph has exactly one *source node* (i.e., the start of the execution with zero incoming edges) and exactly one *sink node* (i.e., the end of the execution with zero outcoming edges). Some operations have two outcoming edges — they spawn two parallel tasks and are called *fork operations*. Some operations have two incoming edges — they wait for two corresponding parallel tasks to complete and are called *join operations*.

Considering the execution graph of the algorithm, our target complexities are: 1) *work* denotes the number of nodes in the graph, i.e., the total number of operations executed; 2) *span* denotes the number of nodes on the longest path from *source* to *sink*, i.e., the length of the critical path in the graph.

2.3 Standard parallel primitives

In this work, we use several standard parallel primitives. Now, we give their descriptions. Their implementations are provided, for example, in [12].

Parallel loop. It executes a loop body for `n` index values (from `0` to `n-1`, inclusive) in parallel. This operation costs $O(n)$ work and $O(\log n)$ span given that the body has time complexity $O(1)$.

Scan. `Result := Scan(Arr)` calculates *exclusive* prefix sums of array `Arr` such that $Result[i] = \sum_{j=0}^{i-1} Arr[j]$. Scan has $O(n)$ work and $O(\log n)$ span.

Filter. `Filter(Arr, predicate)` returns an array, consisting of elements of the given array `Arr` satisfying `predicate` keeping the order. `Filter` has $O(n)$ work and $O(\log n)$ span given that `predicate` has time complexity $O(1)$.

Merge. `Merge(A, B)` merges two sorted arrays `A` and `B` keeping the result sorted. It has $O(|A| + |B|)$ work and $O(\log^2(|A| + |B|))$ span.

Difference. `Difference(A, B)` takes two sorted arrays `A` and `B` and returns all elements from `A` that are not present in `B`, in sorted order. It takes $O(|A|+|B|)$ work and $O(\log^2(|A| + |B|))$ span.

Rank. Given that `A` is a sorted array and `x` is a value, we denote `ElemRank(A, x)` $= |\{e \in A | e \leq x\}|$ as the number of elements in `A` that are less than or equal to `x`. Given that `A` and `B` are sorted arrays, we denote `Rank(A, B) = [r0, r1, ..., r|B|-1]`, where `ri = ElemRank(A, B[i])`. `Rank` operation can be computed in $O(|A| + |B|)$ work and $O(\log^2(|A| + |B|))$ span.

3 Interpolation Search Tree

3.1 Interpolation Search Tree Definition

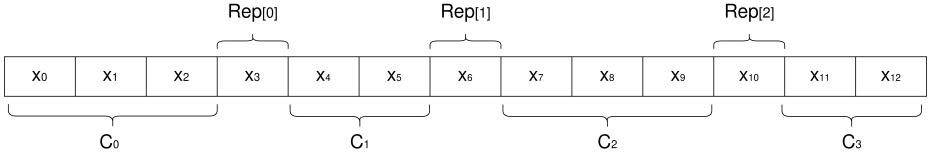
Interpolation Search Tree (IST) is a multiway internal search tree proposed in [16]. IST for a set of keys $x_0 < x_1 < \ldots < x_{n-1}$ can be either *leaf* or *non-leaf*.

Definition 3. Leaf *IST with a set of keys* $x_0 < x_1 < \ldots < x_{n-1}$ *consists of array* `Rep` *with* $Rep[i] = x_i$*, i.e., it keeps all the keys in this sorted array.*

Definition 4. Non-leaf *IST with a set of keys* $x_0 < x_1 < \ldots < x_{n-1}$ *consists of two parts (Fig. 1 and 2):*

- *An array* `Rep` *storing an ordered subset of keys* $x_{i_0}, x_{i_1}, \ldots x_{i_{k-1}}$.
- *Child ISTs* $C_0, C_1 \ldots C_k$*: 1)* C_0 *is an IST storing a subset of keys* $x_0, x_1 \ldots x_{i_0-1}$*; 2) for* $1 \leq j \leq k-1$*,* C_j *is an IST storing a subset of keys* $x_{i_{j-1}+1}, \ldots x_{i_j-1}$*; and 3)* C_k *is an IST storing a subset of keys* $x_{i_{k-1}+1}, \ldots x_{n-1}$*;*

Fig. 1: Example of a non-leaf IST. $Rep[0] = x_3, Rep[1] = x_6, Rep[2] = x_{10}$. C_0 stores keys $x_0 \ldots x_2$, C_2 stores keys $x_4 \ldots x_5$, C_3 stores keys $x_7 \ldots x_9$, C_4 stores keys $x_{11} \ldots x_{12}$.



Any *non-leaf* IST has the following properties: 1) all keys less than $Rep[0]$ are located in C_0; 2) all keys in between $Rep[j-1]$ and $Rep[j]$ are located in C_j, and, finally, 3) all keys greater than $Rep[k-1]$ are located in C_k.

3.2 Interpolation search and the lightweight index

We can optimize operations on ISTs with numeric keys, by leveraging the *interpolation search technique* [19, 16, 24]. Each node of an IST has an index that can point to some place in the `Rep` array close to the position of the key being searched. This approach is named *interpolation search*. The structure of a non-leaf IST with an index is shown in Fig. 3.

