Parallel Longest Increasing Subsequence and van Emde Boas Trees

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Ziyang Men
UC Riverside
zmen002@ucr.edu

Zheqi Shen
UC Riverside
zshen055@ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

Zijin Wan
UC Riverside
zwan019@ucr.edu

ABSTRACT

This paper studies parallel algorithms for the longest increasing subsequence (LIS) problem. Let n be the input size and k be the LIS length of the input. Sequentially, LIS is a simple problem that can be solved using dynamic programming (DP) in $O(n \log n)$ work. However, parallelizing LIS is a long-standing challenge. We are unaware of any parallel LIS algorithm that has optimal $O(n \log n)$ work and non-trivial parallelism (i.e., $\tilde{O}(k)$ or $o(n)$ span).

This paper proposes a parallel LIS algorithm that costs $O(n \log k)$ work, $\tilde{O}(k)$ span, and $O(n)$ space, and is much simpler than the previous parallel LIS algorithms. We also generalize the algorithm to a weighted version of LIS, which maximizes the weighted sum for all objects in an increasing subsequence. To achieve a better work bound for the weighted LIS algorithm, we designed parallel algorithms for the van Emde Boas (vEB) tree, which has the same structure as the sequential vEB tree, and supports work-efficient parallel batch insertion, deletion, and range queries.

We also implemented our parallel LIS algorithms. Our implementation is light-weighted, efficient, and scalable. On input size 10^9, our LIS algorithm outperforms a highly-optimized sequential algorithm (with $O(n \log k)$ cost) on inputs with $k \leq 3 \times 10^5$. Our algorithm is also much faster than the best existing parallel implementation by Shen et al. (2022) on all input instances.

CCS CONCEPTS

• **Theory of computation** → **Design and analysis of algorithms**; **Parallel algorithms**; *Shared memory algorithms*;

KEYWORDS

parallel algorithms, longest increasing subsequence, van Emde Boas tree, dynamic programming, parallel data structure

ACM Reference Format:
Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3558481.3591069

1 INTRODUCTION

This paper studies parallel algorithms for classic and weighted longest increasing subsequence problems (LIS and WLIS, see definitions below). We propose a ***work-efficient parallel LIS algorithm*** with $\tilde{O}(k)$ span, where k is the LIS length of the input. Our WLIS algorithm is based on a new data structure that ***parallelizes the famous van Emde Boas (vEB) tree*** [74]. Our new algorithms improve existing theoretical bounds on the parallel LIS and WLIS problem, as well as enable simpler and more efficient implementations. Our parallel vEB tree supports work-efficient batch insertion, deletion and range query with polylogarithmic span.

Given a sequence $A_{1..n}$ and a comparison function on the objects in A, the LIS of A is the longest subsequence (not necessarily contiguous) in A that is strictly increasing (based on the comparison function). In this paper, we use LIS to refer to both the longest increasing subsequence of a sequence, and the problem of finding such an LIS. LIS is a fundamental problem and has extensive applications (e.g., [5, 28, 30, 31, 43, 60, 62, 79]). In this paper, we use n to denote the input size and k to denote the LIS length of the input. LIS can be solved by dynamic programming (DP) using the following DP recurrence (more details in Sec. 2).

$$dp[i] = \max(1, \max_{j<i, A_j<A_i} dp[j] + 1) \quad (1)$$

Sequentially, LIS is a straightforward textbook problem [29, 39]. We can iteratively compute $dp[i]$ using a search structure to find $\max_{j<i, A_j<A_i} dp[j]$, which gives $O(n \log n)$ work. However, in parallel, LIS becomes challenging both in theory and in practice. In theory, we are unaware of parallel LIS algorithms with $O(n \log n)$ work and non-trivial parallelism ($o(n)$ or $\tilde{O}(k)$ span). In practice, we are unaware of parallel LIS implementations that outperform the sequential algorithm on general input distributions. *We propose new LIS algorithms with improved work and span bounds in theory, which also lead to a more practical parallel LIS implementation.*

Our work follows some recent research [13–15, 17–19, 34, 41, 45, 61, 64, 65] that directly parallelizes sequential iterative algorithms. Such algorithms are usually simple and practical, given their connections to sequential algorithms. To achieve parallelism in a "sequential" algorithm, the key is to identify the ***dependences*** [18, 19, 64, 65] among the objects. In the DP recurrence of LIS, processing an object x ***depends on*** all objects $y < x$ before it, but does not need to wait for objects before it with a larger or equal value.

An "ideal" parallel algorithm should process all objects in a proper order based on the dependencies—it should 1) process as many objects as possible in parallel (as long as they do not depend on each other), and 2) process an object only when it is ***ready*** (all objects it depends on are finished) to avoid redundant work. More formally, we say an algorithm is ***round-efficient*** [64] if its span is $\tilde{O}(D)$ for a computation with the longest logical dependence length D. In LIS, the logical dependence length given by the DP recurrence is the LIS length k. We say an algorithm is ***work-efficient*** if its work is asymptotically the same as the best sequential algorithm. Work-efficiency is *crucial in practice*, since nowadays, the number of processors on one machine (tens to hundreds) is much

SPAA '23, June 17–19, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591069

smaller than the problem size. A parallel algorithm is less practical if it significantly blows up the work of a sequential algorithm.

Unfortunately, there exists no parallel LIS algorithm with both work-efficiency and round-efficiency. Most existing parallel LIS algorithms are not work-efficient [37, 52, 53, 57, 58, 63, 64, 70], or have $\tilde{\Theta}(n)$ span [4]. We review more related work in Sec. 7.

Our algorithm is based on the parallel LIS algorithm and the ***phase-parallel framework*** by Shen et al. [64]. We refer to it as the SWGS algorithm, and review it in Sec. 2. The phase-parallel framework defines a ***rank*** for each input object as the length of LIS ending at it (the dp value in Eq. (1)). Note that an object only depends on lower-rank objects. Hence, the phase-parallel LIS algorithm processes all objects based on the increasing order of ranks. However, the SWGS algorithm takes $O(n \log^3 n)$ work *whp*, $\tilde{O}(k)$ span, and $O(n \log n)$ space, and is quite complicated. In the experiments, the overhead in work and space limits the performance. On a 96-core machine and input size of 10^8, SWGS becomes slower than a sequential algorithm when the LIS length $k > 100$.

In this paper, we propose a parallel LIS algorithm that is ***work-efficient*** ($O(n \log k)$ work), ***round-efficient*** ($\tilde{O}(k)$ span) and ***space-efficient*** ($O(n)$ space), and is ***much simpler than previous parallel LIS algorithms*** [53, 64]. Our result is summarized in Thm. 1.1.

Theorem 1.1 (LIS). *Given a sequence A of size n and LIS length k, the longest increasing subsequence (LIS) of A can be computed in parallel with $O(n \log k)$ work, $O(k \log n)$ span, and $O(n)$ space.*

We also extend our algorithm to the ***weighted LIS (WLIS)*** problem, which has a similar DP recurrence as LIS but maximizes the weighted sum instead of the number of objects in an increasing subsequence.

$$dp[i] = w_i + \max(0, \max_{j<i, A_j<A_i} dp[j]) \tag{2}$$

where w_i is the weight of the i-th input object. We summarize our result in Thm. 1.2.

Theorem 1.2 (WLIS). *Given a sequence A of size n and LIS length k, the weighted LIS of A can be computed using $O(n \log n \log \log n)$ work, $O(k \log^2 n)$ span, and $O(n \log n)$ space.*

Our primary techniques to support both LIS and WLIS rely on better data structures for 1D or 2D ***prefix min/max queries*** in the phase-parallel framework. For the LIS problem, our algorithm efficiently identifies all objects with a certain rank using a ***parallel tournament tree*** that supports 1D dynamic prefix-min queries, i.e., given an array of values, find the minimum value for each prefix of the array. For WLIS, we design efficient data structures for 2D dynamic "prefix-max" queries, which we refer to as dominant-max queries (see more details in Sec. 4). Given a set of 2D points associated with values, which we refer to as their *scores*, a dominant-max query returns the largest score to the bottom-left of a query point. Using dominant-max queries, given an object x in WLIS, we can find the maximum dp value among all objects that x depends on. We propose two solutions focusing on theoretical and practical efficiency, respectively. In practice, we use a ***parallel range tree*** similar to that in SWGS, which results in $O(n \log^2 n)$ work and $\tilde{O}(k)$ span for WLIS. In theory, we parallelize the ***van Emde Boas (vEB) tree*** [74] and integrate it into range trees to achieve a better work bound for WLIS.

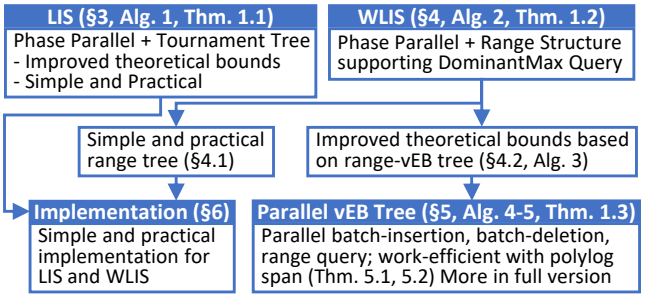


Figure 1: Outline and contributions of this paper.

The van Emde Boas (vEB) tree [74] is a famous data structure for priority queues and ordered sets on integer keys, and is introduced in many textbooks (e.g., [27]). To the best of our knowledge, our algorithm is the **first parallel version of vEB trees**. We believe our algorithm is of independent interest in addition to the application in WLIS. We note that it is highly non-trivial to redesign and parallelize vEB trees because the classic vEB tree interface and algorithms are inherently sequential. Our parallel vEB tree supports a general ordered set abstract data type on integer keys in $[0, U)$ with bounds stated below. We present more details in Sec. 5.

Theorem 1.3 (Parallel vEB Tree). *Let $\mathcal{U}$ be a universe of all integers in range $[0, U)$. Given a set of integer keys from $\mathcal{U}$, there exists a data structure that has the same organization as the sequential vEB tree, and supports:*

- *single-point insertion, deletion, lookup, reporting the minimum (maximum) key, and reporting the predecessor and successor of an element, all in $O(\log \log U)$ work, using the same algorithms for sequential vEB trees;*
- *BatchInsert(B) and BatchDelete(B) that insert and delete a sorted batch $B \subseteq \mathcal{U}$ in the vEB tree with $O(|B| \log \log U)$ work and $O(\log U \log \log U)$ span;*
- *Range(k_L, k_R) that reports all keys in range $[k_L, k_R]$ in $O((1+m) \log \log U)$ work and $O(\log U \log \log U)$ span, where m is the output size.*

Our LIS algorithm and the WLIS algorithm based on range trees are simple to program, and we expect them to be the algorithms of choice in implementations in the parallel setting. We tested our algorithms on a 96-core machine. Our implementation is *light-weighted, efficient and scalable*. Our LIS algorithm outperforms SWGS in all tests, and is faster than highly-optimized sequential algorithms [50] on reasonable LIS lengths (e.g., up to $k = 3 \times 10^5$ for $n = 10^9$). To the best of our knowledge, this is the *first parallel LIS implementation that can outperform the efficient sequential algorithm in a large input parameter space*. On WLIS, our algorithm is up to 2.5× faster than SWGS and 7× faster than the sequential algorithm for small k values. We believe the performance is enabled by the *simplicity* and *theoretical-efficiency* of our new algorithms.

We note that there exist parallel LIS algorithms [22, 53] with better worst-case span bounds than our results in theory. We highlight the *simplicity, practicality, and work-efficiency* of our algorithms, as well as the *parallel vEB trees and the extension to the WLIS problem*. We summarize the contributions of this paper as follows.

Theory: 1) Our LIS and WLIS algorithms improve the existing bounds. Our LIS algorithm is the first work- and space-efficient

parallel algorithm with non-trivial parallelism ($\tilde{O}(k)$ span). 2) We design the first parallel version of vEB trees, which supports work-efficient batch-insertion, batch-deletion and range queries with polylogarithmic span.

