# Parallel Joinable B-Trees in the Fork-Join I/O Model

**Michael T. Goodrich**
University of California, Irvine, CA, USA

**Yan Gu**
University of California, Riverside, CA, USA

**Ryuto Kitagawa**
University of California, Irvine, CA, USA

**Yihan Sun**
University of California, Riverside, CA, USA

## Abstract

Balanced search trees are widely used in computer science to efficiently maintain dynamic ordered data. To support efficient set operations (e.g., union, intersection, difference) using trees, the *join-based* framework is widely studied. This framework has received particular attention in the parallel setting, and has been shown to be effective in enabling simple and theoretically efficient set operations on trees. Despite the widespread adoption of parallel join-based trees, a major drawback of previous work on such data structures is the inefficiency of their input/output (I/O) access patterns. Some recent work (e.g., C-trees and PaC-trees) focused on more I/O-friendly implementations of these algorithms. Surprisingly, however, there have been no results on bounding the I/O-costs for these algorithms. It remains open whether these algorithms can provide tight, provable guarantees in I/O-costs on trees.

This paper studies efficient parallel algorithms for set operations based on search tree algorithms using a join-based framework, with a special focus on achieving I/O efficiency in these algorithms. To better capture the I/O-efficiency in these algorithms in parallel, we introduce a new computational model, the Fork-Join I/O Model, to measure the I/O costs in fork-join parallelism. This model measures the total block transfers (I/O work) and their critical path (I/O span). Under this model, we propose our new solution based on B-trees. Our parallel algorithm computes the union, intersection, and difference of two B-trees with $O(m \log_B(n/m))$ I/O work and $O(\log_B m \cdot \log_2 \log_B n + \log_B n)$ I/O span, where $n$ and $m \leq n$ are the sizes of the two trees, and $B$ is the block size.

**2012 ACM Subject Classification** Computing methodologies → Massively parallel algorithms

**Keywords and phrases** Parallel algorithm, I/O efficiency, search trees, B-trees

**Digital Object Identifier** 10.4230/LIPIcs.ISAAC.2025.37

**Related Version** *Full Version*: `https://arxiv.org/abs/2510.20053`

## 1 Introduction

Balanced search trees are among the most fundamental data structures in computer science. The search tree structure effectively maintains the ordering for a set of elements with dynamic updates, and a balance guarantee, usually meaning to bound the tree height to be logarithmic, enables efficient cost bounds for both updates and queries. Balanced search trees have been used to support basic data types such as ordered sets and maps in various programming languages, either as built-in data types or in standard libraries.

In addition to individual updates such as insertions and deletions, set operations (e.g., set-union, corresponding to merging two trees) are often needed in the interface of ordered sets and maps. To support efficient set operations on trees, many existing papers study the *join-based framework*. The core of the framework is a function *join*. $\mathsf{Join}(T_L, k, T_R)$ takes


two search trees $T_L$ and $T_R$, and one key $k$ in the middle, such that $\forall k_1 \in T_L$ and $k_2 \in T_R$, $k_1 < k < k_2$, and returns a valid balanced tree containing all elements in $T_L \cup \{k\} \cup T_R$. Existing work has shown that abstracting the Join primitive allows for elegant and efficient designs for set operations on various types of balanced trees, both sequentially [1, 2] and in parallel [7, 9, 10, 13, 14, 24].

The join-based framework was first studied in the sequential setting, on red-black trees [25] and weight-balanced trees [1]. The idea then has received particular attention in the parallel setting, since it conceptually allows for applying a batch of updates to the tree. Blelloch and Reid-Miller [10] first used the join-based framework in parallel on treaps, and proved that they are theoretically efficient. This idea was extended to various balancing schemes in 2014 on a data structure called P-tree [7, 23] and later implemented in a parallel library [24]. The theoretical results and implementations of these trees have been used both to support strong theoretical bounds in other parallel algorithms [12, 15, 17, 21], as well as in various applications such as databases [23], graph processing [14, 16], and more [22, 26].

Despite the widespread adoption of parallel join-based trees, a major drawback of previous work on such data structures is the inefficiency of their input/output (I/O) access patterns; e.g., see [4]. For example, in a tree node that only contains a few elements in the set, the number of memory accesses is asymptotically the same as the operations applied, rather than being a function of a block size or cache-line size. In fact, many follow-up works (e.g., C-tree [14] and CPAM [13]), aim to improve the memory-access efficiency (i.e., the I/O efficiency) with blocked tree nodes or leaves. While these new data structures have provable guarantees in work and span, and have shown good performance in experiments due to better memory access patterns, we are unaware of any theoretical improvements in their I/O efficiency. It is therefore worth asking, whether we can adapt the simple join-based framework to also achieve I/O-efficiency with provable guarantees.

This paper studies efficient parallel algorithms for set operations based on search tree algorithms using a join-based framework, with a special focus on achieving I/O efficiency in these algorithms. Our new solution is based on B-trees, and includes novel ideas to overcome challenges in both algorithm design and analysis for this problem. The challenge to achieving I/O-efficiency in parallel is two-fold. The first reason, which lies in the algorithm design aspect, is to apply the idea to an I/O-friendly data structure, such as B-trees, when binary structures are the norm. As discussed above, there exist previous studies that attempt to make join-based algorithms I/O-friendly by grouping multiple keys as a block in the same tree node or in the leaves [13, 14]. However, they tend to still keep the binary structure of the tree, which introduces needless complications. This design preserves compatibility with the existing join-based algorithmic framework, which can be used out of the box with minimal adaptation. However, since the tree height is $O(\log(n/B))$ for block size $B$ and tree size $n$, this design does not lead to an ideal I/O bound for these set algorithms. To achieve non-trivial bounds, allowing multi-way trees (e.g., B-trees) is critical. However, the multi-way structure of these structures inevitably introduces complications into algorithm design. In particular, a multi-way join is required, which basically concatenates a set of trees with keys in between. Accordingly, a multi-split algorithm is needed to divide a tree using multiple splitter keys. Both of them need careful algorithm re-design over the existing ideas.

The second, and perhaps most interesting, reason related to the analysis aspect is the lack of an effective model to capture the I/O efficiency in these algorithms in parallel. While both models for parallel computing (e.g., PRAM) and I/O models have a long history, analyzing the I/O cost in parallel has some inherent difficulties such as shared/separate cache, synchronization schemes, exclusive/concurrent writes, etc. In 2010, Arge, Goodrich, and Sitchinava introduce the parallel external-memory (PEM) model [5], which focuses on

the I/O bottleneck in PRAM algorithms. Built on top of the PRAM model, PEM assumes synchronized threads. On the other hand, the join-based tree algorithms, as well as their implementations in software libraries, are based on the classical fork-join model, which is highly asynchronous.

## Our Results

To accurately analyze I/O cost in asynchronous parallel algorithms, we introduce the *Fork-Join I/O Model* in this paper, which more formally defines the I/O cost in fork-join parallelism. Analogous to work and span for the standard fork-join model, the Fork-Join I/O Model measures the I/O cost of parallel algorithms by both the total number of block transfers (referred to as **I/O work**), and the maximum number of block transfers one depends on the previous (referred to as **I/O span**). Based on this model, we provide and analyze new parallel and I/O-efficient algorithms on set/map operations using B-trees.

Intuitively, the Fork-Join I/O Model provides a software-based asynchronous alternative to the hardware-based synchronous parallel external-memory (PEM) model. We also present parallel I/O-efficient algorithms for B-trees in this model, such as union, intersection, and difference operations. To do this, we first design a $B$-way join algorithm, which may concatenate at most $B$ trees. Using this primitive, we design a general join algorithm that takes any number of $k$ trees and $k-1$ keys in between, and an inverse function, that splits a tree by $k-1$ keys. Based on the two functions, we design an I/O-efficient parallel set algorithms. We also show in the Appendix how to modify the union algorithm to also perform the intersection and difference operations, while achieving the same I/O bounds asymptotically.

We highlight that the design of the algorithm is highly non-trivial. Directly applying the simple $B$-way join (and a corresponding $B$-way split) algorithm to a Union algorithm leads to $O(\log_B n \log_B m)$ I/O span. A specific interesting technical contribution in this paper is to achieve a stronger I/O span bound of $O(\log_B m \cdot \log_2 \log_B n + \log_B n)$, for which we introduce the more sophisticated algorithms of arbitrary-way join and split. We summarize the main results below.

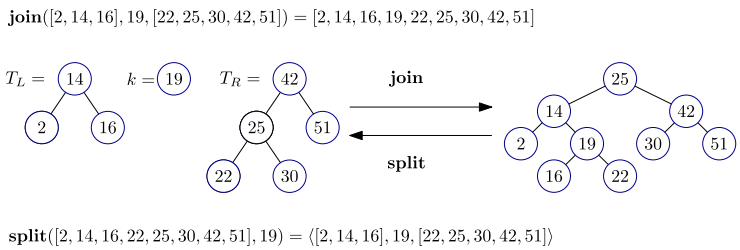
▶ **Theorem 1.1** (Parallel Set Operations on B-trees). *Given two B-trees with sizes $m$ and $n \geq m$, there exists a parallel algorithm that returns a new B-tree containing the union of the two input trees in and $O\big(m \log_B\big(\frac{n}{m}\big)\big)$ I/O work, $O(\log_B m \cdot \log_2 \log_B n + \log_B n)$ I/O span, where $B$ is the block size.*

Due to space limit, we present the Union algorithm and analyze it in the main paper, and show how to achieve the same bounds for Intersection and Difference in Appendix I.