In the original IST, the index uses an array `ID` of size $m \in \Theta(n^\varepsilon)$ with some $\varepsilon \in [\frac{1}{2}; 1)$. `ID[i] = j` iff $Rep[j] < a + i \cdot \frac{b-a}{m} \le Rep[j+1]$ where `a` and `b` are the lower and upper bounds on the values. In [16], `ID[`$\lfloor \frac{x-a}{b-a} \cdot m \rfloor$`]` is the approximate position of `x` in `Rep`.

After finding the approximate location of `x` in `Rep`, we can find its exact location by using the linear search, as described in [16]. Let us denote `i := ID[`$\lfloor \frac{x-a}{b-a} \cdot m \rfloor$`]`. If i points to the right place — we stop. Otherwise, we go in the proper direction: to the right of i (Fig. 4a) or to the left of i (Fig. 4b).

Fig. 2: Example of an IST built on array in Fig. 1.

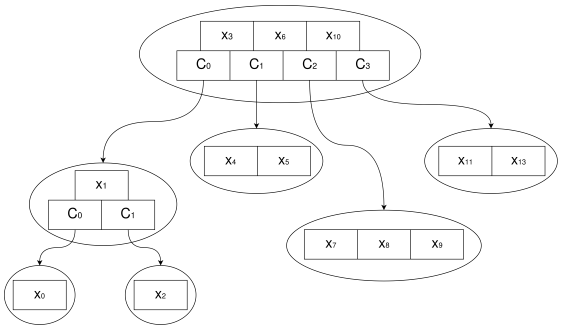
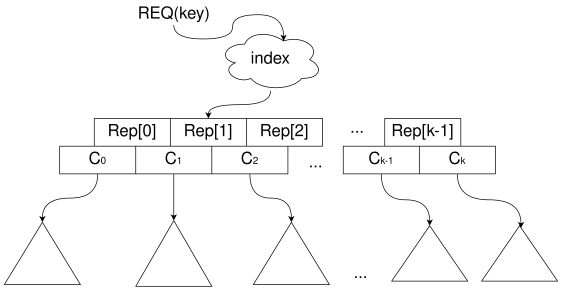


Fig. 3: Non-leaf IST contains: (1) `Rep` array; (2) an array of pointers to child ISTs `C`; (3) an index, allowing for fast lookups of keys in the `Rep` array.



Note, we can use more complex techniques instead of the linear search, e.g., exponential search [4]. However, they are often unnecessary, since the index usually provides an approximation good enough to finish the search only in a couple of operations. Also, we can use other index structures, e.g., a machine learning model [13].

3.3 Search in IST

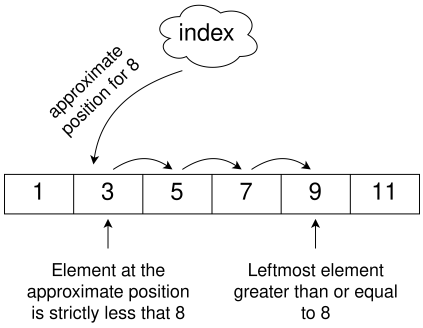
Suppose we want to find a `key` in IST. The search algorithm is iterative: on each iteration we look for the `key` in a subtree of a node `v`. To look for the key in the whole IST we begin the algorithm by setting `v := IST.Root`.

To find `key` in `v`, we do the following (`k` is the length of `v.Rep`):

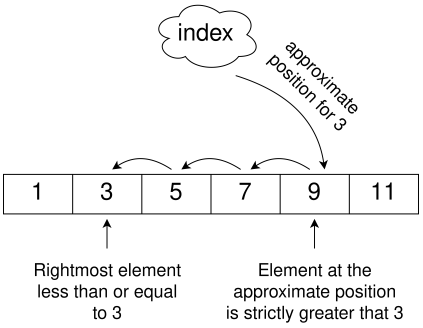
1. If `v` is empty, we conclude that `key` is not there;
2. If `key` is found in `v.Rep` array, then, we found the key;
3. If `key < v.Rep[0]`, the `key` can be found only in `v.C[0]` subtree. Thus, we set `v` $\leftarrow$ `v.C[0]` and continue the search;

Fig. 4: Determining the exact location of the key given the approximate location

(a) Searching for the key on the right to the approximate position.



(b) Searching for the key on the left to the approximate position.



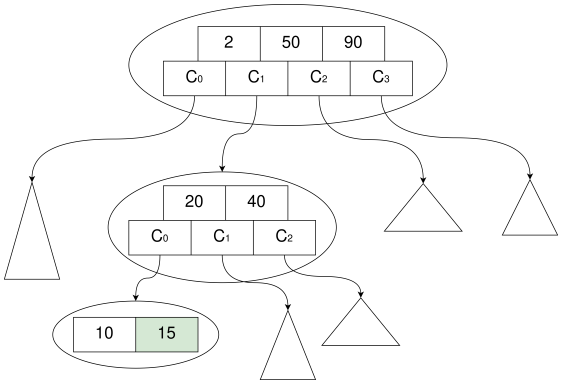
4. If `key > v.Rep[k - 1]`, the `key` can be found only in `v.C[k]` subtree. Thus, we set `v` $\leftarrow$ `v.C[k]` and continue the search;
5. Otherwise, we find `j` such that `v.Rep[j - 1] < key < v.Rep[j]`. In this case `key` can be found only in `v.C[j]`. Thus, we set `v` $\leftarrow$ `v.C[j]` and continue our search in the `j`-th child.