Practice: Our LIS and WLIS algorithms are highly practical and simple to program. Our implementations outperform the state-of-the-art parallel implementation SWGS on all tests, due to better work and span bounds. Our code is available on GitHub [42].

Due to the page limit, we provide the full version of this paper [**?**] to present complete analysis and more experimental results.

2 PRELIMINARIES

Notation and Computational Model. We use $O(f(n))$ *with high probability (*whp*)* (in n) to mean $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$. $\tilde{O}(f(n))$ means $O(f(n) \cdot polylog(n))$. We use $\log n$ as a short form for $1 + \log_2(n+1)$. For an array or sequence A, we use A_i and $A[i]$ interchangeably as the i-th object in A, and use $A[i..j]$ or $A_{i..j}$ to denote the i-th to the j-th objects A.

We use the ***work-span model*** in the classic multithreaded model with ***binary-forking*** [6, 14, 21]. We assume a set of threads that share the memory. Each thread acts like a sequential RAM plus a `fork` instruction that forks two child threads running in parallel. When both child threads finish, the parent thread continues. A parallel-for is simulated by `fork` for a logarithmic number of steps. A computation can be viewed as a DAG (directed acyclic graph). The ***work W*** of a parallel algorithm is the total number of operations in this DAG, and the ***span (depth) S*** is the longest path in the DAG. An algorithm is ***work-efficient*** if its work is asymptotically the same as the best sequential algorithm. The randomized work-stealing scheduler can execute such a computation in $W/P + O(S)$ time *whp* in W on P processor cores [6, 21, 40]. Our algorithms can also be analyzed on PRAM and have the same work and span bounds.

Longest Increasing Subsequence (LIS). Given a sequence $A_{1..n}$ of n input objects and a comparison function $<$ on objects in A, $A'_{1...m}$ is a subsequence of A if $A'_i = A_{s_i}$, where $1 \leq s_1 < s_2 < \ldots s_m \leq n$. The ***longest increasing subsequence*** (LIS) of A is the longest subsequence A^* of A where $\forall i < n, A^*_i < A^*_{i+1}$. Throughout the paper, we use n to denote the input size, and k to denote the LIS length of the input.

LIS can be solved using dynamic programming (DP) with the DP recurrence in Eq. (1). Here $dp[i]$ (called the ***dp value*** of object i) is the LIS length of $A_{1...i}$ ending with A_i.

The LIS problem generalizes to the ***weighted LIS (WLIS) problem*** with DP recurrence in Eq. (2). Sequentially, both LIS and weighted LIS can be solved in $O(n \log n)$ work. This is also the lower bound [35] w.r.t. the number of comparisons. For (unweighted) LIS, there exists an $O(n \log k)$ sequential algorithm [50]. When the input sequence only contains integers in range $[1, n]$, one can compute the LIS in $O(n \log \log n)$ work using a vEB tree. In our work, we assume general input and only use comparisons between input objects. Note that although we use vEB trees in WLIS, we will only use it to organize the indexes of the input sequence (see details in Sec. 5). Therefore, our algorithm is still comparison-based and works on any input type.

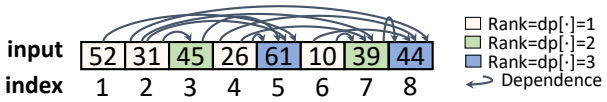


Figure 2: An input for LIS, the dependences and ranks. An object depends on all objects before it and is smaller than it. The rank of an object is the LIS length ending at it, which is also its *dp* value.

Dependence Graph [18, 19, 64, 65]. In a sequential iterative algorithm, we can analyze the logical *dependences* between iterations (objects) to achieve parallelism. Such dependences can be represented in a DAG, called a *dependence graph* (DG). In a DG, each vertex is an object in the algorithm. An edge from u to v means that v can be processed only when u has been finished. We say v ***depends*** on u in this case. Fig. 2 illustrates the dependences in LIS. We say an object is ***ready*** when all its predecessors have finished. When executing a DG with depth D, we say an algorithm is ***round-efficient*** if its span is $\tilde{O}(D)$. In LIS, the dependence depth given by the DP recurrence is the LIS length k. We note that round-efficiency does not guarantee optimal span, since round-efficiency is with respect to a given DG. One can design a different algorithm with a shallower DG and get a better span.

Phase-Parallel Algorithms and SWGS Algorithm [64]. The high-level idea of the phase-parallel algorithm is to assign each object x a ***rank***, denoted as $rank(x)$, indicating the earliest phase when the object can be processed. In LIS, the rank of each object is the length of the LIS ending with it (the *dp* value computed by Eq. (1)). We also define the ***rank of a sequence*** A as the LIS length of A. An object only depends on other objects with lower ranks. The phase-parallel LIS algorithm [64] processes all objects with rank i (in parallel) in round i. We call the objects processed in round i the ***frontier*** of this round. An LIS example is given in Fig. 2.

The SWGS algorithm uses a *wake-up scheme*, where each object can be processed $O(\log n)$ times *whp*. It also uses a range tree to find the frontiers both in LIS and WLIS. In total, this gives $O(n \log^3 n)$ work *whp*, $O(k \log^2 n)$ span, and $O(n \log n)$ space. Our algorithm is also based on the phase-parallel framework but avoids the wake-up scheme to achieve better bounds and performance.

3 LONGEST INCREASING SUBSEQUENCE

We start with the (unweighted) LIS problem. Our algorithm is also based on the phase-parallel framework [64] but uses a much simpler idea to make it work-efficient. The work overhead in the SWGS algorithm comes from two aspects: range queries on a range tree and the wake-up scheme. The $O(\log n)$ space overhead comes from the range tree. Therefore, we want to 1) use a more efficient (and simpler) data structure than the range tree to reduce both work and space, and 2) wake up and process an object only when it is ready to avoid waking up an object multiple times.

Our algorithm is based on a simple observation in Lemma 3.1 and the concept of ***prefix-min*** objects (Definition 3.1). Recall that the *rank* of an object A_i is exactly its *dp value*, which is the *length of LIS ending at* A_i.

Definition 3.1 (Prefix-min Objects). *Given a sequence $A_{1..n}$, we say A_i is a* **prefix-min** *object if for all $j < i$, we have $A_i \leq A_j$, i.e., A_i is (one of) the smallest object among $A_{1..i}$.*

Algorithm 1: The parallel (unweighted) LIS algorithm

Input: A sequence $A_{1..n}$
Output: All dp values (ranks) of $A_{1..n}$

```
1  int rank[1..n]                          // rank[i]: the LIS length ending at A_i.
2  int T[1..(2n − 1)]                      // T: the (implicit) tournament tree.
3  r ← 0                                   // r is the current round.
4  Initialize the tournament tree T
5  Function LIS(sequence A_{1..n})
6      while T[1] ≠ +∞ do                  // T is not empty.
7          r ← r + 1
8          ProcessFrontier()               // process the r-th frontier
9      return rank[1..n]
10 Function ProcessFrontier()
11     PrefixMin(1, +∞)                    // Process all prefix-min objects
   // Deal with subtree rooted at T[i]. Find objects x s.t.: 1) x ≤ any object
      before it, and 2) x ≤ LMin. Collect such objects in a binary tree.
12 Function PrefixMin(int i, int LMin)
13     if T[i] > LMin then return NIL
14     if i ≥ n then                       // Found a leaf node in the frontier.
15         rank[i] ← r                     // Set its rank as r.
16         T[i] ← +∞                       // Remove the object.
17     else                                // An internal node. Process two children in parallel.
18         in parallel:
19             L ← PrefixMin(2i, LMin)
20             R ← PrefixMin(2i + 1, min(LMin, T[2i]))
21         T[i] ← min(T[2i], T[2i + 1])
```

Lemma 3.1. *In a sequence A, an object A_i has rank 1 iff. A_i is a prefix-min object. An object A_i has rank r iff. A_i is a prefix-min object after removing all objects with ranks smaller than r.*

We use Fig. 3 to illustrate the intuition of Lemma 3.1, and prove it in the full version of this paper [?]. Based on Lemma 3.1, we can design an efficient yet simple phase-parallel algorithm for LIS (Alg. 1). For simplicity, we first focus on computing the dp values (ranks) of all input objects. We discuss the algorithm to output a specific LIS for the input sequence in the full version of this paper. The main loop of Alg. 1 is in Lines 6–8. In round r, we identify the frontier $\mathcal{F}_r$ as all the prefix-min objects and set their dp values to r. We then remove the objects in $\mathcal{F}_r$ and repeat. Fig. 3 illustrates Alg. 1 by showing the "prefix-min" value pre_i for each object, which is the smallest value up to each object. Note that this sequence pre_i is not maintained in our algorithm but is just used for illustration. In each round, we find and remove all objects A_i with $A_i = pre_i$. Then we update the prefix-min values pre_i and repeat. In round r, all identified prefix-min objects have rank r.

To achieve work-efficiency, we cannot re-compute the prefix-min values of the entire sequence after each round. Our approach is to design a parallel *tournament tree* to help identify the frontiers. Next, we briefly overview the tournament tree and then describe how to use it to find the prefix-min objects efficiently.

Tournament tree. A tournament tree $\mathcal{T}$ on n records is a complete binary tree with $2n-1$ nodes (see Fig. 4). It can be represented implicitly as an array $\mathcal{T}[1..(2n-1)]$. The last n elements are the leaves, where $\mathcal{T}[i]$ stores the $(i-n+1)$-th record in the dataset. The first $n-1$ elements are internal nodes, each storing the minimum value of its two children. The left and right children of $\mathcal{T}[i]$ are $\mathcal{T}[2i]$ and $\mathcal{T}[2i+1]$, respectively. We will use the following theorem about the tournament tree.

Theorem 3.1. *(Parallel Tournament Trees [14, 32]) A tournament tree can be constructed from n elements in $O(n)$ work and $O(\log n)$ span. Given a set S of m leaves, in the tournament tree with size n, the number of ancestors of all the nodes in S is $O(m\log(n/m))$.*

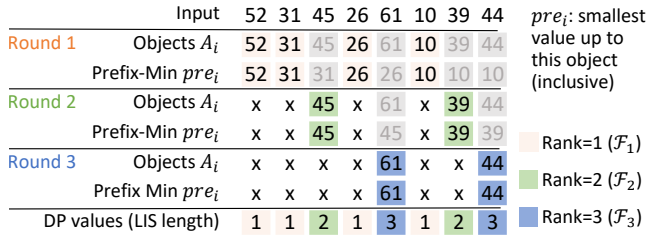


Figure 3: An illustration of Alg. 1. The figure also shows pre_i as the smallest object up to this object (inclusive). If $A_i = pre_i$, it is a prefix-min object. In round r, Alg. 1 finds all prefix-min objects, sets their DP values as r, removes them, and updates the pre_i values.

A tournament tree can be constructed by recursively constructing the left and right trees in parallel, and updating the root value.

Using Tournament Tree for LIS. We use a tournament tree $\mathcal{T}$ to efficiently identify the frontier and dynamically remove objects (see Alg. 1). $\mathcal{T}$ stores all input objects in the leaves. We always round up the number of leaves to a power of 2 to make it a full binary tree. Each internal node stores the minimum value in its subtree. When we traverse the tree at $\mathcal{T}[i]$, if the smallest object to its left is smaller than $\mathcal{T}[i]$, we can skip the entire subtree. Using the internal nodes, we can maintain the minimum value before any subtree and skip irrelevant subtrees to save work.