## 2 Preliminaries

**B-trees.** A B-tree is a multi-way self-balancing tree data structure that maintains an ordered set of keys. With clear context, for a B-tree $T$, we also use $T$ to denote the set of keys in $T$. We use the standard definitions of parent, child, sibling, ancestor, and descendants in B-trees. More formally, a B-tree is either an empty node (external node), or a node with a set of $b$ keys $k_1, k_2, \ldots k_b$ with $b+1$ subtrees $T_1, T_2, \ldots T_{b+1}$ with the following invariants:

- $\lceil B/2 \rceil \leq b \leq B$ for some parameter $B$, except for the root, where $2 \leq b \leq B$.
- Each subtree $T_i$ is a B-tree.
- $\forall x \in T_{i-1}$ and $y \in T_i$, we have $x < k_i < y$.
- All external nodes have the same depth, where the depth of a node is the number of ancestors (inclusive) of it.




```
union(T1, T2) {
    T_{L,1} ← left subtree of T1
    T_{R,1} ← right subtree of T1
    r ← the root of T1
    ⟨T_{L,2}, k, T_{R,2}⟩ ← split(T2, r)
    In parallel:
        L ← union(T_{L,1}, T_{L,2})
        R ← union(T_{R,1}, T_{R,2})
    return join(L, r, R)
}
```

**Figure 1** Illustration for Join and Split algorithms on binary trees, and an example of the set union algorithm. To extend the idea to B-trees, multi-way Join and Split algorithms are needed.

The height of a B-tree $T$, denoted as $h(T)$, is 0 if it is an external node, and $h(T') + 1$ if $T$ has a child $T'$. If $T$ has multiple children, all of them should have the same height.

When we discuss an algorithm on two B-trees, we always use $n$ to denote the one with the larger size and $m \leq n$ the smaller size.

The B-tree includes an operation for combining nodes when the number of keys in the node is below $\lceil B/2 \rceil$, and for splitting a node into two when there are more than $B$ keys in the node. We will refer to these operations as **fuse** and **divide** operations, to avoid conflicting terminology.

**Sequence Notation.** We use $a_{i..j}$ $(i \leq j)$ to denote a sequence of elements $\langle a_i, a_{i+1}, \ldots, a_j \rangle$.

**Background of Join-based Algorithms.** For completeness, we introduce some background of join-based algorithms on binary trees in Appendix A. We present Figure 1 as an illustration for the high-level idea, as well as an example of implementing a Union based on join. We introduce more details about related work in Appendix B.

One technical challenge to extend this idea to B-trees is to support multi-way Join and Split functions, which may join multiple trees with keys in the middle, or split a tree using multiple splitters. Since the input tree heights may vary, these primitives must rebalance the returned tree properly. We introduce our algorithms for these primitives in Section 4.2 and 4.3, and finally present our Union algorithm in Algorithm 4.

## 3 The Fork-Join I/O Model

In this paper, we propose the Fork-Join I/O Model, which is an extension of the classic work-span model to analyze algorithms. The goal of this paper to use this model is to more precisely capture the I/O costs in algorithms based on fork-join parallelism, which is crucial for analyzing and implementing existing join-based algorithms on trees.

Recall that in the work-span model in the classic multithreaded model [6,9,11], we assume a set of threads share their memory. Each thread acts like a sequential RAM plus a fork instruction that forks two threads running in parallel. When both threads are finished with thir task, the original thread continues. A parallel-for is simulated by forks in a logarithmic number of steps. A computation can be viewed as a DAG. The *work* $W$ of a parallel algorithm is the total number of operations, and the *span (depth)* $S$ is the longest path in the DAG. In practice, we can execute the computation with work $W$ and span $S$ using a randomized work-stealing scheduler [11,18] in time $W/P + O(S)$ with $P$ processors with high probability.

The extension we make to this model to adapt for practicality is that, instead of considering executing one word on RAM, we consider unit cost for reading or writing a block of size $B$, where $B$ is a given parameter. Here $B$ can be the cacheline size when referring to in-memory

algorithms, or the block size when considering external-memory algorithms. In either case, this is the basic unit for block transfers, which is indeed the motivation of the design of B-trees that improves the performance as compared to a simpler binary search tree. Here we assume each processor can hold $O(\log n)$ blocks (i.e., the stack memory) to execute the program. Accessing the data in these blocks is free (it does not asymptotically change the complexity, but only makes the analysis easier). We define this modified work as the **I/O work**. Regarding the span, we also adapt similarly (i.e., count the access of an entire block with unit cost), and refer to this as the **I/O span**. Here we assume a fork or join can create or synchronize $B$ threads for simplicity. All I/O span bounds remain the same when translating the bound to PRAM, or are affected by a factor of $\log B$ (mostly a small constant) to the binary fork-join model [9].

An important subroutine we will be using in this model is the Gather primitive. This operation allows us to gather the $O(B)$ elements which are stored in various blocks in external memory into a single block in $O(1)$ I/O span and $O(B)$ I/O work. During the synchronization of each thread, the thread may return a $O(1)$ amount of data, thus the total amount of data returned by all threads is $O(B)$.

# 4 Parallel Joinable B-Tree

In this section, we present primitive algorithms for the joinable B-trees. We first introduce the B-Way-Join operation, which takes a sequence of $(b+1)$ B-trees and $b$ keys in between, and combines them into a valid, balanced B-tree. We then show the B-Way-Join algorithm with detailed cost analysis in Section 4.1. Then, the B-Way-Join algorithm is used as a subroutine to develop the more general Multi-Split and Multi-Join algorithms in Section 4.2 and 4.3 in order to achieve better I/O bounds for the Union algorithm, described in Section 5.

These join and split algorithms are similar to the approach found in the other parallel search trees [7, 10], but adapted for the Fork-Join I/O Model.

These algorithms will be used as the building blocks for the set operations in Section 5. The key idea is to use the Multi-Split operation to split the two B-trees into $\sqrt{n+m}$ subtrees each, where the union of a pair of the subtrees is of size $\sqrt{n+m}$. We then make recursive calls to the union of these pairs, leading to $\sqrt{n+m}$ total subtrees, which we then use the Multi-Join algorithm to combine into a single B-tree.

The Multi-Split operation uses B-Way-Join as a primitive, by first searching for one of the keys they will be splitting on, which creates a series of subtrees which must then be joined together, rebalancing the resulting tree. The Multi-Join operation on the other hand is a more complex operation and differs significantly from the B-Way-Join operation.

## 4.1 B-way Join

### 4.1.1 Algorithm Description

The join operation merges $b+1$ B-trees $T_1, T_2, \ldots, T_{b+1}$, and $b$ separator keys $k_1, k_2, \ldots, k_b$, into a single valid B-tree, where $b \leq B$. Each input tree $T_i$ must be a valid B-tree and $T_i$ contains keys only in the range $(k_{i-1}, k_i)$, where $k_0 = -\infty$ and $k_{b+1} = \infty$.

The primary challenge lies in ensuring the join operation is both computationally efficient and maintains the balance of the tree, even when the height of the input trees may differ significantly. We describe our algorithm below in three steps. An illustration of the three steps on an example input is in Section 4.1. In Appendix C, we prove the correctness and efficiency of this algorithm.

**Algorithm 1** B-Way-Join$(T_{1..(b+1)}, k_{1..b})$.

**Input:** A sequence of $b+1$ B-trees $T_{1..(b+1)}$ and $b$ keys $k_{1..b}$. For any $x \in T_i$ and $y \in T_{i+1}$, we have $x < k_i < y$.
**Output:** A B-tree $T$ with all keys in $\{k_1, k_2, \ldots, k_b\} \cup \bigcup T_i$

```
1  h* ← max_i h(T_i)                                   // h* is the largest tree height
2  A ← the subsequence of ⟨1, ..., b⟩, where i ∈ A iff. h(T_i) = h*   // indexes of tall trees
3  if the first element in A is not 1 then add 0 at the beginning of A
4  ParallelForEach i ∈ A do
5      i' ← the successor of i in A                    // T_{i'} is the next tree with height h*
6      if i = 0 then // Handling the special case when there exist keys to the left of the first tall tree
7          R ← the leftmost subtree of T_{i'}
8          S ← ⟨T_1, ... T_{i'-1}, R⟩                  // The sequence of trees for the leftmost recursive call
9          k' ← ⟨k_1, k_1, ..., k_{i'-1}⟩              // The sequence of keys for the leftmost recursive call
10         X ← B-Way-Join(S, k')                       // All keys to the left of the first tall tree
11         Replace the leftmost subtree of T_{i'} with X   // T_{i'} may be unbalanced for now
12     else
13         L ← the rightmost subtree of T_i
14         S ← ⟨L, T_{i+1}, T_{i+2}, ..., T_{i'-1}⟩    // The sequence of trees for the recursive call
15         k' ← ⟨k_{i+1}, k_{i+2}, ..., k_{i'-1}⟩      // The sequence of keys for the recursive call
16         X ← B-Way-Join(S, k')   // X will contain all keys between the i-th and the i'-th tree
17         Replace the rightmost subtree of T_i with X     // T_i may be unbalanced for now
18 if the first element in A is 0 then remove 0 from A
19 T ← Concatenate all (new) trees T_i, with keys k_{i-1}, for all i ∈ A
20 SimpleRebalance(T)                                  // Introduced in Lemma 5
```