3.4 Executing update operations and maintaining balance

The algorithm for inserting a key into IST is very similar to the search algorithm above. To execute `Insert(key)` we do the following (Fig. 5):

1. Initialize `v := IST.Root`;
2. For the current node `v`, if `key` appears in `v.Rep` array, we finish the operation — the key already exists.
3. If `v` is a leaf and `key` does not exists in `v.Rep`, insert `key` into `v.Rep` keeping it sorted;
4. If `v` is an inner node and `key` does not exists in `v.Rep`, determine in which child the insertion should continue, set `v` $\leftarrow$ `v.C[next_child_idx]` and go to step (2).

Fig. 5: Insert `15`: proceed from the root to the second child and then to the first child.



To remove a key from IST we introduce `Exists` array in each node that shows whether the corresponding key in `Rep` is in the set or not. Thus, we just need to mark an element as removed without physically deleting it. We have to take into account such marked keys during the contains and inserts. The removal algorithm is discussed in more detail in [16].

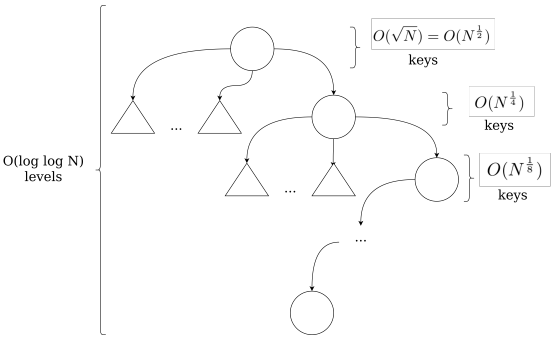
The problem with these update algorithms is that all the new keys may be inserted to a single leaf, making the IST unbalanced. In order to keep the execution time low, we should keep the tree balanced.

Definition 5. Suppose `H` is some small integer constant, e.g., `10`. An IST `T`, storing keys $x_0 < x_1 < \ldots < x_{n-1}$, is said to be *ideally balanced* if either: 1) `T` is a leaf IST and `n` $\leq$ `H`; 2) `T` is a non-leaf IST, `n > H`, and elements in `Rep` are equally spaced, $Rep[i] = x_{(i+1)\cdot\lfloor \frac{n}{k} \rfloor}$, and all child ISTs $\{C_i\}_{i=0}^{k}$ are ideally balanced.

For non-leaf IST, we aim to have the size of `Rep` as $k = \lfloor \sqrt{n} \rfloor$. Consider an ideally balanced IST storing n keys (Fig. 6). As we require, the root of IST contains $\Theta(n^{\frac{1}{2}})$ keys in its `Rep` array; any node on the second level has `Rep` array of size $\Theta(n^{\frac{1}{4}})$; generally, any node on the i-th level has `Rep` array of size $\Theta(n^{\frac{1}{2^i}})$. Thus, an ideally balanced IST with n keys has a height of $O(\log \log n)$.

In order to keep IST balanced we maintain the number of modifications (both insertions and removals) applied to each subtree `T`. When the number of modifications to `T` exceeds the initial size of `T` multiplied by some constant `C`, we rebuild `T` from scratch making it ideally balanced. This rebuilding approach has a proper amortized bounds and is adopted from papers about IST [16, 8, 20].

Fig. 6: Height of an ideal IST



3.5 Time and space complexity

Mehlhorn and Tsakalidis [16] define of a *smooth* probability distribution. For example, the uniform distribution is *smooth*. Suppose we are given μ that is *smooth*. From [16] we know that: 1) IST with n keys takes $O(n)$ space; 2) the expected amortized cost of μ-random insertion and random removal is $O(\log \log n)$; 3) the amortized insertion and removal cost is $O(\log n)$; 4) the expected search time on sets, generated by μ-random insertions and random removal, is $O(\log \log n)$; 5) the worst-case search time is $O(\log^2 n)$.

Therefore, IST can execute operations in $o(\log n)$ time under reasonable assumptions. As our goal, we want to design a parallel-batched version of the IST that processes operations asymptotically faster than known sorted set implementations (e.g., red-black trees).

4 Parallel-batched Contains

In this section, we describe the implementation of `ContainsBatched(keys[])` operation. We suppose that `keys` array is sorted. For simplicity, we assume that IST does not support removals. In Section 6, we explain how to fix it.

We implement `ContainsBatched` operation in the following way. At first, we introduce a function `BatchedTraverse(node, keys[], left, right, result[])`. The purpose of this function is to determine for each index `left` $\leq i <$ `right`, whether `keys[i]` is stored in the `node` subtree. If so, set `result[i] = true`, otherwise, `result[i] = false`. Given the operation `BatchedTraverse`, we can implement `ContainsBatched` with almost zero effort (Listing 1.1):

Listing 1.1: Implementation of `ContainsBatched` on top of `BatchedTraverse` routine

```
fun ContainsBatched(keys[]):
    result[] := [array of size |keys|]
    // search for all keys in the root subtree (i.e., in the whole IST)
    BatchedTraverse(IST.Root, keys, 0, |keys|, result)
    return result
```

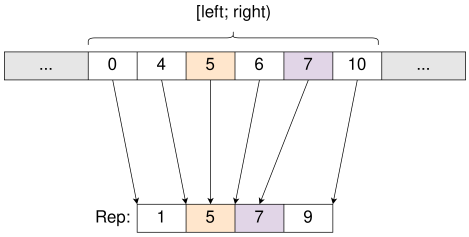
Now, we describe `BatchedTraverse(node, keys[], left, right, result[])`.