In particular, the function ProcessFrontier finds all prefix-min objects from $\mathcal{T}$ by calling PrefixMin starting at the root. PrefixMin(i, *LMin*) traverses the subtree at node i, and finds all leaves v in this subtree s.t. 1) v is no more than any leaf before v in this subtree, and 2) v is no more than *LMin*. The argument *LMin* records the smallest value in $\mathcal{T}$ before the subtree at $\mathcal{T}[i]$. If the smallest value in subtree $\mathcal{T}[i]$ is larger than *LMin*, we can skip the entire subtree (Line 13), because no object in this subtree can be a prefix-min object (they are all larger than *LMin*). Otherwise, there are two cases. The first case is when $\mathcal{T}[i]$ is a leaf (Lines 14–16). Since $\mathcal{T}[i] \le$ *LMin*, it must be a prefix-min object. Therefore, we set its dp value as the current round number r (Line 15) and remove it by setting its value as $+\infty$ (Line 16). In the second case, when $\mathcal{T}[i]$ is an internal node (Lines 17–21), we can recurse on both subtrees in parallel to find the desired objects (Line 18). For the left subtree, we directly use the current *LMin* value. For the right subtree, we need to further consider the minimum value in the left subtree. Therefore, we take the minimum of the current *LMin* and the smallest value in the left subtree ($\mathcal{T}[2i]$), and set it as the *LMin* value of the right recursive call. After the recursive calls return, we update $\mathcal{T}[i]$ (Line 21) because some values in the subtree may have been removed (set to $+\infty$). We present an example in Fig. 4, which illustrates finding the first frontier for the input in Fig. 3.

We now prove the cost of Alg. 1 in Thm. 3.2.

Theorem 3.2. *Alg. 1 computes the LIS of the input sequence A in $O(n\log k)$ work and $O(k\log n)$ span, where n is the length of the input sequence A, and k is the LIS length of A.*

Proof. Constructing $\mathcal{T}$ takes $O(n)$ work and $O(\log n)$ span. We then focus on the main loop (Lines 6–8) of the algorithm. The algorithm runs in k rounds. In each round, ProcessFrontier recurses for $O(\log n)$ steps. Hence, the algorithm has $O(k\log n)$ span.

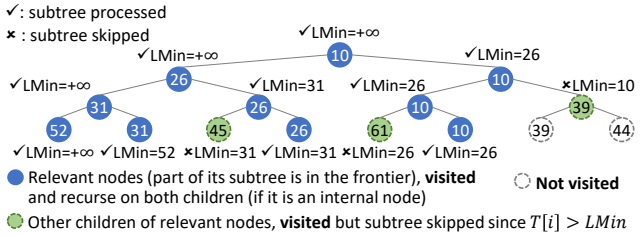


Figure 4: The parallel tournament tree for Alg. 1. The leaves store the input $A_{1..n}$. An internal node stores the minimum value in its subtree. The figure illustrates finding the first frontier for Fig. 3. The algorithm recursively traverses the tree from the root and maintains a *LMin* value for each subtree as the smallest value before this subtree. If *LMin* is smaller than the value at the subtree root, we skip the subtree. For example, the smallest value before the green node 39 is $LMin = 10$. Therefore, no leaves in this subtree can be a prefix-min object, so this subtree is skipped.

Next, we show that the work of ProcessFrontier in round r is $O(m_r \log(n/m_r))$ work, where $m_r = |\mathcal{F}_r|$ is the number of prefix-min objects identified in this round. First, note that visiting a tournament tree node has a constant cost, so the work is asymptotically the number of nodes visited in the algorithm. We say a node is ***relevant*** if at least one object in its subtree is in the frontier. Based on Thm. 3.1, there are $O(m_r \log(n/m_r))$ relevant nodes.

If Line 14 is executed (i.e., Line 13 does not return), the smallest object in this subtree is no more than *LMin* and must be a prefix-min object, and this node is relevant. Other nodes are also visited but skipped by Line 13. Executing Line 13 for subtree i means that i's parent executed Line 17, so i's parent is relevant. This indicates that a node is visited either because it is relevant, or its parent is relevant. Since every node has at most two children, the number of visited nodes is asymptotically the same as all relevant nodes. which is $O(m_r \log(n/m_r))$. Hence, the total number of visited nodes is:

$$\sum_{r=1}^{k} m_r \log(n/m_r) \le \sum_{i=1}^{k} (n/k) \log(n/(n/k)) = n \log k$$

The last step uses the concavity of the function $f(x) = x \log_2(1 + \frac{n}{x})$. This proves the work bound of the algorithm. □

Note that the work bound of Thm. 3.2 is parameterized on the LIS length k. For small k, the work can be $o(n \log n)$. For example, if the input sequence is strictly decreasing, Alg. 1 only needs $O(n)$ work because the algorithm will find all objects in the first round in $O(n)$ work and finishes.

4 WEIGHTED LONGEST INCREASING SUBSEQUENCE

A nice property of the unweighted LIS problem is that the *dp* value is the same as its rank. In round r, we simply set the *dp* values of all objects in the frontier as r. This is not true for the weighted LIS (WLIS) problem, and we need additional techniques to handle the weights. Inspired by SWGS, our WLIS algorithm is built on an efficient data structure $\mathcal{R}$ supporting 2D ***dominant-max*** queries: for a set of 2D points (x_i, y_i) each with a ***score*** s_i, the dominant-max query (q_x, q_y) asks for the maximum score among all points in its lower-left corner $(-\infty, q_x) \times (-\infty, q_y)$. We will use such a data structure to efficiently compute the *dp* values of all objects.

Algorithm 2: The parallel weighted LIS algorithm

Input: A sequence $A_{1..n}$. Object A_i has weight w_i
Output: The DP values $dp[1..n]$ for each object A_i.

1 **Struct** *Point*
2 **int** x, y // y = index, $x = A_y$.
3 **int** dp // The DP value of A_y, used as the score of the point.
4 Point $p[1..n]$
5 **int** $dp[1..n]$ // $dp[i]$: the DP value of A_i.
6 **Struct** *RangeStruct*⟨*Point*⟩
7 Stores points $\langle x_i, y_i, dp_i \rangle$ with coordinate (x_i, y_i) and score dp_i
8 Supports DominantMax(p, q): return the maximum score (the $dp[\cdot]$ value) among all points (x_i, y_i) where $x_i < p$ and $y_i < q$
9 Supports Update(B), where $B = \{\langle x_i, y_i, dp_i \rangle\}$ is a batch of points: update the score of each point (x_i, y_i) to dp_i
10 *RangeStruct* $\mathcal{R}$ // Any data structure that supports DominantMax
11 Run Alg. 1. Sort the *rank* array and get all the k frontiers $\mathcal{F}_{1..k}$. $\mathcal{F}_i$ contains the indexes of all objects with rank i.
12 **parallel-foreach** $A_i \in A$ **do** $p[i] = \langle A_i, i, 0 \rangle$
13 Construct $\mathcal{R}$ from $p[\cdot]$
14 **for** $i \leftarrow 1$ to k **do**
15 **parallel-foreach** $j \in \mathcal{F}_i$ **do** // A_j is an object with rank i
16 $dp[j] \leftarrow \mathcal{R}$.DominantMax$(A_j, j) + w_j$
17 $B \leftarrow \{\langle A_j, j, dp[j] \rangle : j \in \mathcal{F}_i\}$
18 $\mathcal{R}$.Update(B)
19 **return** $dp[\cdot]$

We present our WLIS algorithm in Alg. 2. We view each object as a 2D point (A_i, i) with score $dp[i]$, and use a data structure $\mathcal{R}$ that supports dominant-max queries to maintain all such points. Initially $dp[i] = 0$. We call A_i and i as the x- and y-coordinate of the point, respectively. Given the input sequence, we first call Alg. 1 to compute the rank of each object and sort them by ranks to find each frontier $\mathcal{F}_i$. This can be done by any deterministic parallel sorting with $O(n)$ work and $O(\log^2 n)$ span. We then process all the frontiers in order. When processing $\mathcal{F}_i$, we compute the *dp* values for all $j \in \mathcal{F}_i$ in parallel, using $dp[j] = \max_{j' < j, A_{j'} < A_j} dp[j']$. This can be done by the dominant-max query on $\mathcal{R}$ (Line 16), which reports the highest score (*dp* value) among all objects in the lower-left corner of the object j. Finally, we update the newly-computed *dp* values to $\mathcal{R}$ (Line 18) as their scores.

The efficiency of this algorithm then relies on the data structure to support dominant-max. We will propose two approaches to achieve practical and theoretical efficiency, respectively. The first one is similar to SWGS and uses range trees, which leads to $O(n \log^2 n)$ work and $\tilde{O}(k)$ span for the WLIS problem. By plugging in an existing range-tree implementation [68], we obtain a simple parallel WLIS implementation that significantly outperforms the existing implementation from SWGS. The details of the algorithm are in Sec. 4.1, and the performance comparison is in Sec. 6. We also propose a new data structure, called the Range-vEB, to enable a better work bound ($O(n \log n \log \log n)$ work) for WLIS. Our idea is to redesign the inner tree in range trees as a *parallel vEB tree*. We elaborate on our approach in Sec. 4.2 and 5.

4.1 Parallel WLIS based on Range Tree

We can use a parallel ***range tree*** [8, 67] to answer dominant-max queries. A range tree [8] is a nested binary search tree (BST) where the *outer tree* is an index of the x-coordinates of the points. Each tree node maintains an *inner tree* storing the same set of points

in its subtree but keyed on the y-coordinates (see Fig. 5). We can let each inner tree node store the maximum score in its subtree, which enables efficient dominant-max queries. In particular, for the outer tree, we can search $(-\infty, q_x)$ on the x-coordinates. This gives $O(\log n)$ relevant subtrees in this range (called the ***in-range*** subtrees), and $O(\log n)$ relevant nodes connecting them (called the ***connecting*** nodes). In Fig. 5, when $q_x = 6.5$, the in-range inner trees are the inner trees of points $(2, 6)$ and $(5, 1)$, since their entire subtrees falls into range $(-\infty, 6.5)$. The connecting nodes are $(4, 5)$ and $(6, 4)$, as their x-coordinates are in the range, but only part of their subtrees are in the range. For each in-range subtree, we further search $(-\infty, q_y)$ in the inner trees to get the maximum score in this range, and consider it as a candidate for the maximum score. For each connecting node, we check if its y-coordinates are in the range $(-\infty, q_y)$, and if so, consider it a candidate. Finally, we return the maximum score among the selected candidates (both from the in-range subtrees and connecting nodes). Using the range tree in [14, 67, 68], we have the following result for WLIS.

Theorem 4.1. *Using a parallel range tree for the dominant-max queries, Alg. 2 computes the weighted LIS of an input sequence A in $O(n \log^2 n)$ work and $O(k \log^2 n)$ span, where n is the length of the input sequence A, and k is the LIS length of A.*

4.2 WLIS Using the Range-vEB Tree

We can achieve better bounds for WLIS using parallel van Emde Boas (vEB) trees. Unlike the solution based on parallel range trees, the vEB-tree-based solution is highly non-trivial. Given the sophistication, we describe our solution in two parts. This section shows how to solve parallel WLIS assuming we have a parallel vEB tree. Later in Sec. 5, we will show how to parallelize vEB trees.

We first outline our data structure at a high level. We refer to our data structure for the dominant-max query as the ***Range-vEB tree***, which is inspired by the classic range tree as mentioned in Sec. 4.1. The main difference is that the inner trees are replaced by ***Mono-vEB trees*** (defined below). Recall that in Alg. 2, the *RangeStruct* implements two functions DominantMax and Update. We present the pseudocode of Range-vEB for these two functions in Alg. 3, assuming we have parallel functions on vEB trees.

