Batch-Parallel Euler Tour Trees

Thomas Tseng* Laxman Dhulipala† Guy Blelloch‡

Abstract

The dynamic trees problem is to maintain a forest undergoing edge insertions and deletions while supporting queries for information such as connectivity. There are many existing data structures for this problem, but few of them are capable of exploiting parallelism in the batch setting, in which large batches of edges are inserted or deleted from the forest at once. In this paper, we demonstrate that the Euler tour tree, an existing sequential dynamic trees data structure, can be parallelized in the batch setting. For a batch of k updates over a forest of n vertices, our parallel Euler tour trees perform $O(k \log(1 + n/k))$ expected work with $O(\log n)$ depth with high probability. Our work bound is asymptotically optimal, and we improve on the depth bound achieved by Acar et al. for the batch-parallel dynamic trees problem [1].

Our main building block for parallelizing Euler tour trees is a batch-parallel skip list data structure, which we believe may be of independent interest. Euler tour trees require a sequence data structure capable of joins and splits. Traditionally, balanced binary trees are used, but they are difficult to join or split in parallel when processing batches of updates. We show that skip lists, on the other hand, support batches of joins or splits of size k over n elements with $O(k \log(1 + n/k))$ work in expectation and $O(\log n)$ depth with high probability. We also achieve the same efficiency bounds for augmented skip lists, which allows us to augment our Euler tour trees to support subtree queries.

Our data structures achieve between 67–96× self-relative speedup on 72 cores with hyper-threading on large batch sizes. Our data structures also significantly outperform the fastest existing sequential dynamic trees data structures empirically.

*Computer Science Department, Carnegie Mellon University. thomasts@alumni.cmu.edu

†Computer Science Department, Carnegie Mellon University. ldhulipa@cs.cmu.edu

‡Computer Science Department, Carnegie Mellon University. guyb@cs.cmu.edu

1 Introduction

In the dynamic trees problem proposed by Sleator and Tarjan [45], the objective is to maintain a forest that undergoes *link* and *cut* operations. A link operation adds an edge to the forest, and a cut operation deletes an edge. Additionally, we want to maintain useful information about the forest. Most commonly we are concerned with whether pairs of vertices are connected, but we might also be interested in properties like the size of each tree in the forest. Sleator and Tarjan first studied the dynamic trees problem in order to develop fast network flow algorithms [45]. Dynamic trees are also an important component of many dynamic graph algorithms [45, 21, 24, 4, 26].

In the batch-parallel version of the dynamic trees problem, the objective is to maintain a forest that undergoes *batches* of *link* and *cut* operations. Though many sequential data structures exist to maintain dynamic trees, to the best of our knowledge the only batch-parallel data structure is a very recent result by Acar et al. [1]. Their data structure is based on parallelizing RC-trees, which require transforming a forest into a forest with bounded-degree in order to perform contractions efficiently [2]. Obtaining a data structure without this restriction is therefore of interest. Furthermore, it is of intellectual interest whether the arguably simplest solution to the dynamic trees problem, Euler tour trees (ETTs), are capable of being parallelized when given batches of edge insertions and deletions.

In this paper, we answer this question in the affirmative and show that Euler tour trees, a data structure introduced by Henzinger and King [21] and Miltersen et al. [34], achieve asymptotically optimal work and optimal depth in the batch-parallel setting. We note that batching is not only useful for parallel applications but also for single-threaded applications. Our $O(k \log(1 + n/k))$ work bounds for Euler tour trees and augmented skip lists beat the $O(k \log n)$ bounds achieved by performing each operation one at a time on standard sequential data structures.

Our main contributions are as follows:

Skip lists for simple, efficient parallel joins and parallel splits. We show that we can perform k joins or k splits over n skip list elements with $O(k \log(1 + n/k))$

expected work and $O(\log n)$ depth with high probability[1]. To the best of our knowledge, we are the first to demonstrate such efficiency for batch joins and splits on a sequence data structure supporting fast search. Our skip list data structure can also be augmented to support efficient computation over contiguous subsequences within the same efficiency bounds.

A parallel Euler tour tree. We apply our skip lists to develop Euler tour trees that support parallel bulk updates. Our Euler tour tree algorithms for adding and for removing a batch of k edges achieve $O(k \log(1+n/k))$ expected work and $O(\log n)$ depth with high probability. These are the *best known bounds for the batch-parallel dynamic trees problem.*

Experimental evidence of good performance. Our skip list and Euler tour tree data structures achieve good self-relative speedups, ranging from 67× to 96× on 72 cores with hyper-threading in our experiments. We also show that they significantly outperform the fastest existing sequential alternatives.

2 Related Work

2.1 Sequences. A common data structure for representing sequences is the search tree. Concurrent binary search trees, however, tend to be hard to maintain because of the frequent tree rebalancing necessary to preserve fast access times. Kung and Lehman present a concurrent binary search tree supporting search, insertion, and deletion [28]. Their implementation grabs locks during rebalancing, which blocks searches from proceeding. Ellen et al. provide a lock-free binary search tree with the downside that the tree has no balance guarantees [14]. Braginsky and Petrank design a lock-free balanced tree in the form of a B+ tree [10].

Batch parallelism for search, insertions, and deletions has been studied in 2-3 trees [36], red-black trees [35], and B-trees [23]. All of these data structures achieve $O(k \log n)$ work and $O(\log n + \log k)$ depth.

Very recently, Akhremtsev and Sanders implement parallel joins and splits for (a, b)-trees as subroutines for efficient batch updates [3]. The work for batch joins is $O(k \log(1 + n/k))$, and the work for batch splits is $O(k \log n)$. The depth for both operations is $O(\log n)$. Compared to [3], our skip lists are simpler, allow augmentation, and improve on the work for batch splits. However, as (a, b)-trees are a deterministic data structure, Akhremtsev and Sanders obtain deterministic bounds whereas our bounds are randomized.

Skip lists. Skip lists are a randomized data structure introduced by Pugh for representing ordered sequences [38]. Concurrent skip lists may be used as the basis for dictionaries [47] and priority queues [40]. Skip lists are also used for storing database indices. For example, the popular database system MemSQL builds its indices upon skip lists [32]. To the best of our knowledge, no existing skip-list implementation supports batch-parallel bounds for performing batches of splits or joins.

Pugh [37], Herlihy et al. [22], and Fraser [16] describe concurrent skip lists supporting search, insertion, and deletion. They allow all operations to run concurrently and do not show theoretical bounds. Gabarró et al. present a skip list supporting batch searches, insertions, or deletions in $O(k(\log n + \log k))$ expected work and $O(\log n + \log k)$ expected depth [18].

2.2 Dynamic Trees. Many sequential data structures exist for the dynamic trees problem. Sleator and Tarjan introduced the problem and gave a sequential data structure known as the ST-tree or the link-cut tree for the problem [45]. ST-trees are difficult to parallelize because they rely on a complicated biased search tree data structure. Sleator and Tarjan later showed that ST-trees could be significantly simplified by using splay trees [46]. However, splay trees are not amenable to parallelization due to the major structural changes on every access caused by splaying nodes. Another data structure is Frederickson's topology tree, which works by hierarchically clustering the represented forest [17]. Acar et al.'s RC-trees similarly contracts the forest to obtain a clustering [2]. Unfortunately, both of these data structures require the forest to have bounded degree and thus require modifying the original graph by splitting high degree vertices into several bounded degree vertices. Top trees, devised by Alstrup et al., circumvent this restriction and allow for unbounded degree [4]. They also have the most general interface. The Euler tour tree, developed by Miltersen et al. [34] and Henzinger and King [21], is arguably the simplest data structure for solving the dynamic trees problem, but, unlike many other dynamic trees data structures, they do not support path queries.

Acar et al. very recently developed a batch-parallel solution to the dynamic trees problem [1]. They achieve the same work bound as our solution of $O(k \log(1+n/k))$ in expectation, but their depth bound is $O(C(k) \log n)$ where $C(k)$ is the depth of compacting k elements. As $C(k)$ is $\Omega(\log^* k)$ [30], our Euler tour trees achieve better depth. Their data structures also require transforming the input forest into a bounded-degree forest in order to use parallel tree-contraction efficiently [33].

[1] We say that an algorithm has $O(f(n))$ cost ***with high probability (w.h.p.)*** if it has $O(k \cdot f(n))$ cost with probability at least $1 - 1/n^k$.