4.1 `BatchedTraverse` in a leaf node

If `node` is a leaf node, we determine for each key in `keys[left..right)` whether it exists in `node.Rep`. Since `node` is a leaf, keys cannot be found anywhere else in `node` subtree.

We may use `Rank` function to find the *rank* of each element of `keys[left..right)` in `node.Rep` and, thus, determine for each key whether it exists in `node.Rep` (Fig. 7, Listing 1.2). As presented in Section 2.3, ranks of all keys from subarray `keys[left..right)` may be computed in parallel in linear work and poly-logarithmic span.

Fig. 7: Execution of `BatchedTraverse` in an IST leaf. Here `Rank(node.Rep, keys[left..right)) = [1, 1, 2, 2, 3, 4]`.



Listing 1.2: Using `Rank` to find keys in a leaf node in parallel

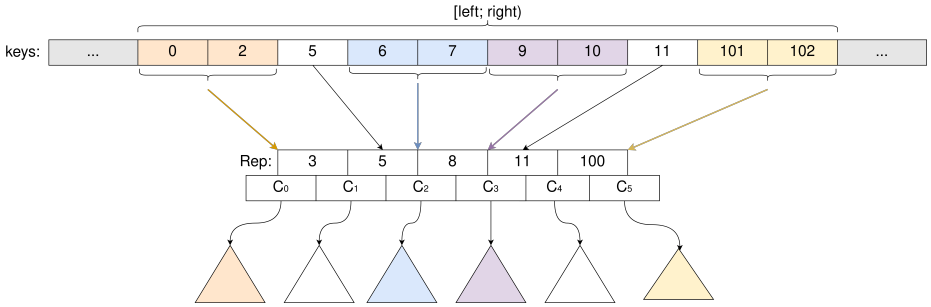
```
rank := Rank(node.Rep, keys[left..right))
pfor i in left..right:
    r := rank[i - left]
    if r = 0 or node.Rep[r - 1] ≠ keys[i]:
        result[i] ← false
    else:
        result[i] ← true
```

4.2 `BatchedTraverse` in an inner node

Consider now the `BatchedTraverse` procedure on an inner node (Fig. 8).

We begin its execution with finding the position for each key from `keys[left..right)` in `node.Rep`. We may do it using `Rank` function as in Section 4.1. However, we can also use the interpolation search (described in Section 3.2): see Listing 1.3. Denote T as an interpolation search time in `node.Rep`. As stated in Section 3.5, T is

Fig. 8: Execution of `BatchedTraverse` in an inner node of an IST.



expected to be $O(1)$. Thus, this algorithm can be executed in $O((right-left) \cdot T)$ work and polylog span in contrast to the algorithm based on the `Rank` function, that takes $O((right - left) + |node.Rep|)$ work.

Listing 1.3: Using interpolation search to find keys in IST leaf node in parallel

```
pfor i in left..right:
    idx := interpolation_search(node.Rep, keys[i])
    result[i] ← node.Rep[idx] = keys[i]
```

Some keys of the input array (e.g., `5` and `11` in Fig. 8) are found in the `Rep` array. For such keys, we set `Result[i]` to `true`. After that, all other keys can be divided into three categories:

- Keys that are strictly less than `Rep[0]` (e.g., `0` and `2` in Fig. 8) lie in `C[0]` subtree. Therefore for such keys we should continue the traversal in `C[0]`;
- Keys that are strictly greater than `Rep[k - 1]` (e.g., `100` and `101` in Fig. 8) can only be found in `C[k]`. Therefore for such keys we continue the traversal in `C[k]`.
- Keys that lie between `Rep[i]` and `Rep[i + 1]` for some $i \in [0; k-2]$ (we can find such `i` for each key using the same search technique as described above). For example, `6` and `7` for `i = 1` or `9` and `10` for `i = 2` in Fig. 8. Such keys can only be found in `C[i + 1]`. Therefore for such keys we should continue the traversal in `C[i + 1]`;

Note that some child nodes (e.g., `C[1]` and `C[4]` in Fig. 8) can not contain any key from `keys[left..right)` thus we do not continue the search in such nodes.

After determining in which child the search of each key should continue we proceed to searching for keys in children in parallel.