Similar to range trees, our Range-vEB tree is a two-level nested structure, where the outer tree is indexed by x-coordinates, and the inner trees are indexed by y-coordinates. For an outer tree node v, we will use S_v to denote the set of points in v's subtree and T_v as the inner tree of v. Like a range tree, the inner tree T_v also corresponds to the set of points S_v, but only the *staircase* of S_v (defined below). Since the y-coordinates are the indexes of the input, which are integers within n, we can maintain this staircase in a vEB tree. Recall that the inner tree stores the y-coordinates as the key and uses the dp values as the scores. For two points $p_1 = \langle x_1, y_1, dp_1 \rangle$ and $p_2 = \langle x_2, y_2, dp_2 \rangle$, we say p_1 ***covers*** p_2 if $y_1 < y_2$ and $dp_1 \geq dp_2$. For a set of points S, the ***staircase*** of S is the maximal subset $S' \subseteq S$ such that for any $p \in S'$, p is not covered by any points in S. In other words, for two input objects A_i and A_j in WLIS, we say A_i ***covers*** A_j if i comes before j and has a larger or equal dp value. This also means that no objects will use the dp value at j since A_i is strictly better than A_j. Therefore, we ignore such A_j in the inner trees, and refer to such a vEB tree maintaining the staircase of a

Algorithm 3: The parallel RangeStruct using Range-vEB trees

```
1  Structures Point and RangeStruct are defined in Alg. 2
2  Function DominantMax(q_x, q_y)
3      In the Range-vEB, find the range of (−∞, q_x), and let S_node be the
         set of connecting nodes and S_tree be the set of in-range inner
         (Mono-vEB) trees
       // For each in-range inner tree, find the max score up to coordinate q_y
4      parallel-foreach t_i ∈ S_tree do
5          ⟨·, ·, σ_i⟩ ← Pred(t_i, q_y)    // σ_i is the score of q_y's predecessor
       // For connecting nodes, check if the y-coordinates are smaller than q
         and get the maximum score for such points
6      foreach ⟨x, y, dp⟩ ∈ S_node s.t. y < q_y do
7          σ' = max(σ', dp)
8      return max(σ', max_i{σ_i})
9  Function Update(B)              // B is a list of points {⟨x_i, y_i, dp_i⟩}
10     Update ⟨x_i, y_i, dp_i⟩ in the outer Range-vEB tree
11     For each relevant inner (Mono-vEB) tree t_i, gather a list of points
         L_i ⊆ B to be added to t_i
12     Points in L_i are sorted by the y-coordinates
13     Let S_tree be the set of inner trees t_i that need new insertions
       // Refine L_i: Remove L_i[j] if any other point covers it
14     For each list L_i, remove L_i[j] if
15     • ∃ l < j, s.t. L_i[j].dp < L_i[l].dp, or
16     • π.dp ≥ L_i[j].dp, where π = Pred(t_i, L_i[j].y) is L_i[j]'s
         predecessor in the corresponding Mono-vEB tree t_i
17     parallel-foreach t_i ∈ S_tree do
           // Find elements in t_i that are covered by points in L_i
18         R ← t_i.CoveredBy(L_i)
19         t_i.BatchDelete(R)                    // Delete points in R
20         t_i.BatchInsert(L_i)                  // Insert points in L_i
```

dataset as a ***Mono-vEB*** tree. In a Mono-vEB tree, with increasing key (y_i), the score (dp values) must also be increasing.

Due to monotonicity, the maximum dp value in a Mono-vEB tree for all points with $y_i < q_y$ is exactly the score (dp value) of q_y's predecessor. Combining this idea with the dominant-max query in range trees, we have the dominant-max function in Alg. 3. We will first search the range $(-\infty, q_x)$ in the outer tree for the x-coordinates and find all in-range subtrees and connecting nodes. For each connecting node, we check if their y coordinates are in the queried range, and if so, take their dp values into consideration. For each in-range inner tree t_i, we call Pred query on the Mono-vEB tree and obtain the score (dp value) σ_i of this predecessor (Line 5). As mentioned, the value of σ_i is the highest score from this inner tree among all points with an index smaller than q_y. Finally, we take a max of all such results (all σ_i and those from connecting nodes), and the maximum among them is the result of the dominant-max query (Line 8). As the Pred function has cost $O(\log \log n)$, a single dominant-max query costs $O(\log n \log \log n)$ on a Range-vEB tree.

Querying dominant-max using a staircase is a known (sequential) algorithmic trick. However, the challenge is how to update (in parallel) the newly computed dp values in each round (the Update function) to a Range-vEB tree. We first show how to implement Update in Alg. 3 while assuming a parallel vEB tree. We later explain how to parallelize a vEB tree in Sec. 5.

Step 1. Collecting insertions for inner trees. Each point $p \in B$ may need to be added to $O(\log n)$ inner trees, so we first obtain a list L_i of points to be inserted for each inner tree t_i. This can be done by first marking all points in B in the outer tree $\mathcal{R}$, and (in

parallel) merging them bottom-up so that each relevant inner tree collects the relevant points in B. When merging the lists, we keep them sorted by the y-coordinates, the same as the inner trees.

Step 2. Refining the lists. Because of the "staircase" property, we have to first refine each list L_i to remove points that are not on the staircase. A point in $L_i[j]$ should be removed if it is covered by its previous point $L_i[j-1]$, or if any point in the Mono-vEB tree t_i covers it. The latter case can be verified by finding the predecessor π of $L_i[j].y$, and check if π has a larger or equal dp value than $L_i[j]$. If so, $L_i[j]$ is covered by $\pi \in t_i$, so we ignore $L_i[j]$. After this step, all points in $L[i]$ need to appear on the staircase in t_i.

Step 3. Updating the inner trees. Finally, for all involved subtrees, we will update the list L_i to t_i in parallel. Note that some points in L_i may cover (and thus replace) some existing points in t_i. We will first use a function CoveredBy to find all points (denoted as set R) in t_i that are covered by any point in L_i. An illustration of CoveredBy function is presented in the full version of this paper. We will then use vEB batch-deletion to remove all points in R from t_i. Finally, we call vEB batch-insertion to insert all points in L_i to t_i.

In Sec. 5, we present the algorithms CoveredBy, BatchDelete and BatchInsert needed by Alg. 2, and prove Thm. 1.3. Assuming Thm. 1.3, we give the proof of Thm. 1.2.

Proof of Thm. 1.2. We first analyze the work. We first show that the DominantMax algorithm in Alg. 3 takes $O(\log n \log\log n)$ work. In Alg. 3, Line 3 finds $O(\log n)$ connecting nodes and in-range inner trees, which takes $O(\log n)$ work. Then for all $O(\log n)$ in-range inner trees, we perform a Pred query in parallel, which costs $O(\log\log n)$. In total, this gives $O(\log n \log\log n)$ work for DominantMax. This means that the total work to compute the dp values in Line 16 in the entire Alg. 2 is $O(n \log n \log\log n)$.

We now analyze the total cost of Update. In one invocation of Update, we first find all keys for each inner tree t_i that appears in B. Using the bottom-up merge-based algorithm mentioned in Sec. 4.2, each merge costs linear work. Similarly, refining a list L_i costs linear work. Since each key in B appears in $O(\log n)$ inner tree, the total work to find and refine all L_i is $O(|B| \log n)$ for each batch, and is $O(n \log n)$ for the entire algorithm.

For each subtree, the cost of running CoveredBy is asymptotically bounded by BatchDelete. For BatchDelete and BatchInsert, note that the bounds in Thm. 1.3 show that the amortized work to insert or delete a key is $O(\log\log n)$. In each inner tree, a key can be inserted at most once and deleted at most once, which gives $O(n \log n \log\log n)$ total work in the entire algorithm.

Finally, the span of each round is $O(\log^2 n)$. In each round, we need to perform the three steps in Sec. 4.2. The first step requires finding the list of relevant subtrees for each element in the insertion batch B. For each element $b \in B$, this is performed by first searching b in the outer tree, and then merging them bottom-up so that each node in the outer tree will collect all elements in B that belong to its subtree. There are $O(\log n)$ levels in the outer tree, and each merge requires $O(\log n)$ span, so this first step requires $O(\log^2 n)$ span.

Step 2 will process all relevant lists in parallel (at most n of them). For each list, it calls Pred for each element in each list, and a filter algorithm at the end. The total span is bounded by $O(\log^2 n)$.

Step 3 requires calling batch insertion and deletion to update all relevant inner trees, and all inner trees can be processed in parallel.

x	: A w-bit integer x from universe $\mathcal{U}$, where $\mathcal{U} = [0, 2^w)$
$high(x)$	: high-bit of x, equals to $\lfloor x/2^{\lceil w/2 \rceil} \rfloor$
$low(x)$	: low-bit of x, equals to $(x \mod 2^{\lceil w/2 \rceil})$
$index(h, l)$	: The integer by concatenating high-bit h and low-bit l
$\mathcal{V}$	: A vEB (sub-)tree / the set of keys in this vEB tree
$\mathcal{V}.min$ ($\mathcal{V}.max$)	: The min (max) value in vEB tree $\mathcal{V}$
$\textsc{Pred}(\mathcal{V}, x)$	: Find the predecessor of x in vEB tree $\mathcal{V}$
$\textsc{Succ}(\mathcal{V}, x)$	: Find the successor of x in vEB tree $\mathcal{V}$
$\mathcal{V}.summary$	: The set of high-bits in vEB tree $\mathcal{V}$
$\mathcal{V}.cluster[h]$	: The subtree of $\mathcal{V}$ with high-bit h
$(^*)\mathcal{P}_{\mathcal{V},B}(x)$	: The survival predecessor of $x \in B$ in vEB tree $\mathcal{V}$ (used in Alg. 5). $\mathcal{P}(x) = \max\{y : y \in \mathcal{V} \setminus B, y < x\}$.
$(^*)\mathcal{S}_{\mathcal{V},B}(x)$	: The survival successor of $x \in B$ in vEB tree $\mathcal{V}$ (used in Alg. 5). $\mathcal{S}(x) = \min\{y : y \in \mathcal{V} \setminus B, y > x\}$.

Table 1: Notation for vEB trees. $(^*)$: We drop the subscript with clear context.

Based on the analysis above, the span for each batch insertion and deletion is $O(\log n \log\log n)$, which is also bounded by $O(\log^2 n)$.

Thus, the entire algorithm has span $O(k \log^2 n)$. We present the details of achieving the stated space bound in the full version of this paper by relabeling all points in each inner tree. □

Making Range-vEB Tree Space-efficient. A straightforward implementation of Range-vEB tree may require $O(n^2)$ space, as a plain vEB tree requires $O(U)$ space. There are many ways to make vEB trees space-efficient ($O(n)$ space when storing n keys); we discuss how they can be integrated in Range-vEB tree to guarantee $O(n \log n)$ total space in the full version of this paper.

5 PARALLEL VAN EMDE BOAS TREES

The van Emde Boas (vEB) tree [74] is a famous data structure that implements the ADTs of priority queues and ordered sets and maps for integer keys. For integer keys in the range 0 to U, single-point updates and queries cost $O(\log\log U)$, better than the $O(\log n)$ cost for BSTs or binary heaps. We review vEB trees in Sec. 5.1.

However, unlike BSTs or binary heaps that have many parallel versions [3, 10, 12, 14, 20, 32, 54, 68, 76, 77, 80], we are unaware of any parallel vEB trees. Even the sequential vEB tree is complicated (compared to most BSTs and heaps) to guarantee the doubly-logarithmic cost. Such complication adds to the difficulty of parallelizing updates (insertions and deletions) on vEB trees. Meanwhile, for queries, we note that vEB trees do not directly support range-related queries—when using vEB trees for ordered sets and maps, many applications heavily rely on repeatedly calling successors and/or predecessors, which is inherently sequential. Hence, we need to carefully redesign the vEB tree to achieve parallelism. In this section, we first review the sequential vEB tree and then present our parallel vEB tree to support the functions needed in Alg. 3.

5.1 Review of the Sequential vEB Tree

A van Emde Boas (vEB) tree [74] is a search tree structure with keys from a universe $\mathcal{U}$, which are integers from 0 to $U-1$. We usually assume the keys are w-bit integers (i.e., $U = 2^w$). A classic vEB tree supports insertion, deletion, lookup, reporting the min/max key in the tree, reporting the predecessor (Pred) and successor (Succ) of a key, all in $O(\log\log U)$ work. Other queries can be implemented

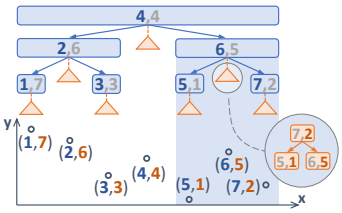


Figure 5: An illustration of a 2D range tree. The outer tree is indexed by x (blue), and inner trees are indexed by y (red).