2.3 Parallel Dynamic Connectivity. The dynamic trees problem with connectivity queries is the dynamic connectivity problem restricted to acyclic graphs. A nearly ubiquitous strategy for dynamic connectivity is to maintain a spanning tree of the graph as it undergoes modifications. The difficulty of dynamic connectivity comes from discovering a replacement edge going across a cut after deleting an edge in the spanning tree that is maintained. Stipulating that the represented graph is acyclic (as we do in this paper) simplifies the problem because it guarantees that edge removal breaks a connected component into two.

Though there is much work on sequential dynamic connectivity [17, 21, 49, 24, 26], parallel dynamic connectivity is not well explored. McColl et al. provide a parallel algorithm for batch dynamic connectivity including edge deletions, but their goal is to achieve fast experimental results on real-world graphs rather than to achieve provable efficiency bounds [31]. We note that the worst-case work and depth of their algorithm is the same as a BFS on the graph. Simsiri et al. give a work-efficient, logarithmic-depth algorithm for batch incremental (no deletions) dynamic connectivity [44]. Kopelowitz et al. have recently shown that the sparsified version of Frederickson's algorithm [17, 15] can be parallelized nearly work-efficiently for a single update [27]. However, they do not consider parallelizing the algorithm when processing batches of edge updates.

3 Preliminaries

In this paper we analyze our algorithms on the MT-RAM, a simple work-depth model which is closely related to the PRAM but more closely models current machines and programming paradigms that are asynchronous and support dynamic forking. We define the model in Appendix A and refer the interested reader to [9] for more details. Our efficiency bounds are stated in terms of work and depth, where ***work*** is the total number of vertices in the thread DAG and where ***depth*** (***span***) is the length of the longest path in the DAG [8].

We design algorithms using *nested fork-join parallelism* in which a procedure can *fork* off another procedure call to run in parallel and then wait for forked calls to complete with a *join* synchronization [8]. In our implementations, we use Cilk Plus [29] for fork-join parallelism. We borrow its use of *spawn* to mean "fork" and *sync* to mean "join" to disambiguate from the other sense in which we use "join" in this work (that is, as an operation that concatenates two sequences).

Our algorithms only require the compare-and-swap atomic primitive, which is widely available on modern multicores. A COMPARE-AND-SWAP$(\&x, o, n)$ (CAS) instruction takes a memory location x and atomically updates the value at location x to n if the value is currently o, returning *true* if it succeeds and *false* otherwise.

Parallel Primitives. The following parallel procedures are used throughout the paper. A ***semisort*** takes an input array of elements, where each element has an associated key and reorders the elements so that elements with equal keys are contiguous, but elements with different keys are not necessarily ordered. The purpose is to collect equal keys together, rather than sort them. Semisorting a sequence of length n can be performed in $O(n)$ expected work and $O(\log n)$ depth with high probability assuming access to a uniformly random hash function mapping keys to integers in the range $\left[1, n^{O(1)}\right]$ [39, 20].

A ***parallel dictionary*** data structure supports batch insertion, batch deletion, and batch lookups of elements from some universe with hashing. Gil et al. describe a parallel dictionary that uses linear space and achieves $O(k)$ work and $O(\log^* k)$ depth with high probability for a batch of k operations [19].

The ***list tail-finding*** problem is to assign each node in a linked list a pointer to the last node in a linked list (note that there are also other variants referred to as list ranking in the literature in which we wish to compute the distance to the last node). There are many solutions for this problem that have $O(n)$ work and $O(\log n)$ depth [11, 5, 12, 6].

The ***pack*** operation takes an n-length sequence A and an n-length sequence B of booleans as input. The output is a sequence A' of all the elements $a \in A$ such that the the corresponding entry in B is *true*. The elements of A' appear in the same order that they appear in A. Packing can be easily implemented in $O(n)$ work and $O(\log n)$ depth [25].

4 Sequences and Parallel Skip Lists

We start by first specifying a high-level interface for batch-parallel sequences. We then describe our batch-parallel skip lists which implement the interface, and finally, end by discussing how our data structure can be extended to support augmentation.

4.1 Batch-Parallel Sequence Interface The goal of a batch-parallel sequence data structure is to represent a collection of sequences under batches of parallel operations that split and join sequences. To *join* two sequences is to concatenate them together. To *split* a sequence A at element x is to separate the sequence into two subsequences, the first of which consists of all elements in A before and including x, the second of which consists of all elements after x.

Sequences. We now give a formal description of the interface for sequences. The data structure supports the following functions:

- **BatchJoin**($\{(x_1, y_1), \ldots, (x_k, y_k)\}$ takes an array of tuples where the i-th tuple is a pointer x_i, to the last element of one sequence, and a pointer y_i, to the first element of a second sequence and concatenates the first sequence with the second sequence. For any two distinct tuples in the input with values (x_i, y_i) and (x_j, y_j), we must have $x_i \neq x_j$ and $y_i \neq y_j$.
- **BatchSplit**($\{x_1, \ldots, x_k\}$) takes an array of pointers to elements, and for each pointer x_i, breaks the sequence immediately after x_i.
- **BatchFindRep**($\{x_1, \ldots, x_k\}$) takes an array of pointers to elements. It returns an array where the i-th entry is the *representative* of the sequence in which x_i lives. The representative is defined so that *representative*(u) = *representative*(v) if and only if u and v live in the same sequence. Note that representatives are invalidated after the sequences are modified.

BatchFindRep is used as a subroutine for connectivity queries on the Euler tour trees in Section 5.

Augmented Sequences. To augment a sequence, we take an associative function $f : D^2 \to D$ where D is an arbitrary domain of values. A value from D is assigned to each element in the sequence, A. An augmented sequence data structure supports querying for the value of f over contiguous subsequences of A. Specifically, our interface for augmented sequences extends the interface for unaugmented sequences with the following function:

- **BatchUpdateValue**($\{(x_1, a_1), \ldots, (x_k, a_k\}$) takes an array of tuples, where the i-th tuple contains a pointer to an element x_i and a new value for the element, a_i. The value for x_i is set to a_i in the sequence.
- **BatchQueryValue**($\{(x_1, y_1), \ldots, (x_k, y_k)\}$) takes an array of k tuples, where the i-th tuple contains pointers to elements x_i and y_i. The return value is an array where the i-th entry holds the value of f applied over the subsequence between x_i and y_i. For $1 \leq i \leq k$, x_i and y_i must be pointers to elements in the same sequence.

4.2 Skip Lists. Skip lists are a simple randomized data structure that can be used to represent sequences [38]. To represent a sequence, skip lists assign a *height* to each element of the sequence, where each height is drawn independently from a geometric distribution. The ℓ-th level of a skip lists consists of a linked list over the subsequence formed by all elements of height at least ℓ. This structure allows efficient search. Figure 1 shows an example skip list.

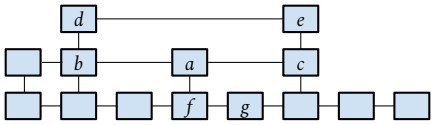

Figure 1: An example skip list over a sequence of eight elements. On the bottom are all the level-1 nodes.

For an element x of height h, we allocate a node v_i for every level $i = 1, 2, \ldots, h$. Each node has four pointers LEFT, RIGHT, UP, and DOWN. We set $v_i \to \text{UP} = v_{i+1}$ and $v_i \to \text{DOWN} = v_{i-1}$ for each i to connect between levels. We set $v_i \to$ RIGHT to the i-th node of the next element of height at least i and similarly $v_i \to$ LEFT to the i-th node of the previous element of height at least i.

Our skip lists support *cyclicity*, which is to say that our algorithms are valid even if we link the tail and head of a skip list together. Though this is not conventionally done with sequence data structures, we will find it useful for representing Euler tours of graphs in Section 5 since Euler tours are naturally cyclic sequences. We cannot join upon cyclic sequences, but splitting a cyclic sequence at element x corresponds to unraveling it into a linear sequence with its last element being x. Figure 2 illustrates joining and splitting on our skip lists.

Definitions. We now introduce definitions that describe the relationship between nodes. Say we have a node v that represents element x at some level i. We call $v \to$ RIGHT v's *successor*. Similarly, $v \to$ LEFT is its *predecessor*. We call $v \to$ UP its *direct parent* and $v \to$ DOWN its *direct child*. For example, in Figure 1, consider node a. Its predecessor is b, its successor is c, its direct child is f, and it has no direct parent.

The *left parent* is the level-$(i+1)$ node of the latest element preceding and including x that has height at least $i + 1$. The *right parent* is defined symmetrically. Under this definition, if v has a direct parent, then its left and right parents are both its direct parent. When we refer to v's parent, we refer to its left parent. In Figure 1, a's (left) parent is d, and a's right parent is e. The *(left) ancestors* consist of v's parent, v's parent's parent, and so on, and similarly for v's *right ancestors*. Thus the ancestors for both f and g in Figure 1 are a and d. A *child* is inverse to a parent, and a *descendant* is inverse to an ancestor.