**Step 1: Grouping.** The algorithm starts by identifying the height of the tallest tree in the input set, which we denote as $h^*$. We say a tree $T_i$ is a **tall tree** if it has height $h^*$, and a **short tree** otherwise. Note that there may be multiple tall trees in the sequence $T$. Let $A \subseteq \langle 0, 1, \ldots, b\rangle$ be the subsequence of indices of trees with height $h^*$. We then partition the input trees into $|A|+1$ groups. At a high level, each group contains all keys and trees between two tall trees, and the leftmost subtree of the previous tall tree. If $T_1$ is not a tall tree, a special group, which we call a **leading group**, to the left of $T_1$ will also be formed.

More formally, assume $i \in A$, and $i' = \mathsf{succ}(i)$ is the successor of $i$ in $A$.[1] Then the corresponding group includes $i' - i$ trees, which are the rightmost subtree in $T_i$, and all trees from $T_{i+1}$ to $T_{i'-1}$, and $i' - i - 1$ keys, which are from $k_i$ to $k_{i'-2}$. Note that the trees are separated by the corresponding keys in the same group. If $T_1$ is not a tall tree, all elements to the left of the first tall tree $T_{i'}$ form the leading group, which includes all trees up to $T_{i'-1}$, the leftmost subtree of $T_{i'}$, and all keys up to $k_{i'-1}$.

**Step 2: Recursing and Subtree Replacing.** After the first step, each group contains a sequence of trees, and the corresponding keys that separate them. The second step simply calls B-Way-Join recursively to reorganize each group into a valid B-tree. After that, for the group corresponding to trees between $T_i$ and $T_{\mathsf{succ}(i)}$, we use the resulting tree to replace the rightmost subtree in $T_i$. If there exists a leading group, it replaces the leftmost subtree in the first tall tree.

In Theorem 7 we will show that B-Way-Join on a sequence of trees with maximum height $h$ will lead to a tree with height at most $h+1$. In this case, all trees involved in the recursive call is either a short tree, which has height at most $h^* - 1$, or a subtree of a tall tree, which

[1] We define $\mathsf{succ}(\max(A))$, where $\max(A)$ the last element in $A$, to be $b+1$.

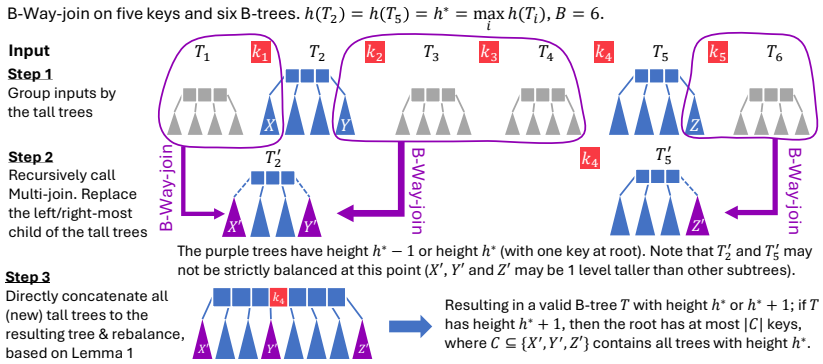



**Figure 2** An illustration of the B-Way-Join algorithm (described in Section 4.1) with 6 trees and 5 keys. The blue trees are the *tall trees* with height $h^*$. All other trees has height less than $h^*$.

has height $h^* - 1$. Therefore, the resulting trees from recursive calls have height at most $h^*$. After replacing a subtree in a tall tree, it can be at most one level taller than the other subtrees.

**Step 3: Concatenating and Rebalancing.** After the previous step, all short trees have been combined into a tall tree. Note that for each tall tree, except for the first one, the key immediately before it is not included in any groups. Therefore, our final step is to concatenate all these remaining components together, which includes all tall trees $T_i$ for all $i \in A$, and $k_{i-1}$ for all $i \in A$. This is performed by creating a B-tree node with keys at the root at $T_{A_1}$, then $k_{A_2-1}$, then all keys at the root of $T_{A_2}$, then $k_{A_3-1}$, so on so forth. The subtrees, from left to right, are all subtrees in $T_{A_1}$, then all subtrees in $T_{A_2}$, so on so forth. Let the resulting tree be $T$.

The tree $T$ is a valid B-tree except for two aspects: 1) some of its subtrees may be taller than other subtrees, but can be taller by at most 1, and 2) the number of keys at the root may be more than $B$. For such minor imbalance, we will use the SimpleRebalance algorithm, introduced in Lemma 5, to rebalance the tree. At a high level, the rebalancing algorithm simply promotes all keys at the subtree root of the taller subtrees to the root of the entire tree, and if an overflow occurs, we can promote every $B$ keys at the root to a higher level, increasing the tree height by 1.

See Figure 2 for an illustration of the algorithm.

▶ **Theorem 4.1** (B-Way Join Analysis). *Let $T_1, T_2, \ldots, T_{b+1}$ be a set of B-trees, with the largest tree height $h_{\max}$ and the shortest tree height $h_{\min}$, and $k_1, k_2, \ldots, k_b$ be a set of separator keys, where $b \leq B$. The B-Way-Join operation can be performed in $O(h_{\max} - h_{\min})$ I/O span and $O(b \cdot (h_{\max} - h_{\min}))$ I/O work.*

We analyze the cost of the B-Way-Join operation in Appendix C.

## 4.2 Multi-Split

The Multi-Split algorithm takes a B-tree $T$ and a sorted list of $d$ keys $k_{1..d}$, and returns $d+1$ subtrees, such that the $i$-th subtree contains all keys in the range $(k_{i-1}, k_i)$, where $k_0 = -\infty$ and $k_{d+1} = \infty$. This is achieved by spawning $d+1$ threads, where each thread $i$ invokes a helper function Thread-Split, which outputs a single subtree containing all keys in the range $(k_{i-1}, k_i)$. See Algorithm 2.

**Algorithm 2** Thread-Split$(T, k_1, k_2)$.

**Input:** A B-trees $T$ and two keys $k_1, k_2$ such that $k_1 < k_2$
**Output:** A B-tree $T$ containing all keys in the range $(k_1, k_2)$

```
1  if T is empty or contains only keys between k1 and k2 then return T
2  if T is a leaf then return the B-tree containing all keys in the range (k1, k2)
3  r_{1..b} ← the keys at the root of T
4  i ← the smallest index such that k1 < r_i, 0 if k1 < r1
5  j ← the largest index such that r_j < k2, b + 1 if r_b < k2
6  C_{i..j} ← all children of T from i to j, inclusive
7  if i = j then return Thread-Split(C_i, k1, k2)
8  C'_i ← Thread-Split(C_i, k1, ∞)
9  C'_j ← Thread-Split(C_j, −∞, k2)
10 r'_{i..j} ← copy of keys r_{i..j}        // Prevents adjacent threads from dividing the same node
11 return B-Way-Join(⟨C'_i, C_{i+1..j−1}, C'_j⟩, r'_{i..j})
```

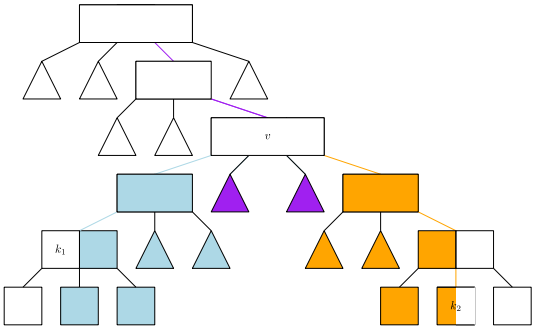



**Figure 3** Above is a B-Tree, where each node rectangle is a node in the B-Tree, whereas the triangles are subtrees shortened for space. There are two search paths to $k_1$ and $k_2$ from the root, sharing the purple path at the start, then separating into the blue and orange paths respectively at node $v$. All nodes colored blue are the result of following the blue path and copying the appropriate keys, and similarly for the orange nodes. The node containing $k_2$ excludes all keys greater than or equal to $k_2$, thus being half orange. The result of Thread-Split will be the union of the blue, orange, and purple nodes. All white sections of the tree will be the search paths of other Thread-Split operations.