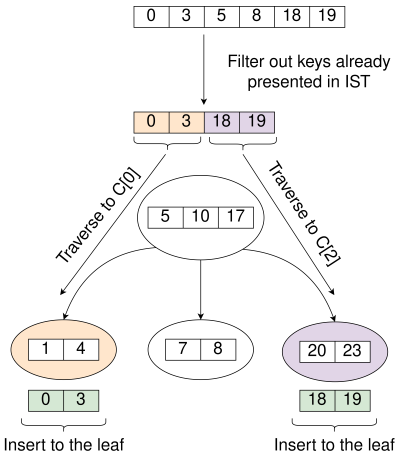
5 Parallel-batched Insert

We now consider the implementation of the operation `InsertBatched(keys[])`. Again, we suppose that array `keys[]` is sorted. For simplicity, we consider `InsertBatched` implementation on an IST without removals. In Section 6 we explain how to fix it.

We begin the insertion procedure with filtering out keys already present in the set. We can do this using the described `ContainsBatched` routine together with the `Filter` primitive: we filter out all the keys for which `ContainsBatched` returns `true`.

We implement our procedure recursively in the same way as `BatchedTraverse`. Note, that each `key` being inserted is not present in IST, thus, for each `key` our traversal finishes in some leaf (Fig. 9).

Fig. 9: Inserting a batch of keys in the IST



After we finish the traversal — we need to insert subarray $\texttt{keys[left}_i \ldots \texttt{right}_i\texttt{)}$ into some leaf $\texttt{leaf}_i$. For example, in Fig. 9 we insert `keys[0..2)` (i.e., `0` and `3`) to the leftmost leaf, while inserting `keys[2..4)` (i.e., `18` and `19`) to the rightmost leaf.

To finish the insertion, we just merge $\texttt{keys[left}_i \ldots \texttt{right}_i\texttt{)}$ with $\texttt{leaf}_i\texttt{.Rep}$ and get the new `Rep` array. Now, each target leaf $\texttt{leaf}_i$ contains all the keys that should be inserted into it.

6 Parallel-batched Remove

We now sketch the implementation of the operation `RemoveBatched(keys[])`. Again, we suppose that array `keys` is sorted.

We use the same approach as our previous algorithms. At first, we filter out the keys that do not exist in the tree. Then, we go recursively, find the keys in `Rep` arrays, and set the corresponding `Exists` cell to `false`.

Since now we have a logical removal, we should modify the implementations of `ContainsBatched` and `InsertBatched`.

During the execution of `ContainsBatched` when we encounter the `key` being searched in the `Rep` array of some node `v` (`v.Rep[i] = key`), we check `v.Exists[i]`: 1) if `v.Exists[i] = true` then `key` exists in the set; 2) otherwise, `key` does not exist in the set.

Now, we explain the updates to `InsertBatched`. As was stated in Section 5, we cannot encounter any of the key being inserted in the `Rep` array of any node of IST, since we filter out all the keys existing in IST. However, when keys can be logically removed this is not true anymore. Such keys have the corresponding entry in `v.Exists` array set to `false`, since the key does not logically exist in IST (Fig. 10a).

Suppose we are inserting `key` and we encounter it in some `v.Rep`. Thus, we can just set `v.Exists[i]` $\leftarrow$ `true` (Fig. 10b). This way the insert operation "revives" a previously removed key.

(a) Keys `1`, `10` and `23` are marked as removed

(b) Keys `10` and `23` are revived by a subsequent insert operation

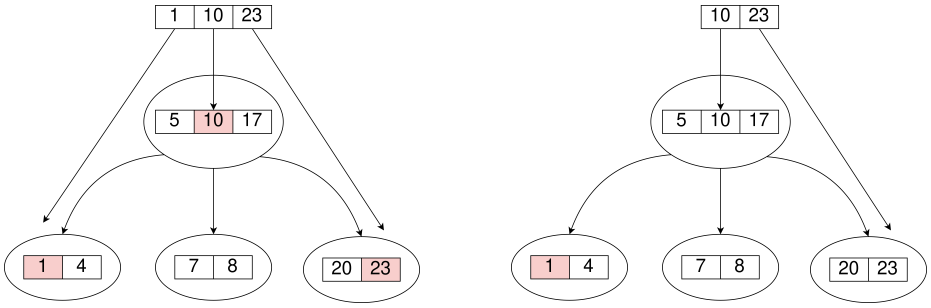


Fig. 10: Insertion of a key, that still exists in the IST physically, but is removed logically

7 Parallel tree rebuilding

7.1 Rebuilding principle

As stated in Section 3.4, we employ the lazy subtree rebuilding approach to keep IST balanced. This algorithm is adopted from papers [16, 8, 20].

For each node of IST we maintain `Mod_Cnt` — the number of modifications (successful insertions and removals) applied to that node subtree. Moreover, each node stores `Init_Subtree_Size` — the number of keys in that node subtree when the node was created.

Suppose we execute an update operation `Op` in node `v` and `Op` increases `v.Mod_Cnt` by `k` (i.e., it either inserts `k` new keys or removes `k` existing keys).

If $\texttt{v.Mod_Cnt + k} \leq \texttt{C} \cdot \texttt{v.Init_Subtree_Size}$ (where `C` is a predefined constant, e.g., `2`) we increment `v.Mod_Cnt` by `k` and continue the execution of `Op` in an ordinary way. Otherwise, we rebuild the whole subtree of `v`.