The following definitions describe the relationship between the links connecting nodes. The *parent* of a link between v and its successor is the link between v's

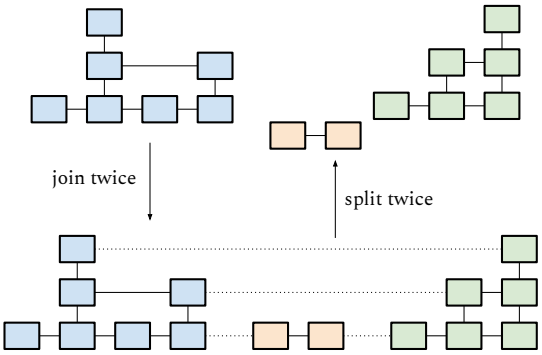

Figure 2: Joins and splits on skip lists.

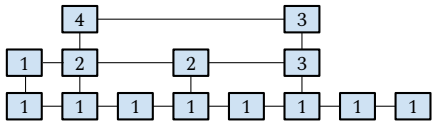

Figure 3: Each node in this skip list is augmented with a size value, summing all the values of its children.

parent and its successor. Similarly, the *ancestors* of the link are links between v's ancestors and their successors. The *children* of the link are the links between v's children and their successors.

Joins, Splits and Augmentation on Skip Lists. Recall that in an augmented sequence, we take an associative function $f : D^2 \to D$ for some domain D. Each element in the sequence A is assigned some value from D. By storing these values in the bottom level of our skip list and partial "sums" at higher levels, we can compute f over contiguous subsequences of A in logarithmic time. For instance, in Figure 3, we assign the value 1 to every element and choose $f : \mathbb{N}^2 \to \mathbb{N}$ to be the sum function. For each node v, we store the sum of the values of v's children. By looking at $O(\log n)$ nodes, we can then compute the size of the sequence. These augmented values are also easy to maintain as the skip list undergoes joins and splits.

Our skip lists support batch joins, batch splits, batch point updates of augmented values, and batch finding representatives in $O(k \log(1 + n/k))$ work in expectation and $O(\log n)$ depth with high probability, where k is the batch size and n is the number of elements in the lists. We analyze efficiency in Appendix C.

This improves on the $\Theta(k \log n)$ expected work bound achieved by conventional sequential joins and splits on augmented skip lists. Intuitively, the reason we can achieve improved work-bounds is that if a node has many updated descendants, our algorithm updates its augmented value once rather than multiple times.

Algorithm 1 Searches for the left parent of the input node. The mirror function SEARCHRIGHT is defined symmetrically.

```
1: procedure SEARCHLEFT(v)
2:     current = v
3:     while current → UP = null do
4:         current = current → LEFT
5:         if current = null or current = v then
6:             return null
7:     return current → UP
```

Algorithm 2 Joins two lists together given their endpoints.

```
1: procedure JOIN(v_L, v_R)
2:     if CAS(&v_L → RIGHT, null, v_R) then
3:         v_R → LEFT = v_L
4:         parent_L = SEARCHLEFT(v_L)
5:         parent_R = SEARCHRIGHT(v_R)
6:         if parent_L ≠ null and parent_R ≠ null then
7:             JOIN(parent_L, parent_R)
```

4.3 Algorithms for unaugmented lists. We begin by describing unaugmented skip lists. For creating elements in our skip list, we fix a probability $0 < p < 1$ representing the expected proportion of nodes at a particular level that have a direct parent at the next level. We generate heights of elements by allocating a node and giving each node a direct parent with probability p independently. This is equivalent to drawing heights from a Geometric$(1 - p)$ distribution.

We give pseudocode for JOIN and SPLIT over unaugmented lists in Algorithms 2 and 3 respectively. To perform a batch of joins, we simply call JOIN on each join operation in the batch concurrently, and similarly for a batch of splits. As each batch of splits and joins must be run in separate phases, our data structure is *phase-concurrent* over joins and splits [41].

Both algorithms employ two simple helper procedures, SEARCHLEFT and SEARCHRIGHT, for finding the left and right parents of a node. We show SEARCHLEFT in Algorithm 1, and SEARCHRIGHT is implemented symmetrically. Note that these procedures avoid looping forever on cyclic skip lists.

Join. Recall that the definition of JOIN takes a pointer to the last element of one list and a pointer to the first element of a second list and concatenates the first list with the second list. Starting at the bottom level, our algorithm links the given nodes, searches upwards to find parents to link at the next level, and repeats. We set the link with a CAS, and if the CAS is lost, the algorithm quits. This permits only one thread to set a particular link, preventing repeated work.

THEOREM 4.1. *Let B be a set of valid* JOIN *inputs. Then calling* JOIN *concurrently over the inputs in B gives the same result as joining over the inputs in B sequentially.*

Proof. (Proof sketch) We argue inductively level-by-level that all necessary links are added and no unneces-

Algorithm 3 Separates the input node from its successor.

```
1: procedure Split(v)
2:     ngh = v → right
3:     if ngh ≠ null and CAS(&v → right, ngh, null) then
4:         ngh → left = null
5:         parent = SearchLeft(v)
6:         if parent ≠ null then
7:             Split(parent)
```

sary links are added. For the base case, at the bottom level, the links we add are exactly those given as input to the algorithm, which are the necessary links to add at that level. For the inductive step, assume that the correct links will be added on level i. Consider any link ℓ from nodes v_L to v_R on level $i+1$ that should be added. In order for this to be a link we need to add, there must be a rightward path from v_L's direct child to v_R's direct child once all links on level i are added. Then consider the last execution of JOIN on level i to add a link on that path by finishing line 3 of Algorithm 2. That execution will have a complete path to find parents v_L and v_R when searching and thus will find ℓ as a link to add. Conversely, any level-$(i+1)$ link from nodes v_L to v_R found by a join execution was found via a complete path (albeit perhaps temporarily missing some LEFT pointers due to some executions of join completing line 2 but not yet completing line 3) between v_L's direct child and v_R's direct child, which indicates that this link should be added. We present a formal proof using this idea in the full version of this paper [50]. □

Split. SPLIT takes a pointer to an element and breaks the list right after that element. Similar to join, it cuts the link at the bottom level and then loops in searching upwards to find a parent link to remove at the next levels. Like JOIN, this uses CAS to avoid duplicate work.

THEOREM 4.2. *Let B be a set of elements. Then calling* SPLIT *concurrently over the elements in B gives the same result as splitting over the elements in B sequentially.*

Proof. (Proof sketch) Like in the proof sketch of Theorem 4.1, we look at the links that are removed inductively level-by-level. The argument is similar, except that in the inductive step, to see that a link ℓ on level $i+1$ from nodes v_L to v_R that should be removed will indeed be removed by the phase of splits, we note that the leftmost split on the path from $v_L \to$ DOWN to $v_R \to$ DOWN will be able to find parent v_L in its SEARCHLEFT call. We present a formal proof using this idea in the full version of this paper [50]. □

Finding representative nodes. A simple phase-concurrent implementation of FINDREP that takes $O(k \log n)$ expected work for k concurrent calls is to start at the input node and walk to the top level of the list. Then on the top level, for an acyclic list, we return the leftmost node, or for a cyclic list, we return the lowest address node.

Algorithm 4 Helper function for BATCHUPDATEVALUES that updates the augmented value for v and all its descendants.

```
1: procedure UpdateTopDown(v)
2:     v → needs_update = false
3:     if v → down = null then                    ▷ Reached bottom level
4:         return
5:     current = v → down
6:     do
7:         if current → needs_update then
8:             spawn UpdateTopDown(current)
9:         current = current → right
10:    while current ≠ null and current → up = null
11:    sync
12:    sum = v → down → val
13:    current = v → down → right
14:    while current ≠ null and current → up = null do
15:        sum = f(sum, current → val)
16:        current = current → right
17:    v → val = sum
```

However, if we are given a batch of k calls up front, we can in fact achieve $O(k \log(1 + n/k))$ expected work and $O(\log n)$ depth with high probability. The idea is that each call of FINDREP takes some path up the skip list to the top level, and calls whose paths intersect somewhere can be combined at that point to avoid duplicate work. Then the return value gets propagated back down to both original calls. The code would look similar to the code for batch updating augmented values in Algorithm 5 for augmented skip lists (Subsection 4.4). We omit the full details.