The high-level idea of the algorithm is to search for the keys $k_i$ and $k_{i+1}$ in $T$, copying the nodes and keys along the search path that fit within the range $(k_i, k_{i+1})$. Any node that lies on the shared prefix of both search paths, i.e. before the paths to $k_i$ and $k_{i+1}$ diverge, are not part of the output tree and can be skipped. Once the paths diverge, recursively split down the search path of $k_1$ and $k_2$. If our entire node falls within the range $(k_1, k_2)$, then a copy of the pointer to the node is made. Otherwise, only the keys which fall within the range $(k_1, k_2)$ and the pointers to the children that fall within the range are copied.

We copy the keys of the node instead of dividing it, which is the natural operation to perform, because dividing requires synchronization between adjacent threads. See Figure 3.

On the way back up the tree, the algorithm performs a B-Way-Join operation to combine the subtrees and keys which contain the keys in the range $(k_1, k_2)$. This serves the purpose of both combining and rebalancing the subtrees. We prove the following theorem in Appendix D.

▶ **Theorem 1** (Multiway Split Cost). *We can split a B-tree $T$ by $k_1, \ldots, k_d$ keys in parallel with $O(\log_B d + \log_B n)$ I/O span and $O(d \log_B n)$ I/O work, where $n$ is $\sum_{i=1}^{d} |T_i|$, in the Fork-Join I/O Model.*

**Algorithm 3** B-Way-Join-Fast$(T_{1..b+1}, k_{1..b})$.

**Input:** A sequence of $b+1$ B-trees $T_{1..b+1}$ and $b$ keys $k_{1..b}$. For any $x \in T_i$ and $y \in T_{i+1}$, we have $x < k_i < y$. For all $T_i$, there is at most one node with $B$ or more keys along their left and right spines.

**Output:** A B-tree $T$ with all keys in $\{k_1, k_2, \ldots, k_b\} \cup \bigcup T_i$

$h^* \leftarrow \max_i h(T_i)$ *// $h^*$ is the largest tree height*

$A \leftarrow$ the subsequence of $\langle 1, \ldots, b \rangle$, where $i \in A$ iff. $h(T_i) = h^*$ *// indexes of tall trees*

**if** the first key in $A$ is not 1 **then**

  $i' \leftarrow$ the first key in $A$

  $T_{i'} \leftarrow$ Join-With-Tall$(T_{1..i'}, k_{1..i'-1})$

**ParallelForEach** $i \in A$ **do**

  $i' \leftarrow$ the successor of $i$ in $A$ *// $T_{i'}$ is the next tree with height $h^*$*

  $T_{i'} \leftarrow$ Join-With-Tall$(T_{i..i'}, k_{i..i'-1})$

**if** the first key in $A$ is 0 **then** remove 0 from $A$;

$T \leftarrow$ Concatenate all (new) trees $T_i$, with keys $k_{i-1}$, for all $i \in A$

SimpleRebalance$(T)$ *// See Lemma 5*

## 4.3 Multi-Join

Suppose we have $d+1$ B-trees $T_{1,\ldots,d+1}$, and a sorted list of $d$ keys $k_{1,\ldots,d}$, such that for any $x \in T_i$ and $y \in T_{i+1}$, we have $x < k_i < y$. The Multi-Join algorithm takes as input $T_{1,\ldots,d+1}$ and $k_{1,\ldots,d}$, and returns a single tree $T$, such that $T$ contains $\bigcup_{i=1}^{d} T_i$ and $\{k_1, k_2, \ldots, k_d\}$.

We first describe here an overview of the Multi-Join algorithm. Section 4.1 shows how to join $B$ trees in $O(\log_B n)$ I/O span and $O(B \log_B n)$ I/O work. Therefore, a simple solution is to join $B$ trees together in parallel across $\log_B d$ rounds of joining. However, this results in an algorithm with $O(\log_B d \log_B n)$ I/O span, which is not optimal for larger values of $d$.

We improve the upper bound on the I/O span by reducing the cost of joining $B$ trees down to an $O(\log_B \log_B n)$ I/O span and $O(\log_B n)$ I/O work, which we call the B-Way-Join-Fast operation. Let us define nodes containing more than $B$ keys as **overloaded**, and nodes with exactly $B$ keys as **full**. In order to achieve these bounds, we use a similar strategy to Akhremtsev and Sanders [3], where the trees are preprocessed such that each tree $T_i$ contains a list of pointers to all nodes along their left and right spines. This allows us to efficiently find the level at which each tree must traverse in order to perform the join operation. For each B-Way-Join-Fast operation, the output tree must also maintain this list of pointers. Then we show how to divide all nodes that will be overloaded during B-Way-Join-Fast in parallel.

### 4.3.1 Faster B-way Join

Let $L_i$ and $R_i$ be the list of pointers to the left and right spines of some tree $T_i$ respectively. $L_i$ and $R_i$ are sorted in reverse order by the height of the node, i.e. the leaf node is at index 0 and the root is at index $h(T_i)$. Implementation details are discussed in Section 4.3.2.

**Step 1: Grouping** As in the B-Way-Join algorithm in Section 4.1, we first identify the tallest trees of $T_{1,\ldots,b}$, and group the trees and keys together.

**Step 2: Fusing Within Groups** Then, we use the list of pointers to the left and right spines in order to jump directly to the correct height of the tree for each group.

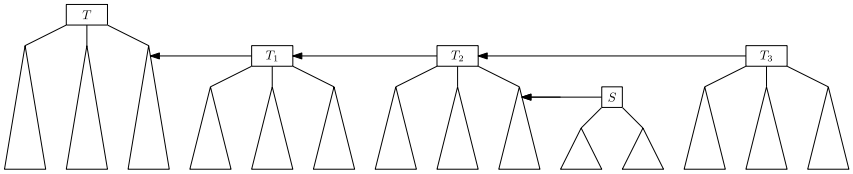



**Figure 4** Trees $T_1, T_2$, and $T_3$ are all trees of the same height in the same group. $T_1$ fuses with a node in $T$, but at the same time, $T_2$ is also fusing with the root node of $T_1$, and $T_3$ is fusing with the root node of $T_2$. Therefore, there is a dependency where $X$ must fuse with $T_2$ first, then $T_2$ must fuse with $T_1$, then $T_1$ with $T$. While $S$ is also fusing with a node in $T_2$, $S$ is fusing with node other than the root of $T_2$, thus there is no conflict.

Each tree $T_j$ is assigned some $T_i$ to be fused with, where $i$ is the largest index within the same group such that $i < j$ and $h(T_i) \geq h(T_j)$. Then a thread is spawned for each $T_j$, and its corresponding key are fused with $L_i[h(T_j)]$ or $R_i[h(T_j)]$. As in Section 4.1, we also account for the special case where $T_1$ is not the tallest tree.

Conflicts occur when multiple trees of the same height are in the same group, since threads may try to fuse with roots of trees that are also fusing with other trees. However, in constant I/O span and work, each thread is able to identify which portion of the tree they will end up fusing with and directly write to that node. Consider the tree $T_i$, and let $X$ be the set of all trees with the same height as $T_i$. $X$ can be constructed using Gather to get the heights of all $T_i$ in a single cache line and constructing each set in internal memory. Then all $B$ threads are spawned, and thread $i$ can check if $|X| > 1$. If so, thread $i$ can also identify the tree in $X$ with the smallest index and the tree $T'$ they would be fusing with. If $T'$ exists, then thread $i$ can fuse $T_i$ with $T'$ instead, while ensuring there are no write conflicts with other threads which are fusing with the same node. For example, in Figure 4, $T_1$ is writing keys to some node in $T$, which we call $v$. $T_2$ will also write to $v$, but starting from the index $|v| + |T_1|$ away from the start of $|v|$.

**Step 3: Dividing Large Nodes** After each thread has finished fusing their trees, they check if the node is overloaded. If so, then the thread will divide the node into $\lceil \frac{b}{B} \rceil$ nodes, where $b$ is the number of keys in the node. However, before pushing $\lceil \frac{b}{B} \rceil - 1$ keys up to the parent node, all threads are first synchronized in order to avoid write conflicts. Then all threads are spawned again to push up their keys to the parent node. We show in Lemma 10, in Appendix E, all overloaded nodes will push at most one key to their parent node.

**Step 4: Dividing Smaller Nodes in Parallel** Any remaining overloaded nodes will only cause consecutive full nodes above it to be divided, since a single key is pushed up per overloaded node. By computing the number of consecutive full nodes above each overloaded node, we can divide these nodes in parallel.

Using the prefix sum algorithm, described in Appendix F of the Appendix, we can accomplish this. Create an array $C_v$ of length $\log_B n$ for each overloaded node $v$, where $C_v[i]$ is set to 1 if the node $i$ levels above is full or overloaded, and 0 otherwise. If there is no node $i$ levels above $v$, then $C_v[i]$ is set to 0. Then take the prefix sum of each $C_v$, and mark the node $i$ levels above $v$ to be divided if $C_v[i] = i$. The number of threads which marks a node to be divided is the number of keys pushed up into it. Finally, a thread is spawned for each node which is marked to be divided, and is divided in parallel.