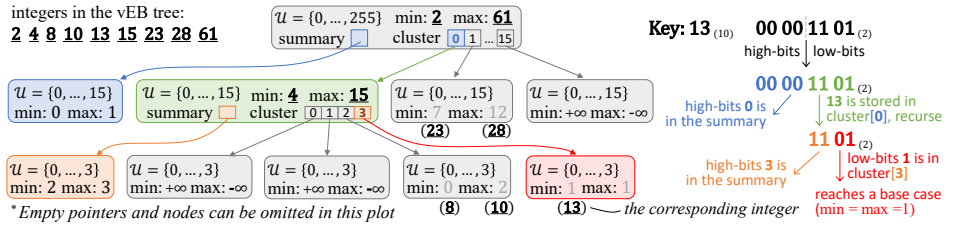


Figure 6: An example vEB tree with $U = 256$ and a demonstration on how 13 is stored. The vEB tree contains the set of keys $\{2, 4, 8, 10, 13, 15, 23, 28, 61\}$.

using these functions. For instance, reporting all keys in a range can be implemented by repeatedly calling Succ in $O(m \log\log U)$ work, where m is the output size.

A vEB tree stores a key using its binary bits as its index. We use $\mathcal{V}$ to denote a vEB tree, as well as the set of keys in this vEB tree. We present the notation for vEB trees in Tab. 1, and show an illustration of vEB trees in Fig. 6. We use 13 as an example to show how a key is decomposed and stored in the tree nodes. A vEB tree is a quadruple $(summary, cluster[\cdot], \min, \max)$. $\mathcal{V}.min$ and $\mathcal{V}.max$ store the minimum and maximum keys in the tree. When $\mathcal{V}$ is empty, we set $\mathcal{V}.min = +\infty$ and $\mathcal{V}.max = -\infty$. For the rest of the keys (other than min/max), their ***high-bits*** (the first $\lceil w/2 \rceil$ bits) are maintained recursively in a vEB tree, noted as $\mathcal{V}.summary$. In Fig. 6, the high-bits are the first 4 bits, and there are two different unique high-bits (0 and 1). They are maintained recursively in a vEB (sub-)tree $\mathcal{V}.summary$ (the blue box). For each unique high-bit, the relevant ***low-bits*** (the last $\lfloor w/2 \rfloor$ bits) are also organized as a vEB (sub-)tree recursively. In particular, the low-bits that belong to high-bit h are stored in a vEB tree $\mathcal{V}.cluster[h]$. In Fig. 6, five keys in $\mathcal{V}$ have high-bit 0 (4, 8, 10, 13, and 15). They are maintained in a vEB (sub-)tree as $\mathcal{V}.cluster[0]$ (the green box and everything below). Each subtree ($summary$ and all $cluster[\cdot]$) has universe size $O(\sqrt{U})$ (about $w/2$ bits). This guarantees traversal from the root to every leaf in $O(\log\log U)$ hops. Note that the min/max values of a vEB tree are not stored again in the summary or clusters. For example, in Fig. 6, at the root, $\mathcal{V}.min = 2$, and thus 2 is not stored again in $\mathcal{V}.cluster[0]$. Such design is crucial to guarantee doubly logarithmic work for Insert, Delete, Pred, and Succ.

Note that although we use "low/high-bits" in the descriptions, algorithms on vEB trees can use simple RAM operations to extract the corresponding bits, without any bit manipulation, as long as the universe size for each subtree is known. Due to the page limit, we refer the audience to the textbook [27] for more details about sequential vEB tree algorithms.

5.2 Our New Results

We summarize our results on parallel vEB tree in Thm. 1.3. Both batch insertion/deletion and range reporting are work-efficient—the work is the same as performing them on a sequential vEB tree. In Alg. 2, the key range $U = n$. Using the Range query, we can implement CoveredBy in Alg. 3 in $O(m' \log\log n)$ work and polylogarithmic span, where m' is the number of objects returned.

Similar to the sequential vEB tree, batch-insertion is relatively straightforward among the parallel operations. We present the algorithm and analysis in Sec. 5.2.1. Batch-deletion is more challenging, as once $\mathcal{V}.min$ or $\mathcal{V}.max$ is deleted, we need to replace it with a proper key k' stored in the subtree of $\mathcal{V}$. However, when finding the replacement k', we need to avoid the values in the deletion batch B and take extra care to handle the case when k' is the min/max of a cluster. We propose a novel technique: ***Survivor Mapping*** (see Definition 5.1) to resolve this challenge. The batch-deletion algorithm is illustrated in Sec. 5.2.2, and analysis in the full version of this paper. For range queries, we need to avoid the iterative solution (repeatedly calling Succ) since it is inherently sequential. Our high-level idea is to divide-and-conquer in parallel, but uses delicate amortization techniques to bound the extra work. Due to the page limit, we summarize the high-level idea of Range and CoveredBy in Sec. 5.2.3 and provide the details in the full version.

5.2.1 Batch Insertion. We show our batch-insertion algorithm in Alg. 4, which inserts a sorted batch $B \subseteq \mathcal{U}$ into $\mathcal{V}$ in parallel. Here we assume the keys in B are not in $\mathcal{V}$; otherwise, we can simply look up the keys in $\mathcal{V}$ and filter out those in $\mathcal{V}$ already. To achieve parallelism, we need to appropriately handle the high-bits and low-bits, both in parallel, as well as taking extra care to maintain the min/max values.

We first set the min/max values at $\mathcal{V}$ (Line 2–5). If $B.min < \mathcal{V}.min$, we update $\mathcal{V}.min$ by swapping it with $B.min$ (Line 3); similarly we update $\mathcal{V}.max$ (Line 4). Since we need the batch B sorted when adding $\mathcal{V}.min$ and/or $\mathcal{V}.max$ back to B (Line 5), we need to insert them to the correct position, causing $O(m)$ work. If B is not empty, we will insert the keys in B to $\mathcal{V}.summary$ and $\mathcal{V}.cluster$. We first find the new high-bits (not yet in $\mathcal{V}.summary$) from keys in B, and denote them as H' (Line 7). This step can be done by a parallel filter. For each new high-bit $h' \in H'$, we select the smallest key with high-bit h' and put them in an array B' (Line 8). We will insert them separately to initialize the new subtrees in $cluster$. In particular, we first insert the new high-bits H' to $\mathcal{V}.summary$ (Line 9). Then we process all $x \in B'$ in parallel and use its low-bit to initialize the min/max values of the new cluster $\mathcal{V}.cluster[high(x)]$ (Line 10–12). When reaching Line 13, all the keys in B' have been inserted, and all relevant $cluster[h]$ for $h \in H$ are non-empty.

We then insert the low-bits for the remaining keys in $B \setminus B'$. We will find all unique high-bits from $B \setminus B'$ as H. For each $h \in H$, we gather all relevant low-bits into an array $L[h]$ (Line 14). We then insert each $L[h]$ into $\mathcal{V}.cluster[h]$ (Line 16) in parallel.

The correctness of the algorithm can be shown by checking that all min/max values for each node are set up correctly. Next, we analyze the cost bounds of Alg. 4 in Thm. 5.1.

Theorem 5.1. *Inserting a batch of sorted keys into a vEB tree can be finished in $O(m \log\log U)$ work and $O(\log U)$ span, where m is batch size and $U = |\mathcal{U}|$ is the universe size.*

Algorithm 4: Batch Insertion Algorithm for vEB tree

Input: Batch of elements B in sorted order, vEB tree $\mathcal{V}$. $B \cap \mathcal{V} = \emptyset$
Output: A veb Tree $\mathcal{V}$ with all keys $x \in B$ inserted

```
1  Function BatchInsert(V, B)
2      S ← {V.min} ∪ {V.max}                                  // Backup min and max
3      V.min ← min{V.min, B.min}
4      V.max ← max{V.max, B.max}
5      B ← B ∪ S \ {V.min} \ {V.max}
6      if B ≠ ∅ then                                          // Deal with high-bits and low-bits of B
           // H' are the new high-bits
7          H' ← {high(x) | x ∈ B, V.cluster[high(x)] is empty}
           // For each new high-bit h', find the smallest key in B to form B'
8          B' ← {x_h' | ∀h' ∈ H', where x_h' = min_{y∈B, high(y)=h'} y}
9          BatchInsert(V.summary, H')                         // Insert H' to summary
10         parallel-foreach x ∈ B' do                         // Initialize each new high-bit
11             V.cluster[high(x)].min ← low(x)
12             V.cluster[high(x)].max ← low(x)
13         H ← {high(x) | ∀x ∈ B \ B'}                        // exclude keys in B'
14         L[h] ← {low(x) | ∀x ∈ B \ B', high(x) = h ∈ H}
15         parallel-foreach h ∈ H do                          // Insert to each cluster
16             BatchInsert(V.cluster[h], L[h])
```

Proof. Let $W(u,m)$ and $S(u,m)$ be the work and span of BatchInsert on a batch of size m and vEB tree with universe size u. In each invocation of BatchInsert, we need to restore the *min*/*max* values, find the high-bits in H' and H, initialize the clusters for the new high-bits, and gather the low-bits for each cluster.

All these operations cost $O(m)$ work and $O(\log m) = O(\log u)$ span. Then the algorithm makes at most $\sqrt{u}+1$ recursive calls, each dealing with a universe size $\sqrt{u}$. Hence, we have the following recurrence for work and span:

$$W(u,m) = \sum_{i=0}^{\sqrt{u}} W(\sqrt{u}, m_i) + O(m) \tag{3}$$

$$S(u,\cdot) = 2S(\sqrt{u},\cdot) + O(\log u) \tag{4}$$

Note that each key in B falls into at most one of the recursions, and thus $\sum_{i=0}^{\sqrt{u}} m_i = m \le u$. By solving them, we can get the claimed bound in the theorem. We solve them in the full version of this paper. Note that we assume an even total bits for u. If not, the number of subproblems and their size become $\sqrt{u/2}+1$ and $\sqrt{2u}$, respectively. One can check that the bounds still hold, the same as the sequential analysis. □

5.2.2 Batch Deletion. The function BatchDelete$(\mathcal{V}, B)$ deletes a batch of sorted keys $B \subseteq \mathcal{U}$ from a vEB tree $\mathcal{V}$. Let $m = |B|$ be the batch size. For simplicity, we assume $B \subseteq \mathcal{V}$. If not, we can first look up all keys in B and filter out those that are not in $\mathcal{V}$ in $O(m \log\log U)$ work and $O(\log m + \log\log U)$ span. We show our algorithm in Alg. 5. The main challenge to performing m deletions in parallel is to properly set the *min* and *max* values for each subtree t. When the *min*/*max* value of a subtree t is in B, we need to replace it with another key in its subtree that 1) does not appear in B, and 2) needs to be further deleted from the corresponding *cluster*$[\cdot]$ (recall that the *min*/*max* values of a subtree should not be stored in its children). To resolve this challenge, we keep the ***survival predecessor*** and ***survival successor*** for all $x \in B$ wrt. a vEB tree, defined as follows.

Definition 5.1 (Survivor Mapping). *Given a vEB tree $\mathcal{V}$ and a batch $B \subseteq \mathcal{V}$, the* **survival predecessor** *$\mathcal{P}(x)$ for $x \in B$ is the maximum key in $\mathcal{V} \setminus B$ that is smaller than x. If no such key exists, $\mathcal{P}(x) := -\infty$. Similarly, the* **survival successor** *$\mathcal{S}(x)$ for $x \in B$ is the minimum key in $\mathcal{V} \setminus B$ that is larger than x, and is $+\infty$ if no such key exists. $\langle \mathcal{P}, \mathcal{S} \rangle$ are called the* **survival mappings**.

