Parallel Minimum Cuts in $O(m \log^2 n)$ Work and Low Depth

Daniel Anderson
Carnegie Mellon University
Pittsburgh, PA, USA
dlanders@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA, USA
guyb@cs.cmu.edu

ABSTRACT

We present a randomized $O(m \log^2 n)$ work, $O(\text{polylog}\, n)$ depth parallel algorithm for minimum cut. This algorithm matches the work bounds of a recent sequential algorithm by Gawrychowski, Mozes, and Weimann [ICALP'20], and improves on the previously best parallel algorithm by Geissmann and Gianinazzi [SPAA'18], which performs $O(m \log^4 n)$ work in $O(\text{polylog}\, n)$ depth.

Our algorithm makes use of three components that might be of independent interest. Firstly, we design a parallel data structure that efficiently supports batched mixed queries and updates on trees. It generalizes and improves the work bounds of a previous data structure of Geissmann and Gianinazzi and is work efficient with respect to the best sequential algorithm. Secondly, we design a parallel algorithm for approximate minimum cut that improves on previous results by Karger and Motwani. We use this algorithm to give a work-efficient procedure to produce a tree packing, as in Karger's sequential algorithm for minimum cuts. Lastly, we design an efficient parallel algorithm for solving the minimum 2-respecting cut problem.

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**; **Parallel algorithms**.

KEYWORDS

minimum cut; parallel algorithms; graph algorithms; dynamic trees

ACM Reference Format:
Daniel Anderson and Guy E. Blelloch. 2021. Parallel Minimum Cuts in $O(m \log^2 n)$ Work and Low Depth. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3409964.3461797

This work is licensed under a Creative Commons Attribution International 4.0 License.

SPAA '21, July 6–8, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8070-6/21/07.
https://doi.org/10.1145/3409964.3461797

1 INTRODUCTION

Minimum cut is a classic problem in graph theory and algorithms. The problem is to find, given an undirected weighted graph $G = (V, E)$, a nonempty subset of vertices $S \subset V$ such that the total weight of the edges crossing from S to $V \setminus S$ is minimized. Early approaches to the problem were based on reductions to maximum s-t flows [16, 17]. Several algorithms followed which were based on edge contraction [21, 25, 30, 31]. Karger was the first to observe that tree packings [32] can be used to find minimum cuts [23]. In particular, for a graph with n vertices and m edges, Karger showed how to use random sampling and a tree packing algorithm of Gabow [11] to generate a set of $O(\log n)$ spanning trees such that, with high probability, the minimum cut crosses at most two edges of one of them. A cut that crosses at most k edges of a given tree is called a *k-respecting* cut. Karger then gives an $O(m \log^2 n)$-time algorithm for finding minimum 2-respecting cuts, yielding a randomized $O(m \log^3 n)$-time algorithm for minimum cut. Karger also gives a parallel algorithm for minimum 2-respecting cuts in $O(n^2)$ work and $O(\log^3 n)$ depth.

Until very recently, these were the state-of-the-art sequential and parallel algorithms for the weighted minimum cut problem. A new wave of interest in the problem has recently pushed these frontiers. Geissmann and Gianinazzi [14] design a parallel algorithm for minimum 2-respecting cuts that performs $O(m \log^3 n)$ work in $O(\log^2 n)$ depth. Their algorithm is based on parallelizing Karger's algorithm by replacing a sequential data structure for the so-called *minimum path* problem, based on dynamic trees, with a data structure that can evaluate a *batch* of updates and queries in parallel. Their algorithm performs just a factor of $O(\log n)$ more work than Karger's sequential algorithm, but substantially improves on the work of Karger's parallel algorithm.

Soon after, a breakthrough from Gawrychowski, Mozes, and Weimann [12] gave a randomized $O(m \log^2 n)$ algorithm for minimum cut. Their algorithm achieves the $O(\log n)$ speedup by designing an $O(m \log n)$ algorithm for finding the minimum 2-respecting cuts, which was the bottleneck of Karger's algorithm. This is the first result to beat Karger's seminal algorithm in over 20 years.