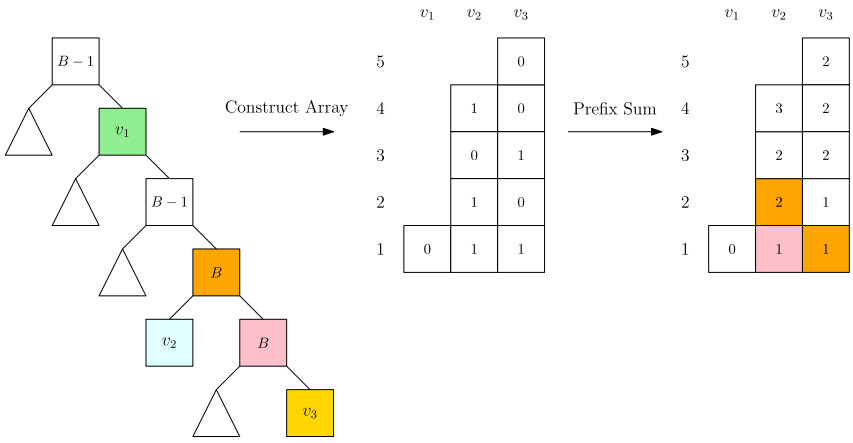



**Figure 5** The B-tree contains 3 nodes $v_1, v_2$, and $v_3$ which are overloaded. We use the list of pointers in order to construct our arrays $C$. A cell within the array is set to 1 if the node is full or overloaded, i.e. it contains $B$ or more keys. Otherwise, it is set to 0. Note that since $v_1$ only contains a single node above it, its array is also of length 1. Then we take the prefix sum of each array, and mark a node to be divided if $C[i] = i$. The three colored cells are the nodes which are to be divided. Two of the cells are orange, referring to the same node, meaning the node takes a key from two of its children.

**Step 5: Concatenating and Rebalancing** Just as in the B-Way-Join algorithm, after the previous steps, all short trees have been combined into a single tall tree, which is then rebalanced.

### 4.3.2 Maintaining the Left and Right Pointers

In order to achieve the I/O span and work bounds we desire, maintaining the list of pointers along the spines of the trees is crucial. We achieve this by implementing the lists as sequential B-trees, where the keys are pointers to the nodes along the spine. By using order statistics information, searching for the node at height $i$ takes $O(\log_B \log_B n)$ I/O span and work.

If a new node is created for the left or right spine after the dividing process, we add the node to the front of $L_i$ or $R_i$ respectively. This also takes $O(\log_B \log_B n)$ I/O span and work. Finally, after the fusing operations complete, we must also update the pointers to the left/right spines of the trees. Recall in Figure 4 that part of the right spine of $T_1$ is being covered by the tree $T_3$, specifically all nodes with a height lower than $T_3$'s root are not on the right spine of the output tree. More generally, suppose $S$ is the set of all trees that were grouped together and has at least one node from their right spine which is part of the right spine of the output tree. Then $T \in S$ will have all nodes along its right spine less than the height of the next smaller tree in $S$ covered. These are the nodes that should be removed from their list of spines. To do this, we simply find the height, $h$, of the next smaller tree in $S$ and perform a sequential split on $T$. Then we use B-Way-Join to join all trees together.

## 5 Union

### 5.1 Algorithm Description

Let $T_1$ and $T_2$ be two B-trees with $n$ and $m$ keys respectively.

The union algorithm is a simple $d$-way divide and conquer algorithm, where $d = \sqrt{n+m}$. If $m < \sqrt{n+m}$, then we have reached the base case, and we call a I/O work inefficient union algorithm. Let $|T_1| \geq |T_2|$, as we can swap the two trees if this is not the case. The I/O work

**Algorithm 4** Union$(T_1, T_2)$.

**Input:** Two B-trees $T_1$ and $T_2$
**Output:** A B-tree $T$ with all keys in $T_1 \cup T_2$

```
1  if |T2| > |T1| then T1, T2 ← T2, T1 ;
2  if |T2| < √(|T1| + |T2|) then return Work-Inefficient-Union(T1, T2) ;
3  d ← √(|T1| + |T2|)
4  ParallelForEach 1 ≤ i ≤ d do
5  |  k_i ← Two-Way-Binary-Search(((|T1|+|T2|)i)/d, T1, T2)
6  ⟨T_{1,1}, T_{1,2}, ..., T_{1,d+1}⟩ ← Multi-Split(T1, k_{1..d})
7  ⟨T_{2,1}, T_{2,2}, ..., T_{2,d+1}⟩ ← Multi-Split(T2, k_{1..d})
8  ParallelForEach 1 ≤ i ≤ d + 1 do
9  |  T_i ← Union(T_{1,i}, T_{2,i})
10 return Multi-Join(T_{1..d+1}, k_{1..d})
```

inefficient union algorithm uses all the keys in $T_2$ and uses them as keys to split on $T_1$ using the Multi-Split operation. Then we use the Multi-Join operation to combine the $d$ subtrees and the $d-1$ keys we split on.

Otherwise, we search $T_1$ and $T_2$ for $d-1$ keys, which we will use to split each tree into $d$ subtrees using the Multi-Split operation. We denote the subtrees of $T_1$ as $T_{1,\{1,\ldots,d\}}$ and $T_2$ as $T_{2,\{1,\ldots,d\}}$. The $i$-th key which we split on is the global $(n+m)\cdot i/d)$-th key. This ensures that $|T_{1,i}| + |T_{2,i}|$ is all approximately $\frac{n+m}{d}$. We then spawn $O(d)$ threads, and recursively call the union algorithm on each pair of subtrees $T_{1,i}$ and $T_{2,i}$. Then we use the Multi-Join operation to combine the $d$ subtrees and the $d-1$ keys we split on.

## 5.2 Cost Analysis

Due to space constraints, we have put the details of the analysis for the base case of the union algorithm to Appendix G.

▶ **Lemma 2** (Union Algorithm Base Case Cost). *The Work-Inefficient-Union algorithm takes $O(\log_B m \cdot \log_2 \log_B n + \log_B n)$ I/O span and $O(m \log_B n)$ I/O work.*

▶ **Lemma 3.** *Algorithm 4 takes $O(\log_B m \cdot \log_2 \log_B n)$ I/O span.*

**Proof.** The I/O span of finding $d-1$, where $d = \sqrt{n+m}$, keys and splitting along them costs $O(\log_B(\sqrt{n+m}))$, and the I/O span of joining all the recursive components costs $O(\log_B(\sqrt{n+m}) \cdot \log_2 \log_B n)$. Recall that $m > \sqrt{n+m}$, since otherwise we would be in the base case of the algorithm. Therefore, the total I/O span of the above operations is $O(\log_B m \cdot \log_2 \log_B n)$.

Then for the next recursive call, the problem size shrinks to $\sqrt{n+m}$. The I/O span of the base case is also $O(\log_B m \cdot \log_2 \log_B n)$. Therefore, the cost of the I/O span is upper bounded by

$$S(n+m) = S(\sqrt{n+m}) + O(\log_B m \cdot \log_2 \log_B (n+m)).$$

Solving for this recurrence relation gives us $O(\log_B m \cdot \log_2 \log_B n)$, when $m < n$. ◀

▶ **Lemma 4.** *The total I/O work of the Union algorithm is $O(m \log_B \frac{n}{m})$.*

**Proof.** The total cost of the base case for the Union algorithm is $O(m \log_B \frac{n}{m})$ from Lemma 2 in the Appendix. Thus, we focus on the cost of the recursive calls.

We see that the I/O work cost is dominated by the cost of Multi-Split, Multi-Join, and searching for the split keys. All of these operations take $O(\sqrt{n+m}\log_B(n+m))$ I/O work.

We perform a similar analysis to the one in Blelloch et al. [9]. Consider the recurrence tree of this algorithm, which is a $\sqrt{n+m}$-way tree, and the cost of the current node is $\sqrt{n+m}\log_B(n+m)$. Since the recursion stops when $m < \sqrt{n+m}$, the recurrence tree is not balanced and some branches will reach the base case earlier than others. However, the cost of a node at the same level also costs the same, and decreases as the tree goes deeper. Let $t = n+m$, then the total I/O work of Union is:

$$t^{1/2}\log_B t + t^{1/2}\cdot t^{1/4}\log_B t^{1/2} + \cdots = \sum_i \frac{\log_B t}{2^i} t^{1-1/2^{i+1}}.$$

We can see the above recurrence is leaf dominated. For all $i$, $t^{1-1/2^{i+1}}$ is the number of split keys which are being searched. Naturally the total number of split keys can never be more than $m$, else we would have already reached the base case, meaning that $t^{1-1/2^{i+1}} \le m$. Let $x = t^{1/2^{i+1}}$, then the total leaf cost is

$$O(t^{1-1/2^{i+1}}\cdot\log_B t^{1/2^{i+1}}) = O\left(\frac{t}{x}\log_B x\right).$$

Clearly the above function decreases as $x$ approaches infinity, and since $t^{1-1/2^{i+1}} \le m$, we know that $x \ge t/m$. Letting $x = t/m$ gives us the maximum value of the function, which is $O(m\log_B(\frac{n}{m}))$, giving us our desired I/O work bounds. ◀

Combining the results above we have the main theorem of this paper.