4.4 Algorithm for augmented lists. We now describe how to augment our skip lists. In addition to its four pointers, each node is given a value VAL from some domain D and a boolean NEEDS_UPDATE. We provide an associative function $f : D^2 \to D$ and, for each element in the list, a value from D. We assign values to VAL on nodes at the bottom level, and then compute VAL at higher levels by applying f over nodes' children. The boolean is used to mark nodes whose values need updating and is initialized to *false*.

We give the main algorithm BATCHUPDATEVALUES for batch point update in Algorithm 5. This takes a set of nodes at the bottom level along with values to give to the associated elements. For each node in the set, we start by updating its value (line 5). Then each of its ancestors have values that need updating, so we walk up its ancestors, CASing on each ancestor's NEEDS_UPDATE variable (line 6). If an execution loses a CAS, then it may quit because some other execution will take care of all the node's ancestors.

Now over all the input nodes that won all CASes on their ancestors, we know the union of their topmost

Algorithm 5 Takes a batch of (node, value) pairs, updates each node with its associated value, and updates the augmented values stored throughout the list.

```
1:  procedure BatchUpdateValues({(v_1, a_1), ..., (v_k, a_k)})
2:      top = {null, null, ..., null}                    ▷ k-length array
3:      for i ∈ {1, ..., k} do in parallel
4:          v_i → val = a_i
5:          current = v_i
6:          while CAS(&current → needs_update, false, true) do
7:              parent = SearchLeft(current)
8:              if parent = null then
9:                  top[i] = current
10:                 break
11:             current = parent
12:     for i ∈ {1, ..., k} do in parallel
13:         if top[i] ≠ null then
14:             UpdateTopDown(top[i])
```

Algorithm 6 Batch join for augmented skip lists.

```
1:  procedure BatchJoin({(l_1, r_1), (l_2, r_2), ..., (l_k, r_k)})
2:      for i ∈ {1, ..., k} do in parallel
3:          Join(l_i, r_i)
4:      BatchUpdateValues({(l_1, l_1 → val), ..., (l_k, l_k → val)})
```

ancestors' descendants contain all the input nodes. By calling the helper function UpdateTopDown (Algorithm 4) on every such topmost ancestor in lines 12–14, we traverse back down and update these descendants' augmented values. Given a node, this helper function calls itself recursively on all the node's children c who need an update as indicated by $c \rightarrow$ needs_update (lines 5–10). Then, after all the childrens' values are updated, we may update the original node's value (lines 11–17).

With this algorithm for batch point update, batch joins (Algorithm 6) and batch splits (algorithm omitted due to similarity to Algorithm 6) are simple. We first perform all the joins or splits. Then we batch update on the nodes we joined or split on. We keep all the values on the bottom level the same, but the update fixes all the values on the higher levels that are changed by adding or removing links.

4.5 Implementation.

We provide details about our skip list implementations in Appendix B.

5 Batch-Parallel Euler Tour Trees

In this section we present batch-parallel Euler tour trees, a solution to the batch-parallel dynamic trees problem. In order to ease exposition, we first present a batch-parallel interface for the dynamic trees problem.

Batch-Parallel Dynamic Trees Interface. A solution to the batch-parallel dynamic trees problem supports representing a forest as it undergoes batches of links, cuts, and connectivity queries. *Links* link two trees in the forest. *Cuts* delete an edge from the forest and break one tree into two trees. *Connected* queries take two vertices in the forest and return whether they

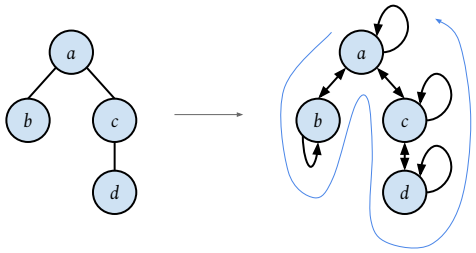

Figure 4: We take the tree on the left and transform it so that we get the following Euler tour of edges: (a, b) (b, b) (b, a) (a, c) (c, d) (d, d) (d, c) (c, c) (c, a) (a, a).

are connected (in the same tree). We now give a formal description of the interface. The data structure maintains a graph $G = (E, V)$, which is assumed to be a forest under the following operations:

- **BatchLink**$(\{(u_1, v_1), \ldots, (u_k, v_k)\})$ takes an array of edges and adds them to the graph G. The input edges must not create a cycle in G.
- **BatchCut**$(\{(u_1, v_1), \ldots, (u_k, v_k)\})$ takes an array of edges and removes them from the graph G.
- **BatchConnected**$(\{(u_1, v_1), \ldots, (u_k, v_k)\})$ takes an array of tuples representing queries. The output is an array where the i-th entry returns whether vertices u_i and v_i are connected by a path in G.

We also support augmenting the trees with an associative and commutative function $f : D^2 \rightarrow D$ with values from D assigned to vertices and edges of the forest. The goal is to compute f over subtrees of the represented forest. Note that we assume that the function is commutative in order to allow implementations to relax the order in which a vertex's children appear in the tour. The interface supports batch updates over vertices and edges. The primitives are similar to the batch updates of values for augmented skip lists, so we elide the details. The interface for subtree queries is different, and we present it below:

- **BatchSubtree**$(\{(u_1, p_1), \ldots, (u_k, p_k)\})$ takes an array of tuples, where the i-th tuple contains a vertex u_i and its parent p_i in the tree. It returns an array where the i'th entry contains the value of f summed over u_i's subtree relative to its parent p_i in G. Note that because the Euler tour tree is unrooted, we require providing the parent p_i in order to determine the intended subtree for u_i.

Euler Tour Trees. We focus on a variant of ETTs presented by Tarjan [48]. To represent a tree as an Euler tour tree, replace each edge $\{u, v\}$ with two directed edges (u, v) and (v, u) and add a loop (v, v) to each

vertex v, as in Figure 4. Note that this construction produces a connected graph in which each vertex has equal indegree and outdegree, and therefore the graph admits an Euler tour. We represent the tree as any of its Euler tours.

Now linking two trees corresponds to splicing their Euler tours together, and cutting a tree corresponds to cutting out part of its Euler tour. Each of these operations reduces to a few joins and splits on the tours. We may also answer whether two vertices u and v are connected by asking whether their loops, (u, u) and (v, v), reside in the same tour. Moreover, because a subtree appears as a contiguous section of an Euler tour, we can efficiently compute information about subtrees if we can efficiently compute information about contiguous sections of tours.

Traditionally, Euler tour trees store an Euler tour by breaking it into a sequence at an arbitrary location and then placing the sequence in a balanced binary tree. We instead store Euler tours as cycles using our skip lists from Section 4. Because skip lists are easy to join and split in parallel, we can process batches of links and cuts on Euler tour trees efficiently.

We show that for a batch of k joins, k splits, or k connectivity queries over an n-vertex forest, we can achieve $O(k \log(1 + n/k))$ expected work and $O(\log n)$ depth with high probability. If we build our Euler tour trees over augmented skip lists, we can also answer subtree queries in logarithmic time.

5.1 Description. Our Euler tour trees crucially rely on our parallel skip lists to represent Euler tours. Since a graph of n vertices has Euler tours whose lengths sum to $O(n)$, the skip lists hold $O(n)$ nodes. Thus a batch of k joins or splits on the Euler tours takes $O(k \log(1 + n/k))$ expected work and $O(\log n)$ depth with high probability.

Construction. For clarity, we describe our ETTs using the phase-concurrent unaugmented skip lists given in Section 4. However, it is easy to organize the joins and splits into batches so as to match the augmented skip list interface seen in Subsection 4.4. We also treat our dictionary data structure as phase-concurrent for clarity, but again, this is easy to circumvent.

We add fields TWIN and MARK to each skip list element. For a node representing a directed edge (u, v), TWIN is a pointer to the node representing the directed edge (v, u) in the opposite direction. We initialize the field MARK to *false* and use it during splitting to mark nodes that will be removed.

At initialization (Algorithm 7), the represented graph is an n-vertex forest with no edges, and we assume the vertices are labeled with integers $1, 2, \ldots, n$. We create an n-length array *verts* such that *verts*$[i]$ stores a pointer to the skip list node representing the loop edge (i, i). As such, in parallel, for $i = 1, \ldots, n$, we create a skip list node, assign it to *verts*$[i]$, and join it to itself to form a singleton cycle. These cycles are the Euler tours in an empty graph. We also keep a dictionary *edges* that maps edges (u, v) with $u \neq v$ to corresponding skip list nodes. Lastly, we create an array *successors* that will be used as scratch space for batch linking.