Algorithm 5: Batch Deletion Algorithm for vEB tree

Input: A vEB tree $\mathcal{V}$ and a batch of keys $B \subseteq \mathcal{V}$ in sorted order
Output: Update $\mathcal{V}$ by deleting all keys $x \in B$

```
1  Function BatchDelete(V, B)
2      Initialize survival mappings P and S with respect to B and V
3      if B ≠ ∅ then BatchDeleteRecursive(V, B, P, S)
4  Function BatchDeleteRecursive(V, B, P, S)
       // Maintaining min/max of current tree
5      ⟨v_min, v_max⟩ ← ⟨V.min, V.max⟩
6      if v_min = B.min then                                  // if V.min ∈ B, it must be B.min
7          y ← S(B.min)
8          if y ≠ V.max and y ≠ +∞ then                       // if y is in the clusters
9              Delete y from V sequentially
10             ⟨P, S⟩ ← SurvivorRedirect(V, B, y, P, S)
11         V.min ← y
12     if v_max = B.max then ...                              // Mostly symmetric to Lines 7–11
13     B ← B \ {v_min} \ {v_max}
14     if V.max = −∞ and V.min ≠ +∞ then V.max ← V.min
15     if B ≠ ∅ then                                          // Recursively deal with the batch
16         H ← {high(x) | ∀x ∈ B}
17         L[h] ← {low(x) | high(x) = h ∈ H, ∀x ∈ B}
18         parallel-foreach h ∈ H do
19             ⟨P_h, S_h⟩ ← SurvivorLow(h, L[h], P, S)
20             BatchDeleteRecursive(V.cluster[h], L[h], P_h, S_h)
21         H' ← {h ∈ H | V.cluster[h] is empty}
22         ⟨P', S'⟩ ← SurvivorHigh(H', L, P, S)
23         BatchDeleteRecursive(V.summary, H', P_H', S_H')
   // Redirect the survival mapping P and S concerning elements in batch B
      after sequential deletion of y from vEB tree V
24 Function SurvivorRedirect(V, B, y, P, S)
25     ⟨p, s⟩ ← ⟨Pred(V, y), Succ(V, y)⟩
26     if p ∈ B then p ← P(p)
27     if s ∈ B then s ← S(s)
28     parallel-foreach x ∈ B do
29         if P(x) = y then P(x) ← p
30         if S(x) = y then S(x) ← s
31     return ⟨P, S⟩
   // Build survival predecessor P_h and successor S_h for elements in L[h]
32 Function SurvivorLow(h, L, P, S)
33     P_h ← ∅, S_h ← ∅
34     parallel-foreach l ∈ L[h] do
35         ⟨p, s⟩ ← ⟨P(index(h, l)), S(index(h, l))⟩
36         if high(p) = h and p ≠ V.min then P_h(l) ← low(p)
37         else P_h(l) ← −∞
38         if high(s) = h and s ≠ V.max then S_h(l) ← low(s)
39         else S_h(l) ← +∞
40     return ⟨P_h, S_h⟩
   // Build survival predecessor P' and successor S' for elements in H
41 Function SurvivorHigh(H, L, P, S)
42     P' ← ∅, S' ← ∅
43     parallel-foreach h ∈ H do
44         ⟨p, s⟩ ← ⟨P(index(h, min{L[h]})), S(index(h, max{L[h]}))⟩
45         if p ≠ V.min then P'(h) ← high(p) else P'(h) ← −∞
46         if s ≠ V.max then S'(h) ← high(s) else S'(h) ← +∞
47     return ⟨P', S'⟩
```

$\mathcal{P}(\cdot)$ and $\mathcal{S}(\cdot)$ are used to efficiently identify the new keys to replace a deleted key. For instance, if $\mathcal{V}.max \in B$ (then it must be $B.max$), we can update the value of $\mathcal{V}.max$ to $\mathcal{P}(B.max)$ directly.

Alg. 5 first initializes the survival mappings (Line 2) as follows. For each $x \in B$, we set (in parallel) $\mathcal{P}(x)$ as its predecessor in $\mathcal{V}$ if this predecessor is not in B, and set $\mathcal{P}(x) = -\infty$ otherwise. Then we compute prefix-max of $\mathcal{P}$, and replace the $-\infty$ values by the proper survival predecessor of x in $\mathcal{V}$.

The initial values of $\mathcal{S}$ can be computed similarly. We then use the BatchDeleteRecursive function to delete batch B from a vEB (sub-)tree $\mathcal{V}$ using the survival mappings, starting from the root. We use m as the batch size of *the current recursive call*, and u as the universe size of *the current vEB subtree*. The algorithm works in two steps: we first set the *min*/*max* values of the tree $\mathcal{V}$ properly, and then recursively deal with the *summary* and *cluster* of $\mathcal{V}$.

Restoring *min*/*max* values. We first discuss how to update $\mathcal{V}.min$ and $\mathcal{V}.max$ if they are deleted, in Line 5–13 of Alg. 5. We first duplicate $\mathcal{V}.min$ and $\mathcal{V}.max$ as v_{min} and v_{max} (Line 5), and then check whether $v_{min} \in B$ (Line 6). If so, we replace it with its survival successor (denoted as y on Line 7). If y is in the clusters ($y \neq \mathcal{V}.max$), y will be extracted from the corresponding cluster and become $\mathcal{V}.min$. To do so, we first delete y sequentially (Line 9), and the cost is $O(\log\log u)$. Then we redirect the survival mapping for keys in B using function SurvivorRedirect since their images may have changed (Line 10)—if any of them have survival predecessor/successor as y, they should be redirected to some other key in $\mathcal{V}$ (Line 29–30). In particular, if $\mathcal{P}(x)$ is y, it should be redirected to y's survival predecessor (Line 26). Similarly, if $\mathcal{S}(x)$ is y, it should be redirected to y's survival successor (Line 27). Regarding the cost of SurvivorRedirect, Line 25 (finding y's predecessor and successor) costs $O(\log\log u)$, but we can charge this cost to the previous sequential deletion on Line 9. The rest of this part costs $O(m)$ work and $O(\log m)$ span. After that, we set the new $\mathcal{V}.min$ value as y. The symmetric case applies to when $\mathcal{V}.max \in B$ (Line 12). We then exclude $v_{\min}$ and $v_{\max}$ from B on Line 13, since we have handled them properly. Finally, on Line 14, we deal with the particular case where only one key remains after deletion, in which case we have to store it twice in both $\mathcal{V}.min$ and $\mathcal{V}.max$.

Recursively dealing with the low/high bits. After we update $\mathcal{V}.min$ and $\mathcal{V}.max$ (as shown above), we will recursively update $\mathcal{V}.summary$ (high-bits) and $\mathcal{V}.cluster$ (low-bits), which requires the algorithm to construct the survival mappings for *summary* and each *cluster*. We first consider the low-bits for $cluster[h]$ and construct the survival mappings as $\mathcal{P}_h$ and $\mathcal{S}_h$ (Line 19). Given a key $x \in B$, where $high(x) = h$, the survival predecessor for its low-bits $\mathcal{P}_h(low(x))$ is the low-bits of its survival predecessor $low(\mathcal{P}(x))$ if x and $\mathcal{P}(x)$ have same high-bits h (Line 36). Otherwise $low(x)$ would become the smallest key in $\mathcal{V}.cluster[h]$ after removing B, therefore we map $\mathcal{P}(low(x))$ to $-\infty$ (Line 37). We can construct $\mathcal{S}(low(x))$ similarly (Line 38–39). Note that we have to exclude $\mathcal{V}.min$ and $\mathcal{V}.max$ since they do not appear in the clusters. Then we can recursively call BatchDeleteRecursive on the $cluster[h]$ using the survival mappings $\mathcal{P}_h$ and $\mathcal{S}_h$ (Line 20). In total, constructing all survival mappings for low-bits costs $O(m)$ work and $O(\log m)$ span.

We then construct the survival mapping $\mathcal{P}'$ and $\mathcal{S}'$ for high-bits (Line 22). Recall that the clusters of $\mathcal{V}$ contain all keys in $\mathcal{V} \setminus \{\mathcal{V}.min\} \setminus \{\mathcal{V}.max\}$. Therefore if $\mathcal{P}(x) = \mathcal{V}.min$, then $\mathcal{P}(high(x))$ should be mapped to $-\infty$. Otherwise let $y \in B$ be the maximum key except $\mathcal{V}.min$ and $\mathcal{V}.max$ with the same high-bits as x, then we have $\mathcal{P}(high(x)) = high(\mathcal{P}(y))$ by definition (Line 45). We can construct $\mathcal{S}(h)$ similarly (Line 46). The total cost of finding survival mappings for high-bits is also $O(m)$ work and $O(\log m)$ span.

Note that the high-bits cannot be processed in parallel with the low-bits, and has to be after the deletion of low-bits is finished. This is because only after deleting the low-bits, we know what high-bits need to be deleted in *summary*. We now analyze the cost of Alg. 5 in Thm. 5.2.

Theorem 5.2. *Given a vEB tree $\mathcal{V}$, deleting a sorted batch $B \subseteq \mathcal{V}$ costs $O(m \log\log U)$ work and $O(\log U \log\log U)$ span, where $m = |B|$ is the batch size and $U = |\mathcal{U}|$ is the universe size.*

Due to the page limit, we show the (informal) high-level ideas here and prove it in the full version of this paper. The span recurrence of the batch-deletion algorithm is similar to batch insertion, which indicates the same span bound. The work-bound proof is more involved. The challenge lies in that a key in B can be involved in both recursive calls on Line 23 and Line 20, which seemingly costs work-inefficiency. However, for each high-bit h to be deleted on the recursive call on Line 23, it indicates that the corresponding $cluster[h]$ will become empty after the deletion of low-bits. Therefore, the smallest low-bit among them must be exceptional and will be handled by the base cases on Lines 7–11. Therefore, for each key in B, only one of the recursive calls will be "meaningful". If the audience is familiar with (sequential) vEB trees, this is very similar to the sequential analysis—for the two recursive calls on the low- and high-bits, only one of them will be invoked in any single-point insertion/deletion, and the $O(\log\log U)$ bound thus holds. In the parallel version, we need to further analyze the cost on Lines 7–11 to restore the *min*/*max* values. We show a formal proof for Thm. 5.2 in the full version of this paper.

5.2.3 Range Query. Sequentially, the range query on vEB trees is supported by repeatedly calling Succ from the start until the range's end. However, such a solution is inherently sequential. Although we can find the start and end of the range directly in the tree, reporting all keys in the range to an array in parallel is difficult—vEB trees cannot maintain subtree sizes efficiently, so we cannot decide the size of the output array to be assigned to each subtree. In this paper, we design novel parallel algorithms for range queries on vEB trees and use them to implement CoveredBy in Alg. 3.

To achieve parallelism, our high-level idea is to use divide-and-conquer to split the range in the middle and search the two sub-ranges in parallel. Even though the partition can be unbalanced, we can still bound the recursion depth while amortizing the work for partitioning to the operations in the sequential execution. We collect results first in a binary tree and then flatten the tree into a consecutive array. Putting all pieces together, our range query has optimal work (same as a sequential algorithm) and polylogarithmic span. On top of that, we show how to implement CoveredBy, which also requires amortization techniques. The details are provided in the full version of this paper.

6 EXPERIMENTS

In addition to the new theoretical bounds, we also show the practicality of the proposed algorithms by implementing our LIS (Alg. 1)

and WLIS algorithms (Alg. 2 using range trees). Our code is scalable yet lightweight. We release our code on GitHub [42]. We use the experimental results to show how theoretical efficiency enables better performance in practice over the existing results. Our LIS implementation is based on Alg. 1, with additional consideration of a simple granularity control, which runs recursive calls sequentially when the winning tree size is smaller than 2^{16}. Our WLIS implementation is based on Alg. 2, and uses the same range tree implementation in SWGS [64].