▶ **Theorem 1.1** (Parallel Set Operations on B-trees). *Given two B-trees with sizes $m$ and $n \ge m$, there exists a parallel algorithm that returns a new B-tree containing the union of the two input trees in and $O\big(m\log_B\big(\frac{n}{m}\big)\big)$ I/O work, $O(\log_B m\cdot\log_2\log_B n + \log_B n)$ I/O span, where $B$ is the block size.*

We show in Appendix I how to achieve the above bounds for the other set operations.

## 6 Conclusion

In this paper, we have introduced the parallel set operations on B-trees that achieve provable I/O efficiency in the context of a novel cost model, the Fork-Join I/O Model, which offers a way to analyze the I/O costs of algorithms in the fork-join setting. This captures the I/O complexity of algorithms executed under asynchronous fork-join parallelism, unlike the Parallel External Memory (PEM) model, which assumes synchronized threads at each step of the algorithm. Our model provides a more realistic abstraction for analyzing parallel algorithms that rely on asynchronous scheduling. We showed how to adapt the join-based paradigm to B-trees by developing new algorithms for Multi-Join, and Multi-Split. These primitives serve as building blocks for implementing more complex operations, such as the union, intersection, and difference operations. This allows us to achieve a near-optimal I/O work of $O\big(m\log_B\big(\frac{n}{m}\big)\big)$, and I/O span of $O(\log_B m\cdot\log_2\log_B n + \log_B n)$. Potential future work would be to reduce the I/O work to the optimal $O\big(\frac{m}{B}\log_B\big(\frac{n}{m}\big)\big)$ I/O complexity, and reduce the I/O span to $O(\log_B(n+m))$.

## References


1 Stephen Adams. Implementing sets effciently in a functional language. Technical Report CSTR 92-10, University of Southampton, 1992.

2 Stephen Adams. Efficient sets – A balancing act. *Journal of functional programming*, 3(04):553–561, 1993. doi:10.1017/S0956796800000885.

3 Yaroslav Akhremtsev and Peter Sanders. Fast parallel operations on search trees. *CoRR*, abs/1510.05433, 2015. arXiv:1510.05433.

4 Yaroslav Akhremtsev and Peter Sanders. Fast parallel operations on search trees. In *IEEE International Conference on High Performance Computing (HiPC)*, 2016.

5 Lars Arge, Michael T. Goodrich, and Nodari Sitchinava. Parallel external memory graph algorithms. In *IPDPS*, pages 1–11. IEEE, 2010. doi:10.1109/IPDPS.2010.5470440.

6 Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001. doi:10.1007/S00224-001-0004-Z.

7 Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

8 Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. *arXiv preprint 1903.04650*, 2019.

9 Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 89–102, 2020. doi:10.1145/3350755.3400227.

10 Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 16–26, 1998. doi:10.1145/277651.277660.

11 Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999. doi:10.1145/324133.324234.

12 Nairen Cao and Jeremy T Fineman. Parallel exact shortest paths in almost linear work and square root depth. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 4354–4372. SIAM, 2023. doi:10.1137/1.9781611977554.CH166.

13 Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. PaC-trees: Supporting parallel and compressed purely-functional collections. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2022.

14 Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 918–934, 2019. doi:10.1145/3314221.3314598.

15 Laxman Dhulipala, David Eisenstat, Jakub Lacki, Vahab Mirrokni, and Jessica Shi. Hierarchical agglomerative graph clustering in poly-logarithmic depth. *arXiv preprint:2206.11654*, 2022.

16 Laxman Dhulipala, Charlie McGuffey, Hongbo Kang, Yan Gu, Guy E Blelloch, Phillip B Gibbons, and Julian Shun. Sage: Parallel semi-asymmetric graph algorithms for nvrams. *arXiv preprint*, 2019. arXiv:1910.12310.

17 Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. Efficient stepping algorithms and implementations for parallel shortest paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 184–197, 2021. doi:10.1145/3409964.3461782.

18 Yan Gu, Zachary Napier, and Yihan Sun. Analysis of work-stealing and parallel cache complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 46–60. SIAM, 2022. doi:10.1137/1.9781611977059.4.

19 Simon Marlow et al. Haskell 2010 language report. *Available online*, 2010. URL: http://www.haskell.org/ (May 2011).

20 Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, New York, NY, USA, 1999.

21 Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. Many sequential iterative algorithms can be parallel and (nearly) work-efficient. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2022.

22 Yihan Sun and Guy Blelloch. Implementing parallel and concurrent tree structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 447–450, 2019. doi:10.1145/3293883.3302576.

23 Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment (PVLDB)*, 13(2):211–225, 2019. doi:10.14778/3364324.3364334.

24 Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. PAM: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018.

25 Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

26 Letong Wang, Xiangyun Ding, Yan Gu, and Yihan Sun. Fast and space-efficient parallel algorithms for influence maximization. *Proceedings of the VLDB Endowment (PVLDB)*, 17(3), 2023. doi:10.14778/3632093.3632104.


## A Background for join-based Algorithms on Binary Trees

For completeness, we briefly introduce the background of join-based algorithms on binary trees. For binary trees, the $\mathsf{Join}(L, k, R)$ function takes two search trees $T_L$ and $T_R$, and one key $k$ in the middle, such that $\forall k_1 \in T_L$ and $k_2 \in T_R$, $k_1 < k < k_2$, and returns a valid balanced tree containing all elements in $L \cup \{k\} \cup R$. As the inversion to Join, a $\mathsf{Split}(T, k)$ is defined to return $\langle T_L, k', T_R \rangle$, where $T_L$ ($T_R$) contains all keys in $T$ smaller (larger) than $k$, and $k'$ is the node containing $k$ if $k \in T$, and null otherwise. In this case, an example of the Union algorithm can be implemented as illustrated in Figure 1. Note that the Union algorithm is independent of the balancing schemes used, as long as the Join and Split functions are properly supported.

## B Related Work

The Join operation was first proposed by Tarjan [25] on red-black trees. In 1992, Adams used Join as a primitive for weight-balanced trees to implement set functions for union, intersection, and difference [1, 2]. Adams' algorithms were proposed in an international competition for the Standard ML community, which is about implementations on "set of integers". Adams won the "elegance" award in the competition, meaning that the algorithm is elegant yet reasonably efficient among all participants in the competition. However, Adams' original paper only informally argued a loose work bound, which is $O(m + n)$ for operating on two sets with sizes $n$ and $m \leq n$.

Blelloch and Reid-Miller [10] first considered using join-based algorithms on parallel data structures. Their paper proposed parallel algorithms on treaps based on Join, and first show that these algorithms have $O(m \log(n/m))$ work and polylogarithmic span (with high probability). The algorithms are work-efficient because the work matches the lower bound of comparisons needed to combine two ordered sets. Later Blelloch et al. [7] studied the join-based algorithms, and showed work-efficiency and $O(\log n \log m)$ span for other balancing schemes, including AVL trees, read-black trees, and weight-balanced trees. In 2020 [9], the span bound for weight-balanced trees was improved to $O(\log n)$.

In 2015, Akhremtsev and Sanders [4] proposed join-based algorithms on $(a, b)$-trees. To make the algorithm more space-efficient and I/O-friendly in practice, Dhulipala et al. proposed C-trees [14] and PaC-trees [13]. The idea is to allow each tree node (or just the

leaves) to store a block of entries instead of one. Both C-trees and PaC-trees are binary. While their goal is to make the algorithms more cache-friendly in practice, no I/O bounds were shown. To the best of our knowledge, our work is the first to study the I/O cost of the join-based algorithms.

To achieve a better span bound, some techniques of our algorithms are inspired by existing work, such as using a list of pointers based on the height of subtrees [4], and using $\sqrt{m+n}$ way divide-and-conquer with a work-inefficient base case [9]. Our solution provides non-trivial and effective adaptions to use these ideas to achieve improved I/O bounds on B-trees.

Due to the elegance of the algorithm, many libraries later used the join-based algorithms for their implementation for sequences, sets, or maps (e.g., Haskell [19], the LEDA library [20], PAM [24], Aspen [14], CPAM [13]).

## C B-Way Join Analysis

We will eventually show the I/O span of the B-Way-Join operation is $O(h_{\max} - h_{\min})$, and the I/O work is $O(B \cdot (h_{\max} - h_{\min}))$, where $h_{\max}$ and $h_{\min}$ are the maximum and minimal tree heights among the input trees. First, we must show that the height of the output tree of each recursive call does not increase the height of the input trees by more than a constant, as this could lead to an increase in the I/O work and span.

▶ **Lemma 5.** *Suppose we have an unbalanced B-tree $T$ with $b$ children and $b-1$ keys. Let $C$ be the subset of children of $T$ with height $h$, and the rest of the children have height $h-1$. Then we can rebalance the tree to have a valid B-tree of height $h$ or $h+1$. If the tree is of height $h+1$, then the root will have at most $|C|$ keys.*

**Proof.** For each child $c \in C$, we can bring the keys up to the root node and reattach the children of $c$ as children of the root node. Suppose $c$ contains $b'$ keys, then the number of keys at the root and the number of children both increase by $b'$.