Algorithm 7 Euler tour tree data structure initialization.

```
1: procedure INITIALIZE(n)
2:     verts = {}                                   ▷ n-length array
3:     for i ∈ {1, ..., n} do in parallel
4:         verts[i] = CREATENODE()
5:         JOIN(verts[i], verts[i])
6:     edges = DICT()                               ▷ empty dictionary
7:     successors = {}                              ▷ n-length array
```

Algorithm 8 Add a batch of edges to Euler tour tree.

```
1: procedure BATCHLINK({{u_1, v_1}, {u_2, v_2}, ..., {u_k, v_k}})
2:         ▷ Adding input edges must not create a cycle in graph
3:                          ▷ Create nodes representing new edges
4:     for i ∈ {1, ..., k} do in parallel
5:         uv = CREATENODE()
6:         vu = CREATENODE()
7:         uv → TWIN = vu
8:         vu → TWIN = uv
9:         edges[(u_i, v_i)] = uv
10:        edges[(v_i, u_i)] = vu
11:            ▷ Cut at locations at which we splice in other tours
12:    for i ∈ {1, ..., k} do in parallel
13:        for w ∈ {u_i, v_i} do
14:            w_node = verts[w]
15:            w_succ = w_node → RIGHT
16:            if w_succ ≠ null then
17:                ▷ benign race; this assignment and split are idem-
                     potent
18:                successors[w] = w_succ
19:                SPLIT(w_node)
20:    sorted_edges =
           SEMISORT({(u_1, v_1), (v_1, u_1), ..., (u_k, v_k), (v_k, u_k)})
21:            ▷ Join together tours with new edge nodes in between
22:    for i ∈ {1, ..., 2k} do in parallel
23:        (u, v) = sorted_edges[i]
24:        (u_prev, v_prev) = sorted_edges[i − 1]
25:        (u_next, v_next) = sorted_edges[i + 1]
26:        if i = 1 or u ≠ u_prev then
27:            JOIN((verts[u], edges[(u, v)]))
28:        if i = 2k or u ≠ u_next then
29:            JOIN((edges[(v, u)], successors[u]))
30:        else
31:            JOIN((edges[(v, u)], edges[(u_next, v_next)]))
```

Connectivity queries. To check whether two vertices are connected, we simply check whether they live in the same Euler tour by comparing the representatives of their tours' skip lists. The complexity of this can be made $O(k \log(1 + n/k))$ expected work and $O(\log n)$ depth with high probability using an efficient batch FINDREP algorithm.

Batch Link. Algorithm 8 shows our algorithm for adding a batch of edges. The algorithm takes an array of edges A to add as input. We assume that adding the input edges preserves acyclity.

Algorithm 9 Computes locations at which to join for batch split.

```
1:  procedure GetNextUnmarked(elements = {z_1, z_2, ..., z_k})
2:                                  ▷ Input is a set of skip list elements
3:      for i ∈ {1, ..., k} do in parallel
4:          next = z_i → TWIN → RIGHT
5:          if next → MARK then
6:              z_i → NEXT_EDGE = next
7:          else
8:              z_i → NEXT_EDGE = null
9:                  ▷ Use list tail-finding on the linked lists induced by
                    NEXT_EDGE pointers. Get an array last_marked
                    such that last_marked[i] points to the last node
                    in z_i's linked list.
10:     last_marked = ListRank(elements)
11:     result = {}                                  ▷ k-length array
12:     for i ∈ {1, ..., k} do in parallel
13:         result[i] = last_marked[i] → TWIN → RIGHT
14:     return result
```

To add a single edge $\{u, v\}$ sequentially, we can find locations where u and v appear in their tours by looking up $verts[u]$ and $verts[v]$. We split on those locations and join the resulting cut up tours back together with new nodes representing (u, v) and (v, u) in between. If we want to add several edges at once, we need to be careful when inserting edges that are incident to the same vertex and thus attempt to join on the same location.

With that in mind, we proceed to describe our algorithm. In lines 3-10, for each input edge $\{u, v\}$, we allocate new list elements representing directed edges (u, v) and (v, u). Then, in lines 11-19, for each vertex u that appears in the input, we split u's list at $verts[u]$ as a location to splice in other tours. We also save the successor of $verts[u]$ in $successors[u]$ so that we can join everything back together at the end.

For each vertex u, say that the input tells us that we want to newly connect u to vertices $w_1, w_2, \ldots, w_k$. Then we join together the nodes representing (u, u) to (u, w_1), (w_i, u) to (u, w_{i+1}) for $1 \le i < k$, and (w_k, u) to what was the successor to (u, u) before splitting. In our code, we arrange this in lines 20-31 by semisorting the input to collect together all edges incident on a vertex. The ordering of $w_1, w_2, \ldots, w_k$ is unimportant, only corresponding to the order in which they appear after u in the Euler tour.

Using our skip lists and an efficient semisort [20], we see that the work is $O(k \log(1 + n/k))$ in expectation, and the depth is $O(\log n)$ with high probability.

Batch Cut. Algorithm 10 describes how to remove a batch of edges. Our algorithm assumes that each edge exists in the forest and that there are no duplicates.

Cutting a single edge is simple. If we cut an edge $\{u, v\}$, we split before and after (u, v) and (v, u) in the tour and join their neighbors together appropriately. However, as with batch linking, the task gets more difficult if we want to cut many edges out of a single node, because those neighbors that we want to join together may themselves be split off.

Algorithm 10 Remove a batch of edges from Euler tour tree.

```
1:  procedure BatchCut({{u_1, v_1}, {u_2, v_2}, ..., {u_k, v_k}})
2:                  ▷ Input edges must be in graph and must have no
    duplicates.
3:      directed_edges = {}                          ▷ 2k-length array
4:      for i ∈ {1, ..., k} do in parallel
5:          directed_edges[2i − 1] = edges[(u_i, v_i)]
6:          directed_edges[2i] = edges[(v_i, u_i)]
7:      for i ∈ {1, ..., k} do in parallel
8:          edges → RemoveFromDict((u_i, v_i))
9:          edges → RemoveFromDict((v_i, u_i))
10:     join_lefts = {}                              ▷ 2k-length array
11:     for i ∈ {1, ..., 2k} do in parallel
12:         join_lefts[i] = directed_edges[i] → LEFT
13:         directed_edges[i] → MARK = true
14:     join_rights = GetNextUnmarked(directed_edges)
15:                                          ▷ Cut edges out of tour
16:     for i ∈ {1, ..., 2k} do in parallel
17:         Split(directed_edges[i])
18:         pred = directed_edges[i] → Left
19:         if pred ≠ null then
20:             Split(pred)
21:                                  ▷ Join tours back together
22:     for i ∈ {1, ..., 2k} do in parallel
23:         if not join_lefts[i] → MARK then
24:             Join(join_lefts[i], join_rights[i])
25:     for i ∈ {1, ..., 2k} do in parallel
26:         DeleteNode(directed_edges[i])
```

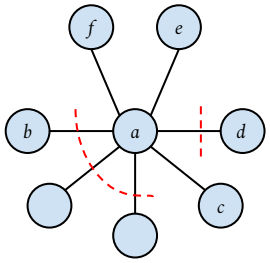

Figure 5: Batch cutting four edges. If we take an Euler tour counter-clockwise around this graph, this batch cuts may require us to join (f, a) to (a, c) in the tour.

As an example, consider the graph in Figure 5 in which we remove four edges. If our tour on the graph goes counter-clockwise around the diagram, then we may need to join (c, a) to (a, e) in the tour as a result of cutting $\{a, d\}$ and join (f, a) all the way around to (a, c) as a result of the three contiguous cuts. How do we identify that we need to join (f, a) to (a, c)? We could mark edges that are going to be cut, then start from (f, a) and walk along "adjacent" edges incident to a using the TWIN pointers until we reach an edge that will not be cut. Then we would know to join (f, a) to that edge. However, the search for an unmarked edge will have poor depth if lots of edges will be cut.

To achieve low depth in this step, we use list tail-finding. Consider the linked lists induced by having each edge point at its adjacent edge if it is marked. Note that each linked list must terminate because traversing adjacent edges will eventually reach a loop edge of the

form (v, v), which will certainly be unmarked. Then running list tail-finding on these linked lists finds for every edge the next unmarked edge as desired.

In Algorithm 10, we first fetch all the skip list nodes corresponding to the edges in lines 2-6. Then we invoke Algorithm 9 on line 14, which performs the list tail-finding described above. We cut out all the input edges on lines 15-20 and rejoin all the tours together on lines 21-24. In total these steps take $O(k \log(1 + n/k))$ expected work and $O(\log n)$ depth with high probability.