The subtree rebuilding works in the following way. At first, we *flatten* the subtree into an array: we collect all non-removed keys from the subtree to array `subtree_keys[]` in ascending order. This operation is described in more detail in Section 7.2. If the operation, that triggered the rebuilding, was `InsertBatched`, we merge the keys, we are inserting, with the keys from the `subtree_keys`. Otherwise, (that operation is `RemoveBatched`) we remove the required keys from the `subtree_keys` via the `Difference` operation (see Section 2.3 for details). Finally, we build an ideal IST `new_subtree`, containing all entries from `subtree_keys`. This operation is described in more detail in Section 7.3.

7.2 Flattening an IST into an array in parallel

First of all, we need to know how many keys are located in each node subtree. We store this number in a `Size` variable in each node and maintain it the following way: 1) when creating new node `v`, set `v.Size` to the initial number of keys in its subtree; 2) when inserting `m` new keys to `v`'s subtree, increment `v.Size` by `m`; 3) when removing `m` existing keys from `v`'s subtree, decrement `v.Size` by `m`.

To flatten the whole subtree of `node` we allocate an array `keys` of size `node.Size` where we shall store all the keys from the subtree. We implement the flattening recursively, via the `Flatten(v, keys[], left, right)` procedure, which fills subarray `keys[left..right)` with all the keys from the subtree. To flatten the whole subtree of `node` into newly-allocated array `subtree_keys` of size `node.Size` we use `Flatten(node, subtree_keys, 0, node.Size)`.

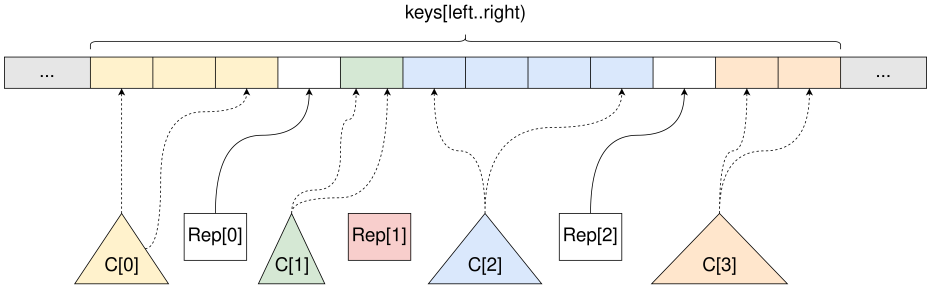
Note that non-leaf node `v` has `2k + 1` sources of keys: `v.C[i]` with `v.C[i].Size` keys and `v.Rep[i]`. `C[i]` is $2 \cdot$ `i`-th key source and `Rep[i]` is $2 \cdot$ `i` $+ 1$-th key source. Note that for a leaf node all children just contain `0` keys.

Now for each key source we must find its position in the `keys` array. To do this we calculate array `sizes` of size `2k + 1`. `i`-th source of keys stores its keys count in `sizes[i]`. After that we calculate `positions := Scan(sizes)` to find the prefix sums of sizes. After that `positions[i]` $= \sum_{j=0}^{i-1}$ `sizes[j]`. Consider now `i`-th key source. All prior key sources should fill `positions[i]` keys, thus, `i`-th key source should place its keys into the array starting from `left + positions[i]` position (Fig. 11). Therefore:

- `v.C[i]` places its keys in the `keys` array starting from `left + positions[2 · i]` by running `Flatten(v.C[i], left + positions[2 · i], left + positions[2 · i] + v.C[i].Size)`.
- If `v.Exists[i] = false` then `v.Rep[i]` should not be put in the `keys` array;
- Otherwise, `v.Exists[i] = true` and `v.Rep[i]` should be placed at `keys[left + positions[2 · i + 1]]` since `v.Rep[i]` is the `(2 · i + 1)`-th key source.

Each key source can be processed in parallel, since there are no data dependencies between them.

Fig. 11: Parallel flattening of an IST node



In Fig. 11, `C[0]` will place its keys in `keys[left .. left + 3)` subarray, `Rep[0]` will be placed in `keys[left + 3]`, `C[1]` will place its keys in `keys[left + 4 .. left + 5)` subarray, `Rep[1]` will not be placed in `keys` array since its logically removed, `C[2]` will place its keys in `keys[left + 5 .. left + 9)` subarray, `Rep[2]` will be placed in `keys[left + 9]` and `C[3]` will place its keys in `keys[left + 10 .. left + 12)` subarray.

7.3 Building an ideal IST in parallel

Suppose we have a sorted array of keys and we want to build an *ideally balanced IST* (see Section 3.4) with these keys. We implement this procedure recur-

sively via `build_IST_subarray(keys[], left, right)` procedure — it builds an ideal IST containing keys from the `keys[left..right)` subarray and returns the root of the newly-built subtree. Thus, to build a new subtree from array `keys` we just use `new_root := build_IST_subarray(keys[], 0, |keys|)`.

If the size of the subarray (i.e., `right - left`) is less than a predefined constant H, we return a leaf node with all the keys from `keys[left..right)` in `Rep` array.