The increase in the number of keys is straightforward. The number of children increases by the same amount because bringing $c$ up adds $b'+1$ children, and we lost a child by bringing up $c$.

Performing this operation across all nodes in $C$ will result in an at most $|C|B$ keys at the root. If the root node also contained $B$ keys initially, then there would be $(|C|+1)B$ keys at the root, which would then be split into a tree of height $h+1$, and with at most $|C|$ keys at the root.

We know that the split will only ever occur once since $|C|$ is upperbounded to be at most $B$, thus a second split would not occur. ◀

▶ **Lemma 6.** *Suppose we have a list of trees $T_1, \ldots, T_{b+1}$ and keys $k_1, \ldots, k_b$, such that the height of each tree is either $h$ and the root contains at most $B$ keys, or $h+1$ and the root contains at most 1 key, except for $T_1$ which may contain 2 keys. Then we can join the trees such that the node at level $h+1$ contains at most $b + |H_i|$ keys, where $H_i$ is a subset of trees $T_1, \ldots, T_i$ with height $h+1$. Note that if a tree is of height $h$, then the number of keys at the node of level $h+1$ is defined to be 0.*

**Proof.** We prove the lemma by induction for $i = 1, \ldots, b+1$. The base case trivially holds for $i = 1$, where if the height of the first tree is $h$, then the number of keys at the node of level $h+1$ is 0. If the height of the first tree is $h+1$, then the number of keys at the node of level $h+1$ is at most 2, maintaining the stated bound. We assume the claim is true for $i$, which means the first $i$ trees produces B-tree $T^*$ with at most $i - 1 + |H_i|$ keys at the node of level $h+1$.

If $T^*$ is of height $h$, and the height of $T_{i+1}$ is $h$, then the join will result in at most 1 key at level $h+1$, if $T^*$ and $T_{i+1}$ contain $B$ keys each. In this case, $T^*$ will be made the left child, $T_{i+1}$ will be made the right child, and $k_i$ will be the key at the root.

If $T^*$ is of height $h+1$ and $T_{i+1}$ is of height $h$, then the join will cause $T_{i+1}$ to be the right child of $T^*$, and $k_i$ will be the right most key of $T^*$. This increases the number of keys at the node of level $h+1$ by 1, and thus the bound is maintained. The same logic applies if $T^*$ is of height $h$ and $T_{i+1}$ is of height $h+1$.

If instead both trees are of height $h+1$, then the join will cause the trees to be merged at the same height, and the number of keys increases by 1 plus the number of keys in $T_{i+1}$. Recall in the statement of the lemma that every tree except for $T_1$ contains at most 1 key at the root. Therefore, we can bound the increase by at most 2 keys, and we can see the bound is maintained since $i$ increase by 1 and $|H_i|$ also increases by 1. ◀

▶ **Theorem 7.** *Let $T_1, \ldots, T_{b+1}$ and $k_1, \ldots, k_b$ be the set of input trees and keys respectively, where $b \leq B$. If $h$ is the height of the tallest tree in $T$, then after performing* B-Way-Join*$(T, k)$, the height of the output tree will be either $h$ or $h+1$.*

**Proof.** We prove the theorem by induction. In the base case, if all trees in $T_{1,\ldots,b+1}$ are of height $h$, joining them maintains the theorem trivially. We create a new root node $v$ containing all keys of $k$, and each tree in $T_{1,\ldots,b+1}$ is made a child of $v$. If any child node contains less than $B/2$ keys, then we can use a standard rebalancing algorithm to fix it. This may reduce the number of keys in $v$ to less than $b$, but it will never increase. If all the keys in $v$ are removed, then the output tree will also be of height $h$. Otherwise, it will be $h+1$.

Then we prove the inductive step. Let $T'_{0,1,\ldots}$ be the set of trees which contain children that are the result of a recursive call, i.e. the tallest trees in $T_{1,\ldots,b+1}$. Due to the inductive hypothesis, all the trees will have a right most child of height either $h-1$ or $h$. The first tree may also have a left most child of height $h$. Due to the possible discrepancy in heights, we must rebalance these trees. We know from Lemma 5, that we can rebalance these trees to have heights of $h$ or $h+1$. If the height of a tree in $T'$ is $h+1$, then we know there is at most 1 key at the root, except for $T'_0$ which may have 2 keys at the root.

According to Lemma 6, we can see that merging the trees in $T'$ will result in a tree of height $h$ or $h+1$. If the tree is of height $h$, then the proof is complete. Otherwise, we must ensure the root node has at most $B$ keys. We know the root node has at most $b' + |H_{b'}|$ keys, where $b' = |T'|$. We know that $b' + |H_{b'}| \leq b$, since in order for a tree to have height $h+1$, it must have been passed into the recursive call with at least one other tree as part of the input set. Therefore, in order for $|H_{b'}|$ to increase by 1, the value of $b'$ must have decreased by at least 1. Therefore, the number of keys at the root is upper bounded by $b$, and thus cannot overflow again, maintaining the height of the tree $h+1$. ◀

▶ **Theorem 4.1** (B-Way Join Analysis). *Let $T_1, T_2, \ldots, T_{b+1}$ be a set of B-trees, with the largest tree height $h_{\max}$ and the shortest tree height $h_{\min}$, and $k_1, k_2, \ldots, k_b$ be a set of separator keys, where $b \leq B$. The* B-Way-Join *operation can be performed in $O(h_{\max} - h_{\min})$ I/O span and $O(b \cdot (h_{\max} - h_{\min}))$ I/O work.*

**Proof.** We can see each separator key is assigned a thread, which is then responsible for traversing down a pair of input trees until it reaches the point where all trees are of the same height, which is a traversal of length of at most $O(h_{\max} - h_{\min})$.

At each level of the tree, each thread will perform at most $O(1)$ I/Os. We know from Theorem 7 at each level of the tree, the resulting subtree will always stay either the same height or increase by 1. Which means no additional I/Os must be performed in order to compensate for the increased height of the tree, resulting in an I/O span of $O(h_{\max} - h_{\min})$.

The I/O work comes readily from the I/O span. ◀

## D Multiway Split: Cost Analysis

▶ **Theorem 1** (Multiway Split Cost). *We can split a B-tree $T$ by $k_1, \ldots, k_d$ keys in parallel with $O(\log_B d + \log_B n)$ I/O span and $O(d \log_B n)$ I/O work, where $n$ is $\sum_{i=1}^{d} |T_i|$, in the Fork-Join I/O Model.*

**Proof.** The algorithm spawns $d+1$ threads, where each thread performs a search down the tree. Each thread will perform $O(\log_B n)$ I/Os, in order to perform the search and copying the keys and pointers to the children takes $O(1)$ I/Os per node.

When performing the B-Way-Join operation, recall that the I/O span of the B-Way-Join operation is $O(h_{\max} - h_{\min})$, which is the difference in height between the tallest and shortest trees being joined together, and that the output tree must be of height at least $h_{\max}$.

We will be performing B-Way-Join along the path from the first divergence to each key. This means if at least one key is within the range of the split, then every B-Way-Join occurs with trees with height difference at most $O(1)$, and satisfying the I/O span bound of $O(\log_B n)$.

For the cases where a node contains keys within the range, but some of their ancestor's does not, we may use an amortized argument to charge a constant I/O for each node which does not contain keys within the range, which would otherwise not incur any cost. Then when joining with trees of height difference larger than a constant, we use the charge to pay for the extra I/O. The height difference can only be at most the number of consecutive skipped B-Way-Join operations.

For the I/O work bounds, observe that for each B-Way-Join operation, at most two trees being joined are of different heights, which are the trees along the search path to the split key. Therefore, the I/O work bounds is $O(1)$ instead of $O(B)$ for each B-Way-Join operation, which gives us the I/O work bound of $O(d \log_B n)$ I/Os. ◀

## E Multiway Join Cost Analysis

▶ **Lemma 8.** *The cost of preprocessing $d$ B-trees such that each tree $T$ contains a list of pointers to all nodes along their left and right spines is $O(\log_B n + \log_B d)$ I/O span and $O(d \log_B n)$ I/O work, where $n$ is the total number of keys in all trees.*

We can see that the fuse operations are completely independent of each other, and since each thread is able to access the number of keys that will be copied to the same tree ahead of them, then this process is able to occur in $O(1)$ I/O span and $O(B)$ I/O work, which gives us the following lemma.

▶ **Lemma 9.** *The cost of fusing $B$ trees in the B-Way-Join-Fast operation is $O(1)$ I/O span and $O(B)$ I/O work.*

Next we show the following lemma, so that we can guarantee that after the first round of divisions, no overloaded node will push more than a single key up to their parent node.

▶ **Lemma 10.** *After fusing $B$ trees together in the B-Way-Join-Fast operation, we can guarantee after one round of dividing the overloaded nodes, that each subsequent overloaded node may push up at most one key to their parent node.*

**Proof.** Recall that we are joining at most $B$ trees together. Since the root and all nodes it may fuse with contains at most $B$ keys, then including the paired key which the tree is associated with, each tree contributes at most 1 key to be pushed up to their parent node. In other words, for every tree that a node fuses with, they contribute at most 1 key to push to the parent per node.