Augmentation. We build our augmented Euler tour trees over the concurrent augmented skip lists from Subsection 4.4 and achieve the same efficiency bounds. Recall that we have an associative and commutative function $f : D^2 \to D$ and assign values from D to vertices and edges of the forest. The goal is to compute f over subtrees of the represented forest.

Say we want to compute f over a vertex v's subtree relative to v's parent in the tree, p. Then if we look up the skip list elements corresponding to (p, v) and (v, p) in *edges*, the value of f over v's subtree is the result of applying f on the subsequence between (p, v) and (v, p). This may be done by calling BatchQueryValue with the elements corresponding to (p, v) and (v, p) in the underlying augmented skip list.

5.2 Implementation.

We provide details about our implementation of Euler tour trees in Appendix B.

6 Experiments

We run our experiments on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Our programs use Cilk Plus to express parallelism and are compiled with the g++ compiler (version 5.4.1) with the `-O3` flag. When running in parallel, we use the command `numactl -i all` to evenly distribute the allocated memory among the processors. On our figures, a thread count of 72(h) denotes using all 72 cores with hyper-threading, i.e. 144 threads.

6.1 Unaugmented Skip Lists.

We evaluate the performance of our skip lists (with the probability of a node having a direct parent set to $p = 1/2$) by comparing them against other sequence data structures. In particular, we compare against sequential skip lists, which are the same as our skip lists except that they do not use CAS to set pointers. In addition, for an element of height h, they allocate an array of exactly length h for holding pointers rather than an array of length $O(h)$ as our skip list does (see Appendix B). We also implemented splay trees [46] and treaps [7].

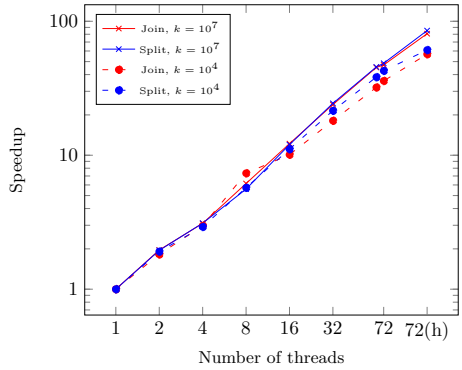

Figure 6: Speedup of our concurrent skip lists with $n = 10^8$.

So that we can compare against another parallel data structure, we implement parallel batch join and batch split operations on treaps. To batch join, we first ignore a constant fraction of the joins. If we imagine each join from treap T to treap S as a pointer from T to S, we get lists on the treaps. No list can be very long because of the ignored joins. We get parallelism by processing each list independently. If we store extra information on the treap nodes, we can walk along a list and perform its joins sequentially. Then we recursively process the previously ignored joins. For batch split, we semisort the splits keyed on the root of the treap to be split. This lets us find all splits that act on a particular treap. We process each treap independently. When performing multiple splits on a treap, we get parallelism by divide and conquer—we perform a random split and recursively split the resulting two treaps in parallel. The randomized efficiency bounds are $O(k \log n)$ work for batch join, $O(k \log n \log k)$ work for batch split, and $O(\log n \log k)$ depth for both. In the future, we would like to further compare our skip lists against other parallel data structures, such as the (a, b)-trees of Akhremtsev and Sanders [3].

For an experiment, we take $n = 10^8$ elements and fix a batch size k. We set up a trial by joining all the elements in a chain, and then we time how long it takes to split and rejoin the sequence at k pseudorandomly sampled locations. We report the median time over three trials. As an artifact of this set up, the splay tree has an advantage on joining small batches after splitting due to how splay trees exploit locality.

Figure 6 illustrates that our skip list implementation running on 72 cores with hyper-threading demonstrates over 80× speedup relative to the implementation running on a single thread for $k = 10^7$ and over 55× speedup for $k = 10^4$. We compare our skip list to our other sequence data structures in Figure 7. To conserve space, we only show the plot for splitting. Our imple-

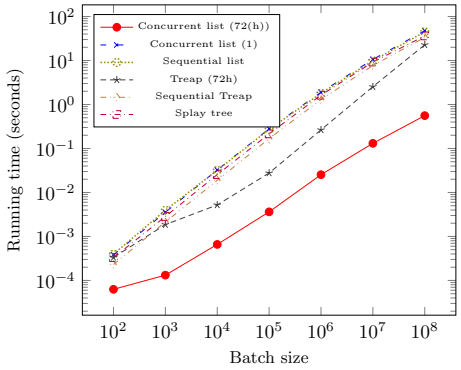

Figure 7: Running time of splitting sequence data structures with varying batch size.

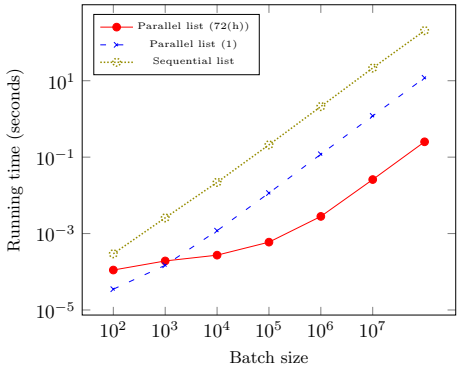

Figure 8: Running time of splitting augmented skip lists with $n = 10^8$ as batch size varies with splits taking off single elements off the end of the list.

mentation of parallel batch join on treaps is 1.4× faster than our batch join on skip lists on the largest batch sizes, but, as seen in Figure 7, the parallel batch split on treaps is much slower due to lots of overhead work. Moreover, through parallelism, our data structure is significantly faster than all the sequential algorithms at all batch sizes. When used sequentially, our data structure behaves similarly to a traditional sequential skip list, suggesting that using CAS does not significantly degrade the performance of a skip list.

6.2 Augmented Skip Lists. We compare the performance of our batch-parallel augmented skip lists against a sequential augmented skip list. Besides not using CAS, the sequential skip list updates augmented values after every join and split. This achieves only an $O(k \log n)$ work bound for k operations. Our experiment is the same as in Subsection 6.1.

When running our augmented skip list with a random batch of size $k = 10^7$ on 72 cores with hyper-threading, we see a speedup of 67× for joins and 78× for splits. For $k = 10^4$, we found a speedup of 33× for joins and 48× for splits. The running times are a factor of two worse than the times for the unaugmented skip list of Subsection 6.1, which is expected due to the extra passes through the skip list to update augmented values. The batch-parallel skip list hugely outperforms single-threaded skip lists on all tested batch sizes. We omit plots here due their visual similarity to Figures 6 and 7.

To more prominently display the work savings that batching provides, we try different test case where a batch of k splits, rather than being chosen at random, consists of splitting at the last k elements of the sequence in right-to-left order. This is particularly bad for the sequential skip list because after every split, it walks to the top of the skip list to update augmented values. In comparison, when processing the splits as a batch, we update the augmented values in only one pass. Figure 8 shows that, as expected, even on a single thread, our skip list is significantly faster than the standard sequential one in this adversarial experiment.

6.3 Euler Tour Trees. We compare against sequential dynamic trees data structures. Using the sequential skip list and splay trees from Subsection 6.1, we build traditional Euler tour trees. We also compare to ST-trees built on splay trees [45, 46]. They achieve $O(\log n)$ amortized work links and cuts. Though conceptually more complicated than Euler tour trees, ST-trees are a more streamlined data structure that do not require allocation beyond initialization and do not require dictionary lookups. We wrote all of these implementations. In future work, we would like to compare against parallel data structures such as that of Acar et al. [1]

Because one of the important uses of Euler tour trees is to answer connectivity queries, we also compare with statically computing the connected components of the graph after a batch of updates. We use the work-efficient parallel connectivity algorithm designed and implemented by Shun et al. [43]. We optimistically measure the execution time of the implementation based only on the execution time of the connectivity algorithm. We do not include the time taken to maintain the graph itself, which is non-trivial because the adjacency array graph representation used in their implementation does not support edge updates easily.