Experimental Setup. We run all experiments on a 96-core (192-hyperthread) machine equipped with four-way Intel Xeon Gold 6252 CPUs and 1.5 TiB of main memory. Our implementation is in C++ with ParlayLib [11]. All reported numbers are the averages of the last ten runs among eleven repeated tests. The running time is reasonably stable, and we observe that the standard deviations are mostly within 10% (and always within 2% when running time is more than 5 seconds) of the running time.

Input Generator. We run experiments of input size $n = 10^8$ and $n = 10^9$ with varying ranks (LIS length k). We use two generators and refer to the results as the ***range*** pattern and the ***line*** pattern, respectively. The ***range*** pattern is a sequence consisting of integers randomly chosen from a range $[1, k']$. The values of k' upper bounds the LIS length. When k is large, and the largest possible rank of a sequence of size n is expected to be $2\sqrt{n}$ [48]. To generate inputs with larger ranks, we use a ***line*** pattern generator that draws A_i as $t \cdot i + s_i$ for a sequence $A_{1...n}$, where s_i is an independent random variable chosen from a uniform distribution. We vary t and s_i to achieve different ranks. For the weighted LIS problem, we always use random weights from a uniform distribution.

Baseline Algorithms. We compare to standard sequential LIS algorithms and the existing parallel LIS implementation from SWGS [64]. We also show the running time of our algorithm on one core to indicate the work of the algorithm. SWGS works on both LIS and WLIS problems with $O(n \log^3 n)$ work and $\tilde{O}(k)$ span, and we compare both of our algorithms (Alg. 1 and 2) with it.

For the LIS problem, we also use a highly-optimized sequential algorithm from [50] and call it Seq-BS. Seq-BS maintains an array B, where $B[r]$ is the smallest value of A_i with rank r. Note that B is monotonically increasing. Iterating i from 1 to n, we binary search A_i in B, and if $B[r] < A_i \le B[r+1]$, we set $dp[i]$ as $r+1$. By the definition of $B[\cdot]$, we then update the value $B[r+1]$ to A_i if A_i is smaller than the current value in $B[r+1]$. The size of B is at most k, and thus this algorithm has work $O(n \log k)$. This algorithm only works on the unweighted LIS problem.

For WLIS, we implement a sequential algorithm and call it Seq-AVL. This algorithm maintains an augmented search tree, which stores all input objects ordered by their values, and supports range-max queries. Iterating i from 1 to n, we simply query the maximum dp value in the tree among all objects with values less than A_i, and update $dp[i]$. We then insert A_i (with $dp[i]$) into the tree and continue to the next object. This algorithm takes $O(n \log n)$ work, and we implement it with an AVL tree.

Due to better work and span bounds, our algorithms are always faster than the existing parallel implementation SWGS. Our algorithms also outperform highly-optimized sequential algorithms up to reasonably large ranks (e.g., up to $k = 3 \times 10^5$ for $n = 10^9$). For our tests on 10^8 and 10^9 input sizes, our algorithm outperforms the sequential algorithm on ranks from 1 to larger than $2\sqrt{n}$. We believe this is the ***first parallel LIS implementation that can outperform the efficient sequential algorithm in a large input parameter space***.

Longest Increasing Subsequence (LIS). Fig. 7(a) shows the results on input size $n = 10^8$ with ranks from 1 to 10^7 using the line generator. For our algorithm and Seq-BS, the running time first increases with k getting larger because both algorithms have work $O(n \log k)$. When k is sufficiently large, the time drops slightly—larger ranks bring up better cache locality, as each object is likely to extend its LIS from an object nearby. Our algorithm is faster than the sequential algorithm for $k \le 3 \times 10^4$ and gets slower afterward. The slowdown comes from the lack of parallelism ($\tilde{O}(k)$ span). Our algorithm running on one core is only 1.4–5.5× slower than Seq-BS due to work-efficiency. With sufficient parallelism (e.g., on low-rank inputs), our performance is better than Seq-BS by up to 16.8×.

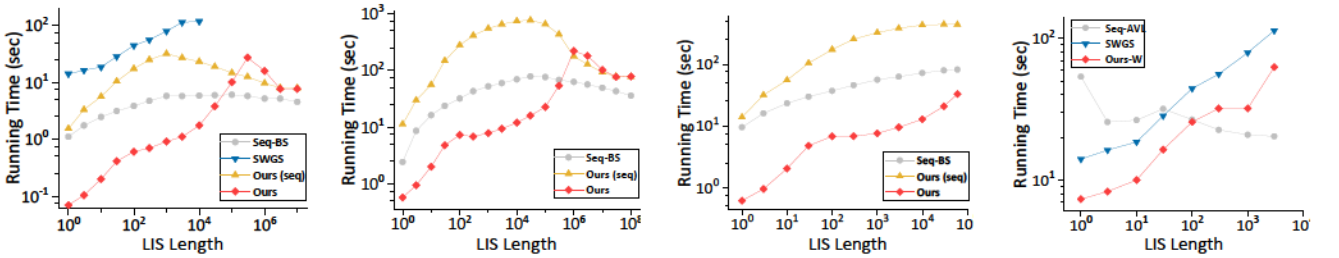
We only test SWGS on ranks up to 10^4 because it costs too much time for larger ranks. In the existing results, our algorithm is always faster than SWGS (up to 188×) because of better work and span. We believe the simplicity of code also contributes to the improvement.

We evaluate our algorithm on input size $n = 10^9$ with varied ranks from 1 to 10^8 using line the generator (see Fig. 7(b)) and with varied ranks from 1 to 6×10^4 using the range generator (see Fig. 7(c)). We exclude SWGS in the comparison due to the space-inefficiency, as it ran out of memory to construct the range tree on 10^9 elements. For $k \le 3 \times 10^5$, our algorithm is consistently faster than Seq-BS (up to 9.1×). When the rank is large, the work in each round is not sufficient to get good parallelism, and the algorithm behaves as if it runs sequentially. Because of work-efficiency, even with large ranks, our parallel algorithm introduces limited overheads, and its performance is comparable to Seq-BS (at most 3.4× slower).

We also evaluate the self-relative speedup of our algorithm on input size $n = 10^9$ with rank 10^2 and rank 10^4 using both line and range generators. In all settings from Fig. 8, our algorithm scales well to 192 hyperthreads, reaching the self-speedup of up to 25.6× for $k = 10^2$ and up to 37.0× for $k = 10^4$. With the same rank, our algorithm has almost identical speedup for both patterns in all scales. Our algorithm outperforms Seq-BS (denoted as dash lines in Fig. 8) when using 8 or 16 cores, and is always better afterwards.

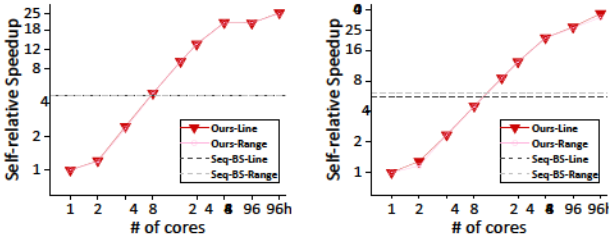
Overall, our LIS algorithm performs well with reasonable ranks, achieving up to 41× self-speedup with $n = 10^8$ and up to 70× self-speedup with $n = 10^9$. Due to work-efficiency, our algorithm is scalable and performs especially well on large data because larger input sizes result in more work to utilize parallelism better.

Weighted LIS. We compare our WLIS algorithm (Alg. 2) with SWGS and Seq-AVL on input size $n = 10^8$. We vary the rank from 1 to 3000, and show the results in Fig. 7(d). Our algorithm is always faster than SWGS (up to 2.5×). Our improvement comes from better work bound (a factor of $O(\log n)$ better, although in many cases SWGS's work bound is not tight). Our algorithm also outperforms the sequential algorithm Seq-AVL with ranks up to 100. The running time of the sequential algorithm decreases with increasing ranks k because of the better locality. In contrast, our algorithm performs worse with increasing k because of the larger span.



(a). LIS. $n = 10^8$. Line pattern. (b). LIS. $n = 10^9$. Line pattern. (c). LIS. $n = 10^9$. Range pattern. (d). Weighted LIS. $n = 10^8$. Line Pattern.

Figure 7: Experimental results on the LIS and WLIS. We vary the output size for each test. "Ours"= our LIS algorithm in Alg. 1 using 96 cores. "Ours (seq)"= our LIS algorithm in Alg. 1 using one core. "Ours-W"=our WLIS algorithm in Alg. 2 using 96 cores. "Seq-BS"= the sequential Seq-BS algorithm based on binary search. "Seq-AVL"= the sequential Seq-AVL algorithm based on the AVL tree. "SWGS"= the parallel algorithm SWGS from [64].



(a). LIS. $k = 10^2$. (b). LIS. $k = 10^4$.

Figure 8: Experimental results of Self-relative Speedup. "Ours-Line"= our LIS algorithm in Alg. 1 using a line pattern generator. "Ours-Range"= our LIS algorithm in Alg. 1 using a range pattern generator. "Seq-BS-Line"= Seq-BS algorithm using a line pattern generator. "Seq-BS-Range"= Seq-BS algorithm using a range pattern generator. The data generators are described at the beginning of Sec. 6.

The results also imply the importance of work-efficiency in practice. To get better performance, we believe an interesting direction is to design a work-efficient parallel algorithm for WLIS.

7 RELATED WORK

LIS is widely studied both sequentially and in parallel. Sequentially, various algorithms have been proposed [9, 28, 35, 50, 62, 78]. and the classic solution uses $O(n \log n)$ work. This is also the lower bound [35] w.r.t. the number of comparisons. In the parallel setting, LIS is studied both as general dynamic programming [16, 25, 37, 69] or on its own [4, 52, 57, 58, 63, 70, 71]. However, we are unaware of any work-efficient LIS algorithm with non-trivial parallelism ($o(n)$ or $\tilde{O}(k)$ span). Most existing parallel LIS algorithms introduced a polynomial overhead in work [37, 52, 57, 58, 63, 70], and/or have $\tilde{\Theta}(n)$ span [4, 16, 25, 69] (many of them [16, 25, 69] focused on improving the I/O bounds). The algorithm in [53] translates to $O(n \log^2 n)$ work and $\tilde{O}(n^{2/3})$ span, but it relies on complicated techniques for Monge Matrices [71]. Most of the parallel LIS algorithms are complicated and have no implementations. We are unaware of any parallel LIS *implementation* with competitive performance to the sequential $O(n \log k)$ or $O(n \log n)$ algorithm.

Many previous papers propose general frameworks to study dependencies in sequential iterative algorithms to achieve parallelism [13, 14, 18, 64]. Their common idea is to (implicitly or explicitly) traverse the DG. There are two major approaches, and both have led to many efficient algorithms. The first one is edge-centric [14, 15, 17–19, 34, 45, 49, 64], which identifies the ready objects by processing the successors of the newly-finished objects. The second approach is vertex-centric [13, 61, 64, 65, 72], which checks all unfinished objects in each round to process the ready ones. However, none of these frameworks directly enables work-efficiency for parallel LIS. The edge-centric algorithms evaluate all edges in the dependence graph, giving $\Theta(n^2)$ worst-case work for LIS. The vertex-centric algorithms check the readiness of all remaining objects in each round and require k rounds, meaning $\Omega(nk)$ work for LIS. The SWGS algorithm [64] combines the ideas in edge-centric and vertex-centric algorithms. SWGS has $O(n \log^3 n)$ work *whp* and is round-efficient ($\tilde{O}(k)$ span) using $O(n \log n)$ space. It is sub-optimal in work and space. Our algorithm improves the work and space bounds of SWGS in both LIS and WLIS. Our algorithm is also simpler and performs much better than SWGS in practice.