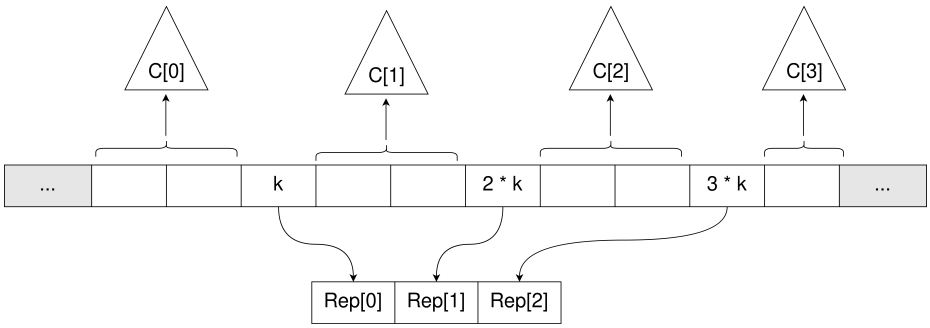
Otherwise (i.e., if `right - left` is big enough), we have to build non-leaf node. Let us denote `m := right - left; k :=` $\lfloor\sqrt{m}\rfloor - 1$. As follows from Definition 5, `Rep` array should have size $\Theta(\sqrt{m})$ and its elements must be equally spaced keys of the initial array. Thus, we copy each k-th key (k-th, $2 \cdot k$-th, etc.) into array `Rep`. Note, our subarray begins at position `left` of the initial array, since we are building IST from the subarray `keys[left..right)`. Thus, we copy `keys[left + (i + 1) · k]` into `Rep[i]`. All the copying can be done in parallel since there are no data dependencies. This way we obtain `Rep` array of size $\Theta(\sqrt{m})$ filled with equally-spaced keys of the initial subarray (Fig. 12).

Now we should build the children of the newly-created node (Fig. 12):

- `Rep[0] = keys[left + k]`. Thus, all keys less than `keys[left + k]` will be stored in `C[0]` subtree: `C[0]` ← `build_IST_subarray(keys[], left, left + k)`;
- for $1 \le i \le k-2$, `Rep[i - 1] = keys[left + i · k]` and `Rep[i] = keys[left + (i + 1) · k]`. Thus, all keys x such that `Rep[i-1] < x < Rep[i]` should be stored in `C[i]` subtree. Since `keys` array is sorted, `C[i]` must be built from the subarray `keys[left + i · k + 1..left + (i + 1) · k)`, thus, `C[i]` ← `build_IST_subarray(keys[], left + i · k + 1, left + (i + 1) · k)`;
- `Rep[k - 1] = keys[left +` k^2`]`. Thus, all keys greater than `keys[left +` k^2`]` are stored in `C[k]` subtree: `C[k]` ← `build_IST_subarray(keys[], left +` k^2`, right)`.

We can build all children in parallel, since there is no data dependencies between them.

Fig. 12: Building children of a new node



To finish the construction of a node we need to calculate `node.ID` array described in Section 3.2. We can build it in the following way:

- Create an array `bounds` of size `m + 1` such that `bound[i]` $=$ `keys[left]` $+ i \cdot$ (`keys[right - 1]` $-$ `keys[left]`)$/L$ where $L = \Theta(n^{\varepsilon})$ with some $\varepsilon \in [\frac{1}{2}; 1)$;
- Use `Rank` primitive to find the rank of each `bounds[i]` in the `Rep` array.

8 Theoretical results

In this section, we present the theoretical bounds for our data structure. These bounds are quite trivial, so we just give intuition.

Theorem 1. *The flatten operation of an IST with n elements has $O(n)$ work and $O(\log^3 n)$ span. The building procedure of an ideal IST from an array of size n has $O(n)$ work and $O(\log n \cdot \log\log n)$ span. Thus, the rebuilding of IST with n elements costs $O(n)$ work and $O(\log^3 n)$ span.*

Proof (Sketch). While the work bounds are trivial, we are more interested in span bounds. From [16] we know that in the worst case, the height of IST with n keys does not exceed $O(\log^2 n)$. Thus, the flatten operation just goes recursively into $O(\log^2 n)$ levels and spends $O(\log n)$ span at each level. This gives $O(\log^3 n)$ span in total. The construction of an ideal IST has $O(\log\log n)$ recursive levels while each level can be executed in $O(\log n)$ time, i.e., copy the elements into `Rep` array. This gives us the result.

This brings us closer to our main complexity theorem.

Theorem 2. *The work of a batched operation on our parallel-batched IST has the same complexity as if we apply all m operations from this batch sequentially to the original IST of size n (from [16], the expected execution time is $O(m \log\log n)$). The total span of a batched operation is $O(\log^4 n)$.*

Proof (Sketch). The work bound is trivial — the only difference with the original IST is that we can rebuild the subtree in advance before applying some of the operations. Now, we get to the span bounds. From [16], we know that the height of IST with n keys does not exceed $O(\log^2 n)$. On each level, we spend: 1) at most $O(\log^2 n)$ span for merge and rank operations; or 2) we rebuild a subtree at that level and stop. The first part gives us $O(\log^4 n)$ span, while rebuilding takes just $O(\log^3 n)$ span. This leads us to the result of the total $O(\log^4 n)$ span.