For our experiment, we fix a tree. We set up a trial by adding all the edges of the tree in pseudorandom order to our data stucture. Then we time how long it takes to cut and relink the forest at k pseudorandomly sampled edges. We report median times over three trials. Again, note that our experimental setup may give the splay tree data structures an advantage on linking small batches after cutting due to how splay trees

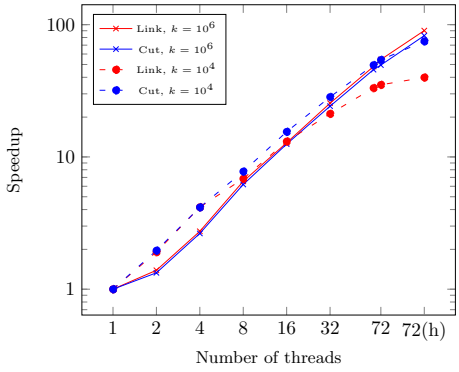

Figure 9: Speedup of our parallel Euler tour trees on a random recursive tree of size $n = 10^7$.

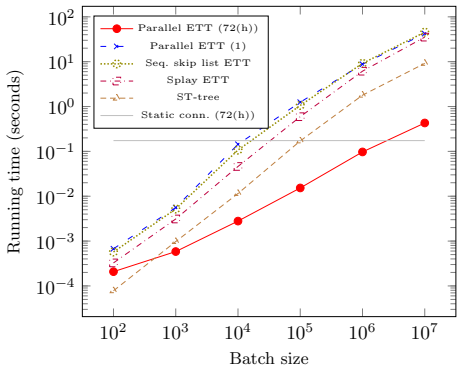

Figure 10: Running time of linking dynamic trees data structures on random recursive tree of size $n = 10^7$ with varying batch size. The plot for splitting is visually similar.

exploit locality. We experimented on three trees, all with $n = 10^7$ vertices: a path graph, a star graph, and a random recursive tree. To form a random recursive tree over n vertices, for each $1 < i \leq n$, draw j uniformly at random from $\{1, 2, \ldots, i-1\}$ and add the edge $\{j, i\}$. Due to space constraints we only plot results for the random recursive tree here. We include all of our experimental results in the full version of the paper [50].

Figure 9 displays the speedup of our parallel Euler tour tree algorithms with a batch sizes of $k = 10^4$ and $k = 10^6$ on a random recursive tree. When running on 72 cores with hyper-threading, we get good speedup ranging from 82× to 96× for $k = 10^6$ across all tested graphs. For $k = 10^4$, we found speedup ranging from 7.5× to 70× where the worst speedup was on the star graph. In Figure 10, we show the running times of our Euler tour tree along with the times for sequential dynamic trees data structures and static parallel connectivity on a random recursive tree. On large batch sizes, parallelism beats all the sequential data structures, as expected. Though ST-trees are faster than Euler tour trees sequentially and are unusually fast on the star graph due to running well on graphs with small diameter, our parallel Euler tour tree eventually outspeeds ST-trees on large batches even on the star graph. In addition, the performance of our Euler tour tree running on a single thread is similar to that of conventional sequential Euler tour trees. We also see in Figure 10 that the time to update our Euler tour tree is much less than the time to statically compute connectivity for all but the largest batches.

7 Discussion

We showed that skip lists are a simple, fast data structure for parallel joining and splitting of sequences and that we can use these skip lists to build a batch-parallel Euler tour tree. Both of these data structures have good theoretical efficiency bounds and achieve good performance in practice. We hope that our work not only brings greater understanding to the field of parallel data structures but also serves as a stepping stone towards efficient parallel algorithms for dynamic graph problems. In particular, Holm et al. give a dynamic connectivity algorithm achieving $O\left(\log^2 n\right)$ amortized work per update in which augmented Euler tour trees play a crucial role [24]. We believe that applying our parallel Euler tour trees along with several other techniques can efficiently parallelize Holm et al.'s algorithm. Obtaining a batch-parallel solution for the general dynamic connectivity problem is an interesting research problem that has not been adequately addressed in the literature.

Acknowledgements

Thanks to Umut Acar and Sam Westrick for helpful conversations. This work was supported in part by NSF grants CCF-1408940, CCF-1533858, and CCF-1629444.

References

[1] U. A. Acar, V. Aksenov, and S. Westrick, *Brief announcement: Parallel dynamic tree contraction via self-adjusting computation*, in SPAA, 2017.

[2] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo, *Dynamizing static algorithms, with applications to dynamic trees and history independence*, in SODA, 2004.

[3] Y. Akhremtsev and P. Sanders, *Fast parallel operations on search trees*, in HiPC, 2016.

[4] S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup, *Maintaining information in fully dynamic trees with top trees*, TALG, 1 (2005).

[5] R. J. Anderson and G. L. Miller, *Deterministic parallel list ranking*, in Aegean Workshop on Computing, 1988.

[6] R. J. Anderson and G. L. Miller, *A simple randomized parallel algorithm for list-ranking*, IPL, 33 (1990).

[7] C. R. Aragon and R. G. Seidel, *Randomized search trees*, in FOCS, 1989.

[8] G. E. Blelloch, *Programming parallel algorithms*, CACM, 39 (1996).

[9] G. E. Blelloch and L. Dhulipala, *Introduction to parallel algorithms 15-853: Algorithms in the real world*. 2018.

[10] A. Braginsky and E. Petrank, *A lock-free b+ tree*, in SPAA, 2012.

[11] R. Cole and U. Vishkin, *Approximate parallel scheduling. part i: The basic technique with applications to optimal parallel list ranking in logarithmic time*, SICOMP, 17 (1988).

[12] R. Cole and U. Vishkin, *Faster optimal parallel prefix sums and list ranking*, Information and computation, 81 (1989).

[13] E. Demaine and S. Goldwasser, *6.046J/18.410J lecture notes on skip lists*. March 2004, `https://courses.csail.mit.edu/6.046/spring04/handouts/skiplists.pdf`.

[14] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, *Non-blocking binary search trees*, in PODC, 2010.

[15] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, *Sparsificationa technique for speeding up dynamic graph algorithms*, JACM, 44 (1997).

[16] K. Fraser, *Practical lock-freedom*, tech. report, University of Cambridge, Computer Laboratory, 2004.

[17] G. N. Frederickson, *Data structures for on-line updating of minimum spanning trees, with applications*, SICOMP, 14 (1985).

[18] J. Gabarró, C. Martínez, and X. Messeguer, *A design of a parallel dictionary using skip lists*, TCS, 158 (1996).

[19] J. Gil, Y. Matias, and U. Vishkin, *Towards a theory of nearly constant time parallel algorithms*, in FOCS, 1991.

[20] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch, *A top-down parallel semisort*, in SPAA, 2015.

[21] M. R. Henzinger and V. King, *Randomized dynamic graph algorithms with polylogarithmic time per operation*, in STOC, 1995.

[22] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, *A provably correct scalable concurrent skip list*, in OPODIS, 2006.

[23] L. Higham and E. Schenk, *Maintaining B-trees on an EREW PRAM*, JPDC, 22 (1994).

[24] J. Holm, K. De Lichtenberg, and M. Thorup, *Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity*, JACM, 48 (2001).

[25] J. Jaja, *Introduction to Parallel Algorithms*, Addison-Wesley Professional, 1992.

[26] B. M. Kapron, V. King, and B. Mountjoy, *Dynamic graph connectivity in polylogarithmic worst case time*, in SODA, 2013.

[27] T. Kopelowitz, E. Porat, and Y. Rosenmutter, *Improved worst-case deterministic parallel dynamic minimum spanning forest*, in SPAA, 2018.

[28] H. Kung and P. L. Lehman, *Concurrent manipulation of binary search trees*, TODS, 5 (1980).

[29] C. E. Leiserson, *The Cilk++ concurrency platform*, in Proceedings of the 46th Annual Design Automation Conference, 2009, pp. 522–527.

[30] P. D. MacKenzie, *Load balancing requires $\Omega(\log^* n)$ expected time*, in SODA, 1992.

[31] R. McColl, O. Green, and D. A. Bader, *A new parallel algorithm for connected components in dynamic graphs*, in HiPC, 2013.

[32] MemSQL, *MemSQL Docs: Indexes*, `https://docs.memsql.com/concepts/v6.5/indexes/`.

[33] G. Miller and J. Reif, *Parallel tree contraction and its application*, in FOCS, 1985.

[34] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia, *Complexity models for incremental computation*, TCS, 130 (1994).

[35] H. Park and K. Park, *Parallel algorithms for red–black trees*, TCS, 262 (2001).

[36] W. Paul, U. Vishkin, and H. Wagener, *Parallel dictionaries on 2–3 trees*, in ICALP, 1983.

[37] W. Pugh, *Concurrent maintenance of skip lists*, University of Maryland, Inst. for Advanced Computer Studies, 1990.

[38] W. Pugh, *Skip lists: a probabilistic alternative to balanced trees*, CACM, 33 (1990).