The vEB tree was proposed by van Emde Boas in 1977 [74], and has been widely used in sequential algorithms, such as dynamic programming [23, 33, 36, 46, 47, 59], computational geometry [1, 24, 26, 66], data layout [7, 44, 73, 75], and others [2, 38, 51, 55, 56]. However, to the best of our knowledge, there was no prior work on supporting parallelism on vEB trees.

8 CONCLUSION

In this paper, we present the first work-efficient parallel algorithm for the longest-increasing subsequence (LIS) problem that has non-trivial parallelism ($\tilde{O}(k)$ span for an input sequence with LIS length k). Theoretical efficiency also enables a practical implementation with good performance. We also present algorithms for parallel vEB trees and show how to use them to improve the bounds for the weight LIS problem. As a widely-used data structure, we believe our parallel vEB tree is of independent interest, and we plan to explore other applications as future work. Other interesting future directions include achieving work-efficiency and good performance for WLIS in parallel and designing a work-efficient parallel LIS algorithm with $o(n)$ or even a polylogarithmic span.

ACKNOWLEDGEMENT

This work is supported by NSF grants CCF-2103483, IIS-2227669, NSF CAREER award CCF-2238358, and UCR Regents Faculty Fellowships. We thank Huacheng Yu for some initial discussions on this topic, especially for the vEB tree part. We thank anonymous reviewers for the useful feedbacks.

REFERENCES

[1] Peyman Afshani and Zhewei Wei. 2017. Independent range sampling, revisited. In *esa*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[2] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. 2011. Best position algorithms for efficient top-k query processing. *Information Systems* 36, 6 (2011), 973–989.

[3] Yaroslav Akhremtsev and Peter Sanders. 2016. Fast Parallel Operations on Search Trees. In *IEEE International Conference on High Performance Computing (HiPC)*.

[4] Muhammad Rashed Alam and M Sohel Rahman. 2013. A divide and conquer approach and a work-optimal parallel algorithm for the LIS problem. *Inform. Process. Lett.* 113, 13 (2013), 470–476.

[5] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. 1990. Basic local alignment search tool. *Journal of molecular biology* 215, 3 (1990), 403–410.

[6] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems (TOCS)* 34, 2 (2001), 115–144.

[7] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. 2000. Cache-oblivious B-trees. In *focs*. IEEE, 399–409.

[8] Jon Louis Bentley and Jerome H Friedman. 1979. Data structures for range searching. *Comput. Surveys* 11, 4 (1979), 397–409.

[9] Sergei Bespamyatnikh and Michael Segal. 2000. Enumerating longest increasing subsequences and patience sorting. *Inform. Process. Lett.* 76, 1-2 (2000), 7–11.

[10] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Transactions on Parallel Computing (TOPC)* 9, 2 (2022), 1–41.

[11] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

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

[13] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[14] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[15] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[16] Guy E. Blelloch and Yan Gu. 2020. Improved Parallel Cache-Oblivious Algorithms for Dynamic Programming. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*.

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

[18] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Parallelism in Randomized Incremental Algorithms. *J. ACM* (2020).

[19] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Randomized Incremental Convex Hull is Highly Parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[20] Guy E. Blelloch and Margaret Reid-Miller. 1998. Fast Set Operations Using Treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[21] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. on Computing* 27, 1 (1998).

[22] Nairen Cao, Shang-En Huang, and Hsin-Hao Su. 2023. Nearly optimal parallel algorithms for longest increasing subsequence. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[23] Wun-Tat Chan, Yong Zhang, Stanley PY Fung, Deshi Ye, and Hong Zhu. 2007. Efficient algorithms for finding a longest common increasing subsequence. *Journal of Combinatorial Optimization* 13, 3 (2007), 277–288.

[24] Y-J Chiang and Roberto Tamassia. 1992. Dynamic algorithms in computational geometry. *Proc. IEEE* 80, 9 (1992), 1412–1434.

[25] Rezaul A. Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM.

[26] Francisco Claude, J Ian Munro, and Patrick K Nicholson. 2010. Range queries over untangled chains. In *International Symposium on String Processing and Information Retrieval*. Springer, 82–93.

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

[28] Maxime Crochemore and Ely Porat. 2010. Fast computation of a longest increasing subsequence and application. *Information and Computation* 208, 9 (2010), 1054–1059.

[29] Sanjoy Dasgupta, Christos H Papadimitriou, and Umesh Virkumar Vazirani. 2008. *Algorithms*. McGraw-Hill Higher Education New York.

[30] Percy Deift. 2000. Integrable systems and combinatorial theory. *Notices AMS* 47 (2000), 631–640.

[31] Arthur L Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L Salzberg. 1999. Alignment of whole genomes. *Nucleic acids research* 27, 11 (1999), 2369–2376.

[32] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[33] David Eppstein, Zvi Galil, and Raffaele Giancarlo. 1988. Speeding up dynamic programming. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 488–496.

[34] Manuela Fischer and Andreas Noever. 2018. Tight analysis of parallel randomized greedy MIS. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2152–2160.

[35] Michael L Fredman. 1975. On computing the length of longest increasing subsequences. *Discrete Mathematics* 11, 1 (1975), 29–35.

[36] Zvi Galil and Kunsoo Park. 1992. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science (TCS)* 92, 1 (1992).

[37] Zvi Galil and Kunsoo Park. 1994. Parallel algorithms for dynamic programming recurrences with more than O(1) dependency. *J. Parallel Distrib. Comput.* 21, 2 (1994).

[38] Paweł Gawrychowski, Shunsuke Inenaga, Dominik Köppl, Florin Manea, et al. 2015. Efficiently Finding All Maximal α-gapped Repeats. *arXiv preprint arXiv:1509.09237* (2015).

[39] Michael T Goodrich and Roberto Tamassia. 2015. *Algorithm design and applications*. Wiley Hoboken.

[40] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60.

[41] Yan Gu, Zachary Napier, Yihan Sun, and Letong Wang. 2022. Parallel Cover Trees and their Applications. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 259–272.

[42] Yan Gu, Zheqi Shen, Yihan Sun, and Zijin Wan. 2022. Work-Efcient Parallel Implementations on Longest Increasing Subsequence. https://github.com/ucrparlay/nLogn-LIS.

[43] Dan Gusfield. 1997. Algorithms on stings, trees, and sequences: Computer science and computational biology. *Acm Sigact News* 28, 4 (1997), 41–60.

[44] Hoai Phuong Ha, Ngoc Nha Vi Tran, Ibrahim Umar, Philippas Tsigas, Anders Gidenstam, Paul Renaud-Goud, Ivan Walulya, and Aras Atalar. 2014. Models for energy consumption of data structures and algorithms. (2014).

[45] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering heuristics for parallel graph coloring. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[46] James W Hunt and Thomas G Szymanski. 1977. A fast algorithm for computing longest common subsequences. *Commun. ACM* 20, 5 (1977), 350–353.

[47] Takafumi Inoue, Shunsuke Inenaga, Heikki Hyyrö, Hideo Bannai, and Masayuki Takeda. 2018. Computing longest common square subsequences. In *cpm*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl Publishing.

[48] Kurt Johansson. 1998. The longest increasing subsequence in a random permutation and a unitary random matrix model. *Mathematical Research Letters* 5, 1 (1998), 68–82.

[49] Mark T. Jones and Paul E. Plassmann. 1993. A parallel graph coloring heuristic. 14, 3 (1993), 654–669.

[50] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

[51] Arie MCA Koster, Hans L Bodlaender, and Stan PM Van Hoesel. 2001. Treewidth: computational experiments. *Electronic Notes in Discrete Mathematics* 8 (2001), 54–57.

[52] Peter Krusche and Alexander Tiskin. 2009. Parallel longest increasing subsequences in scalable time and memory. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 176–185.

[53] Peter Krusche and Alexander Tiskin. 2010. New algorithms for efficient parallel string comparison. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 209–216.

[54] Wei Quan Lim. 2019. Optimal Multithreaded Batch-Parallel 2-3 Trees. *arXiv preprint arXiv:1905.05254* (2019).

[55] Witold Lipski and Franco P Preparata. 1981. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica* 15, 4 (1981), 329–346.

[56] Witold Lipski Jr. 1983. Finding a Manhattan path and related problems. *Networks* 13, 3 (1983), 399–409.

[57] Takaaki Nakashima and Akihiro Fujiwara. 2002. Parallel algorithms for patience sorting and longest increasing subsequence. In *International Conference in Networks, Parallel and Distributed Processing and Applications*. 7–12.

[58] Takaaki Nakashima and Akihiro Fujiwara. 2006. A cost optimal parallel algorithm for patience sorting. *Parallel processing letters* 16, 01 (2006), 39–51.

[59] Shintaro Narisada, Kazuyuki Narisawa, Shunsuke Inenaga, Ayumi Shinohara, et al. 2017. Computing longest single-arm-gapped palindromes in a string. In *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 375–386.

[60] Ryan O'Donnell and John Wright. [n. d.]. A primer on the statistics of longest increasing subsequences and quantum states. *SIGACT News* ([n. d.]).
[61] Xinghao Pan, Dimitris Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael I. Jordan. 2015. Parallel correlation clustering on big graphs. In *Advances in Neural Information Processing Systems (NIPS)*. 82–90.
[62] Craige Schensted. 1961. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics* 13 (1961), 179–191.
[63] David Semé. 2006. A CGM algorithm solving the longest increasing subsequence problem. In *International Conference on Computational Science and Its Applications*. Springer, 10–21.
[64] Zheqi Shen, Zijin Wan, Yan Gu, and Yihan Sun. 2022. Many Sequential Iterative Algorithms Can Be Parallel and (Nearly) Work-efficient. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[65] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B Gibbons. 2015. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 431–448.
[66] Jack Snoeyink. 1992. Two-and Three-Dimensional Point Location in Rectangular Subdivisions. In *swat*, Vol. 621. Springer Verlag, 352.
[67] Yihan Sun and Guy E Blelloch. 2019. Parallel Range, Segment and Rectangle Queries with Augmented Maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*. 159–173.
[68] Yihan Sun, Daniel Ferizovic, and Guy E Blelloch. 2018. PAM: Parallel Augmented Maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.
[69] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul A Chowdhury. 2015. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.
[70] Garcia Thierry, Myoupo Jean-Frédéric, and Semé David. 2001. A work-optimal CGM algorithm for the LIS problem. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 330–331.
[71] Alexander Tiskin. 2015. Fast distance multiplication of unit-Monge matrices. *Algorithmica* 71, 4 (2015), 859–888.
[72] Daniel Tomkins, Timmie Smith, Nancy M Amato, and Lawrence Rauchwerger. 2014. SCCMulti: an improved parallel strongly connected components algorithm. *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)* 49, 8 (2014), 393–394.
[73] Ibrahim Umar, Otto Anshus, and Phuong Ha. 2013. Deltatree: A practical locality-aware concurrent search tree. *arXiv preprint arXiv:1312.2628* (2013).
[74] Peter van Emde Boas. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.* 6, 3 (1977), 80–82.
[75] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. 1976. Design and implementation of an efficient priority queue. *Mathematical systems theory* 10, 1 (1976), 99–127.
[76] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. 2021. A Parallel Batch-Dynamic Data Structure for the Closest Pair Problem. In *ACM Symposium on Computational Geometry (SoCG)*.
[77] Marvin Williams, Peter Sanders, and Roman Dementiev. 2021. Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues. In *European Symposium on Algorithms (ESA)*.
[78] I-Hsuan Yang, Chien-Pin Huang, and Kun-Mao Chao. 2005. A fast algorithm for computing a longest common increasing subsequence. *Inform. Process. Lett.* 93, 5 (2005), 249–253.
[79] Hongyu Zhang. 2003. Alignment of BLAST high-scoring segment pairs based on the longest increasing subsequence algorithm. *Bioinformatics* 19, 11 (2003), 1391–1396.
[80] Tingzhe Zhou, Maged Michael, and Michael Spear. 2019. A Practical, Scalable, Relaxed Priority Queue. In *International Conference on Parallel Processing (ICPP)*. 1–10.