Therefore, after a single round of divisions, even in the worst case where all keys were pushed up to a single node, this node would contain at most $2B$ keys, which would only push up a single key to the parent. ◀

▶ **Lemma 11.** *The cost of dividing the nodes up the tree in the B-Way-Join-Fast operation is $O(\log_2 \log_B n)$ I/O span and $O(B \log_B n)$ I/O work.*

**Proof.** The first set of divide operations cost $O(1)$ I/O span and $O(B)$ I/O work, since at most $B$ new nodes are created, which can all be done in parallel.

Then we gather the number of keys in each node along the path from the root to every overloaded node. This takes $O(\log_B \log_B n)$ I/O span and $O(B \log_B n)$ I/O work, as we can use the list of pointers to jump directly to their assigned node along the path, since the path must consist of only nodes which were along the spines of a fused tree. There are at most $B$ overloaded nodes, which means we create $B$ arrays for the paths to the root.

We then perform the prefix sum operation along each of these arrays, which takes $O(\log_2 \log_B n)$ I/O span and $O(B \log_B n)$ I/O work. Finally, using the information provided by the prefix sum, each node can be divided in a $O(1)$ I/O for each thread, which gives us our desired results. ◀

▶ **Lemma 12.** *The cost of updating the list of pointers in the B-Way-Join-Fast operation is $O(\log_B \log_B n)$ I/O span and $O(B \log_B \log_B n)$ I/O work.*

**Proof.** Each list of pointers is a standard sequential B-tree. Therefore, the split operation takes $O(\log_B \log_B n)$ I/O span and work for each list. There are $B$ lists of pointers, which gives us $O(B \log_B \log_B n)$ I/O work.

Then by using the B-Way-Join operation from Section 4.1, we can join these trees in $O(\log_B \log_B n)$ I/O span and $O(B \log_B \log_B n)$ I/O work. ◀

Now we can prove the main theorem of this section.

▶ **Theorem 13.** *We can join $T_1, \ldots, T_d$ B-trees and $k_1, \ldots, k_d$ keys together in parallel with $O(\log_B d \cdot \log_2 \log_B n + \log_B n)$ I/O span and $O(d \log_B n)$ I/O work, where $n$ is $\sum_{i=1}^{d} |T_i|$, in the Fork-Join I/O Model.*

**Proof.** We get from Lemma 8 that the cost of preprocessing is $O(\log_B n + \log_B d)$ I/O span and $O(d \log_B n)$ I/O work.

The Multi-Join operation performs $O(\log_B d)$ rounds of B-Way-Join-Fast operations. This and preprocessing gives us the I/O span of $O(\log_B d \cdot \log_2 \log_B n + \log_B n)$.

To analyze the I/O work cost, we can see that the total number of B-Way-Join-Fast operations reduces by a factor of $B$ each round. Therefore, we calculate the total I/O work cost from the following equation

$$\sum_{i=1}^{\log_B d} B^i \log_B n = O(d \log_B n).$$ ◀

## F Prefix Sum

The prefix sums problem is a fundamental aspect of parallel computing. The applications for this problem extends to a wide variety of domains, including for the Parallel Joinable B-Tree.

▶ **Definition 14.** *Given an ordered set $A$ of $n$ elements, the prefix sums operation returns an ordered set $A'$ of $n$ elements, such that $B[i] = \sum_{j=0}^{i} A[j]$ for all $0 \leq i < n$.*

We show that if the input set $A$ is located in contiguous main memory, then the prefix sums problem can be solved in the Fork-Join I/O Model in $O(\log_2 n)$ I/O span and $O(n)$ I/O work. By using the Gather operation, $n$ non-contiguous elements can be gathered into a single contiguous array in $O(\log_B n)$ I/O span and $O(n)$ I/O work, thus the above result also holds for non-contiguous input sets.

▶ **Lemma 15.** *The prefix sum problem can be solved in the Fork-Join I/O Model in $O(\log_2 n)$ I/O I/O span and $O(n)$ I/O work.*

**Proof.** The Fork-Join I/O Model solution to the prefix sums problem is performed by simulating the optimal I/O work efficient PRAM algorithm. The algorithm performs two passes: building a tree bottom-up and then traversing the tree top-down. Each element in $A$ is a leaf node, ordered from left to right, and we combine the elements in pairs to form a binary tree, where each node is the sum of its two children. Naturally this costs $O(\log_2 n)$ I/O span and $O(n)$ I/O work. Then from top down the tree, we pass the sum of the left child to the right child, summing up the left child's values until we reach the leaf, which then sums up the values of the left children passed. This top-down traversal also costs $O(\log_2 n)$ I/O span and $O(n)$ I/O work.

The tree can be implemented as an array, which would take advantage of the locality of the data. However, for the sake of simplicity, we will not use this optimization in our implementation as it does not affect the bounds of our algorithm. ◀

## G Union Algorithm: Base Case Cost

▶ **Lemma 16.** *The Work-Inefficient-Union algorithm takes $O(\log_B m \cdot \log_2 \log_B n + \log_B n)$ I/O span and $O(m \log_B n)$ I/O work.*

**Proof.** The cost of the Work-Inefficient-Union algorithm is dominated by the Multi-Join operation, which comes readily from Theorem 13, where $d = m$. This gives us the I/O span $O(\log_B m \cdot \log_2 \log_B n + \log_B n)$ and I/O work $O(m \log_B n)$. ◀

▶ **Lemma 2** (Union Algorithm Base Case Cost)**.** *The Work-Inefficient-Union algorithm takes $O(\log_B m \cdot \log_2 \log_B n + \log_B n)$ I/O span and $O(m \log_B n)$ I/O work.*

**Proof.** Suppose we have $k$ base cases, then the total cost of all base cases in the union algorithm is $\sum_{i=1}^{k} O(m_i(\log_B n_i + B))$, where $m_i$ and $n_i$ are the number of keys in the $i$-th base case. Recall that for each base case, $m_i < \sqrt{n_i + m_i}$.

From Lemma B.2 from Blelloch et al. [8], we know that $\sum_{i=1}^{k} m_i \log n_i = O(m \log n)$, where $\sum_{i=1}^{k} m_i = m, \sum_{i=1}^{k} n_i = n$, and for all $i$, $m_i \leq \sqrt{n_i + m_i}$. In the proof, the base of the logarithm is independent of the result, thus we can use the same proof to show that the I/O work is $O(m \log_B(\frac{n}{m}))$. ◀

## H Additional Pseudocode for Multiway Split and Join

The pseudocode for the Multi-Split, Multi-Join, Join-With-Tall, and Divide-Node algorithms are ommitted from this version due to space constraints. However, they are provided in the full version of this paper. [2]

## I Additional Set Operations

The Intersection and Difference operations are similar to the Union algorithm under the join-based framework. Hence, due to the space limit, we postpone the details of these two operations here in the appendix. The main difference lies in how subtrees are joined back.

▶ **Theorem 17.** *Given two B-trees with sizes $m$ and $n \geq m$, there exists a parallel algorithm that returns a new B-tree containing the intersection and difference of the two input trees in $O\big(m \log_B\big(\frac{n}{m}\big)\big)$ I/O work, $O(\log_B m \cdot \log_2 \log_B n + \log_B n)$ and I/O span, where $B$ is the block size.*

**Proof.** We first modify the Multi-Join operation slightly to take in an array $d$ boolean values, where each boolean value indicates whether the corresponding key should be included in the final result. Suppose $k_i$ is not to be part of the final result. Then we instead find the largest key in $T_i$, denoted as $k_i'$, and split on $T_i$ using $k_i'$. Then we replace $k_i$ with $k_i'$, and perform the join operation as normal.

In the Intersection operation, after searching for the $d-1$ split keys, we check if all $d-1$ keys are in both trees. If a key is in both trees, then we mark the key to be included in the final result. Then we perform the Multi-Split operation as normal, make our recursive calls, then use the modified Multi-Join operation to combine the results, excluding any keys which did not appear in both trees.

The base case of the Intersection operation is also very similar to the Work-Inefficient-Union algorithm. Let $|T_1| \geq |T_2|$, as we can always swap the two trees if this is not the case. Then we use the keys from $T_2$ to search for the keys in $T_1$. Any keys which are not found in $T_1$ are discarded. All remaining keys are then joined together using the traditional Multi-Join operation.

For the Difference operation, if a split key is found to be in $T_1$ and $T_2$ during the main recursive algorithm, then it is marked as excluded, and all the same steps as the Intersection operation are performed. For the base case, we use a slightly different approach depending on which tree is larger. If $|T_1| \geq |T_2|$, then we split $T_1$ using the keys in $T_2$, and join with none of the keys from $T_2$. Otherwise, we use all keys in $T_1$ to search for the keys in $T_2$, and join all keys from $T_1$ which were not found in $T_2$.

As such, the same analysis used to get the bounds from the Union algorithm can also be applied to the Intersection and Difference operations and achieve the same bounds . ◀

[2] The full version of this paper is available at `https://arxiv.org/abs/2510.20053`.