[39] J. H. Reif and S. Sen, *Parallel computational geometry: An approach using randomization*, in Handbook of Computational Geometry, 1999, ch. 18.

[40] N. Shavit and I. Lotan, *Skiplist-based concurrent priority queues*, in IPDPS, 2000.

[41] J. Shun and G. E. Blelloch, *Phase-concurrent hash tables for determinism*, in SPAA, 2014.

[42] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, *Brief announcement: the problem based benchmark suite*, in SPAA, 2012.

[43] J. Shun, L. Dhulipala, and G. E. Blelloch, *A simple and practical linear-work parallel algorithm for connectivity*, in SPAA, 2014.

[44] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K. Wu, *Work-efficient parallel union-find with applications to incremental graph connectivity*, in EuroPar, 2016.

[45] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, JCSS, 26 (1983).

[46] D. D. Sleator and R. E. Tarjan, *Self-adjusting binary search trees*, JACM, 32 (1985).

[47] H. Sundell and P. Tsigas, *Scalable and lock-free concurrent dictionaries*, in SAC, 2004.

[48] R. E. Tarjan, *Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm*, Mathematical Programming, 78 (1997).

[49] M. Thorup, *Near-optimal fully-dynamic graph connec-*

tivity, in STOC, 2000.

[50] T. Tseng, L. Dhulipala, and G. Blelloch, *Batch-parallel euler tour trees*, CoRR, abs/1810.10738 (2018).

A Model

The *Multi-Threaded Random-Access Machine (MT-RAM)* consists of a set of threads that share an unbounded memory. Each thread is essentially a Random Access Machine—it runs a program stored in memory, has a constant number of registers, and uses standard RAM instructions (including an `end` to finish the computation). The only difference between the MT-RAM and a RAM is the `fork` instruction that takes a positive integer k and forks k new child threads. Each child thread receives a unique identifier in the range $[1, \ldots, k]$ in its first register and otherwise has the same state as the parent, which has a 0 in that register. All children start by running the next instruction. When a thread performs a `fork`, it is suspended until all the children terminate (execute an `end` instruction). A computation starts with a single root thread and finishes when that root thread terminates. This model supports what is often referred to as nested parallelism. Note that if root thread never forks, it is a standard sequential program.

We note that we can simulate the CRCW PRAM equipped with the same operations with an additional $O(\log^* n)$ factor in the depth due to load-balancing. Furthermore, a PRAM algorithm using P processors and T time can be simulated in our model with PT work and T depth. We equip the model with a compare-and-swap operation (see Section 3) in this paper.

Lastly, we define the cost-bounds for this model. A computation can be viewed as a series-parallel DAG in which each instruction is a vertex, sequential instructions are composed in series, and the forked subthreads are composed in parallel. The ***work*** of a computation is the number of vertices and the ***depth*** (***span***) is the length of the longest path in the DAG. We refer the interested reader to [9] for more details about the model.

B Algorithm Implementation

Skip Lists. In our implementation of our skip lists, instead of representing an element of height h as h distinct nodes, we instead allocate an array holding h left and right pointers. This avoids jumping around in memory when traversing up direct parents. In fact, we allocate an array whose size is h rounded up to the next power of two. This decreases the number of distinct-sized arrays, which makes memory allocation easier when performing concurrent allocation. We also cap the height at 32, again for easier allocation. We set the probability of a node having a direct parent to be $p = 1/2$.

We also need to be careful about read-write reordering on architectures with relaxed memory consistency. For Join (Algorithm 2), if the reads from the searches in lines 4 to 5 are reordered before the write on line 3, then we can fail find a parent link that should be added. Thus we disallow reads from being reordered before line 3.

For augmented skip lists, instead of keeping h needs_update booleans for each element, we keep a single integer that saves the lowest level on which the element needs an update. This works because if a node needs updating, then all its direct ancestors need updating as well.

Another optimization saves a constant factor in the work for batch split. In BatchUpdateValues, we first walk up the skip list claiming nodes and then walk back down to update all the augmented values. We perform these two passes because in order to update a node's value, we need to know all of its childrens' values are already updated too, which is easier to coordinate when walking top-down through the list. However, in a batch split, after cutting up the list, no nodes on the bottom level share any ancestors. As a result, we can update all augmented values in a single pass walking up the list without even using CAS.

Euler Tour Trees. We implemented our parallel Euler tour tree algorithms, making several adjustments for performance and for ease of implementation. For simplicity, we use the unaugmented skip lists and do not support subtree queries.

To achieve good parallelism, we need to allocate and deallocate skip list nodes in parallel. We use lock-free concurrent fixed-size allocators that rely on both global and local pools. To reduce the number of fixed-size allocators used, we constrain the skip list heights and arrays as described previously.

For the dictionary *edges*, we use the deterministic hash table dictionary from the Problem Based Benchmark Suite (PBBS) [42]. This hash table is based upon a phase-concurrent hash table developed by Shun and Blelloch [41]. As an additional storage optimization, for an edge $\{u, v\}$ where $u < v$, we only store (u, v) in our dictionary and use the twin pointer to look up (v, u).

Instead of performing a semisort when batch joining, we found it faster to use the parallel radix sort from PBBS.

For batch cut, we do not use list tail-finding because efficient list tail-finding is challenging to implement. Instead, we opt for a recursive batch cut algorithm. Recall why we used list tail-finding in Algorithm 10: we do not want to spend too much time walking around adjacent edges to find one that is unmarked. In our recursive batch cut algorithm, we resolve the issue by randomly selecting a constant fraction of the edges from

the input to ignore and not cut. Then if we walk around adjacent edges naively, the number of edges we need to walk around until we see an unmarked edge is constant in expectation and $O(\log n)$ with high probability. Thus we can cut out the unignored edges quickly. Then we recurse on the ignored edges. This increases the depth by a factor of $O(\log k)$ but does not asymptotically affect the work.

C Skip list efficiency

Recall that in our skip lists, each node independently has a direct parent with probability $0 < p < 1$.

C.1 Work. We prove a work bound of $O(k \log(1 + n/k))$ in expectation over a set of k splits over n elements for unaugmented skip lists. The same strategy proves the bound for joins and for operations on augmented skip lists.

For a set of k splits, we first show that $O(k \log(1 + n/k))$ links need to be cut in expectation. Over the first $h = \left\lceil \log_{1/p}(1 + n/k) \right\rceil$ levels, each split needs to remove at most h links (one at each level), so there are $O(kh)$ pointers to remove over the first h levels. For each level $k > h$, the number of links to remove is bounded by the number of nodes on the level. The probability that a particular node has height at least ℓ is $p^{\ell-1}$, so the expected number of nodes reaching level ℓ is $np^{\ell-1}$. Then the number of links summed across all levels $\ell > h$ is at most

$$n \sum_{\ell=h+1}^{\infty} p^{\ell-1} = np^h \frac{1}{1-p} \leq np^{\log_{1/p}(1+n/k)} \frac{1}{1-p}$$
$$= \frac{n}{(1-p)(1+n/k)} \leq \frac{n}{(1-p)(n/k)} = O(k).$$

Therefore, the expected number of links we need to cut in total is $O(kh) + O(k) = O(k \log(1 + n/k))$.

For each link to remove, the amount of work to find that link from the previous child link in a split is $O(1)$ in expectation. To search for a link at level $i + 1$ that needs removal, we call SearchLeft, which walks left from the previous place we removed a link on level i until we see a direct parent. The amount of work is proportional to the number of nodes we touch when walking left. The probability a node has a direct parent is p independently, so the number of nodes we must touch until we see a direct parent is distributed according to Geometric(p) with expected value $1/(1-p) = O(1)$. (If we quit early due to reaching the beginning of a list or due to detecting a cycle, that only reduces the amount of work we do.) Thus this traversal work to find parent links only affects the expected work by a constant factor.

Moreover, no two split operations can both remove the same link because we remove links with a CAS. Whoever CASes the link first successfully clears the link, and whoever comes afterwards quits. The quitting execution only does $O(1)$ extra work from the extra traversal to find the already claimed link. Thus there is no significant duplicate work per split.

Therefore, the work overall for k splits is $O(k \log(1+ n/k))$.

C.2 Depth. For analyzing depth, we know that with high probability, every search path (a path from the top level of a skip list to a particular node on the bottom level, or the reverse) in an n-element skip list has length $O(\log n)$. A proof is given in [13]. The main critical paths of our operations consist of traversing search paths and doing up to a constant amount of extra work at each step, so we get a depth bound of $O(\log n)$ with high probability for any of our operations.