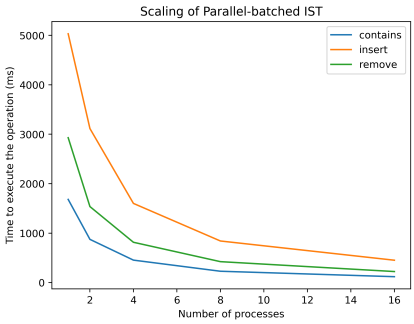
9 Experiments

We have implemented the Parallel Batched IST in C++ using OpenCilk [7] as a framework for fork-join parallelism.

We tested our parallel-batched IST on three workloads. We initialize the tree with elements from the range $[-10^8; 10^8]$ with probability $1/2$. Thus, the expected size of the tree is 10^8. Then we call: 1) search for a batch of 10^7 keys, taken uniformly at random from the range; 2) insert a batch of random 10^7 keys, taken uniformly at random from the range; 3) remove a batch of random 10^7 keys, taken uniformly at random from the range.

The experimental results are shown in Fig. 13. The OX axis corresponds to the number of worker processes and the OY axis corresponds to the time required to execute the operation in milliseconds. Each point of the plot is obtained as an average of 10 runs. We run our code on an Intel Xeon Gold 6230 machine with 16 cores.

Fig. 13: Benchmark results for Parallel-batched Interpolation Search Tree



As shown in the results, we achieve good scalability. Indeed: 1) 14x scaling on `ContainsBatched` operation for 16 processes; 2) 11x scaling on `InsertBatched` operation for 16 processes; 3) 13x scaling on `RemoveBatched` operation for 16 processes.

We also compared our implementation in a sequential mode with `std::set`. `std::set` took 9257 ms to check the existence of 10^7 keys in a tree with 10^8 elements while our IST implementation took only 3561 ms. We achieve such speedup by using interpolation search as described in Section 3.2.

10 Conclusion

In this work, we presented the first parallel-batched version of the interpolation search tree that has an optimal work in comparison to the sequential implementation and has a polylogarithmic span. We implemented it and got very promising results. We believe that this work will encourage others to look into parallel-batched data structures based on something more complex than binary search trees.

References

1. Acar, U.A., Blelloch, G.E.: Algorithms: Parallel and sequential. https://www.umut-acar.org/algorithms-book **6** (2019)
2. Akhremtsev, Y., Sanders, P.: Fast parallel operations on search trees. In: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC). pp. 291–300. IEEE (2016)
3. Aksenov, V., Kokorin, I., Martsenyuk, A.: Parallel-batched interpolation search tree. arXiv preprint (2023), https://arxiv.org/abs/2306.13785
4. Bentley, J.L., Yao, A.C.C.: An almost optimal algorithm for unbounded searching. Information processing letters **5**(SLAC-PUB-1679) (1976)
5. Blelloch, G.E., Ferizovic, D., Sun, Y.: Just join for parallel ordered sets. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 253–264 (2016)

6. Blelloch, G.E., Reid-Miller, M.: Fast set operations using treaps. In: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures. pp. 16–26 (1998)
7. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of parallel and distributed computing **37**(1), 55–69 (1996)
8. Brown, T., Prokopec, A., Alistarh, D.: Non-blocking interpolation search trees with doubly-logarithmic running time. In: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 276–291 (2020)
9. Comer, D.: Ubiquitous b-tree. ACM Computing Surveys (CSUR) **11**(2), 121–137 (1979)
10. Graefe, G., et al.: Modern b-tree techniques. Foundations and Trends® in Databases **3**(4), 203–402 (2011)
11. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: 19th Annual Symposium on Foundations of Computer Science (sfcs 1978). pp. 8–21. IEEE (1978)
12. JáJá, J.: An introduction to parallel algorithms. Reading, MA: Addison-Wesley **10**, 133889 (1992)
13. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The case for learned index structures. In: Proceedings of the 2018 international conference on management of data. pp. 489–504 (2018)
14. Lea, D.: A java fork/join framework. In: Proceedings of the ACM 2000 conference on Java Grande. pp. 36–43 (2000)
15. Medidi, M., Deo, N.: Parallel dictionaries using avl trees. Journal of Parallel and Distributed Computing **49**(1), 146–155 (1998)
16. Mehlhorn, K., Tsakalidis, A.: Dynamic interpolation search. Journal of the ACM (JACM) **40**(3), 621–634 (1993)
17. Park, H., Park, K.: Parallel algorithms for red–black trees. Theoretical Computer Science **262**(1-2), 415–435 (2001)
18. Paul, W., Vishkin, U., Wagener, H.: Parallel dictionaries on 2–3 trees. In: International Colloquium on Automata, Languages, and Programming. pp. 597–609. Springer (1983)
19. Peterson, W.W.: Addressing for random-access storage. IBM journal of Research and Development **1**(2), 130–146 (1957)
20. Prokopec, A., Brown, T., Alistarh, D.: Analysis and evaluation of non-blocking interpolation search trees. arXiv preprint arXiv:2001.00413 (2020)
21. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM **33**(6), 668–676 (1990)
22. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the ACM (JACM) **32**(3), 652–686 (1985)
23. Sun, Y., Ferizovic, D., Belloch, G.E.: Pam: parallel augmented maps. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 290–304 (2018)
24. Willard, D.E.: Searching unindexed and nonuniformly generated files in \log\logn time. SIAM Journal on Computing **14**(4), 1013–1029 (1985)