An open question posed by Karger was whether a deterministic algorithm can achieve an $O\left(m^{1+o(1)}\right)$ runtime. This was recently resolved in the affirmative by Li [26] by derandomizing the construction of the spanning trees.

In our work, we combine ideas from Gawrychowski et al. and Geissmann and Gianinazzi with several new techniques to close the gap between the parallel and sequential algorithms. Our contribution can be summarized by:

Theorem 1.1. *The minimum cut of a weighted graph can be computed with high probability in $O(m \log^2 n)$ work and $O(\log^3 n)$ depth.*

We achieve this using a combination of results that may be of independent interest. Firstly, we design a framework for evaluating mixed batches of updates and queries on trees work efficiently in low depth. This algorithm is based on parallel Rake-Compress Trees (RC trees) [1]. Roughly, we say that a set of update and query operations implemented on an RC tree is *simple* (defined formally in Section 3) if the updates maintain values at the leaves that are modified by an associative operation and combined at the internal nodes, and the queries read only the nodes on a root-to-leaf path and their children. Simple operation sets include updates and queries on path and subtree weights.

Theorem 1.2. *Given a bounded-degree RC tree of size n and a simple operation set, after $O(n)$ work and $O(\log n)$ depth preprocessing, batches of k operations from the operation-set, can be processed in $O(k \log(kn))$ work and $O(\log n \log k)$ depth. The total space required is $O(n + k_{max})$, where k_{max} is the maximum size of a batch.*

This result generalizes and improves on Geissmann and Gianinazzi [14] who give an algorithm for evaluating a batch of k path-weight updates and queries in $\Omega(k \log^2 n)$ work.

Next, we design a faster parallel algorithm for approximating minimum cuts, which is used as an ingredient in producing the tree packing used in Karger's approach (Section 4). To achieve this, we design a faster sampling scheme for producing graph skeletons, leveraging recent results on sampling binomial random variables, and a transformation that reduces the maximum edge weight of the graph to $O(m \log n)$ while approximately preserving cuts.

Lastly, we show how to solve the minimum 2-respecting cut problem efficiently in parallel, using a combination of our new mixed batch tree operations algorithm and the use of RC trees to efficiently perform a divide-and-conquer search over the edges of the 2-constraining trees (Section 5).

Theorem 1.3. *The minimum 2-respecting cut of a weighted graph with respect to a given spanning tree can be computed in $O(m \log n)$ work and $O(\log^3 n)$ depth with high probability.*

Application to the unweighted problem. The unweighted minimum cut problem, or edge connectivity problem was recently improved by Ghafarri, Nowicki, and Thorup [15] who give an $O(m \log n + n \log^4 n)$ work and $O(\text{polylog}\, n)$ depth randomized algorithm which uses Geissmann and Gianinazzi's algorithm as a subroutine. By plugging our improved algorithm into Ghafarri, Nowicki, and Thorup's algorithm, we obtain an algorithm that runs in $O(m \log n + n \log^2 n)$ work and $O(\text{polylog}\, n)$ depth w.h.p.

2 PRELIMINARIES

Model of computation. We analyze algorithms in the *work-depth* model using fork-join parallelism. A procedure can *fork* another procedure call to run in parallel and then wait for forked procedures to complete with a *join*. Work is defined as the total number of instructions performed by the algorithm and depth (also called span) is the length of the longest chain of sequentially dependent instructions [6]. The model can work-efficiently cross simulate the classic CRCW PRAM model [6], and the more recent Binary Forking model [7] with at most a logarithmic-factor difference in the depth.

Randomness. We say that a statement happens *with high probability* (w.h.p) in n if for any constant c, the constants in the statement can be set such that the probability that the event fails to hold is $O(n^{-c})$. In line with Karger's work on random sampling [22], we assume that we can generate $O(1)$ random bits in $O(1)$ time. Since some of the subroutines we use require random $\Theta(\log n)$-bit words, these take $O(\log n)$ work to generate. The depth is unaffected since we can always pre-generate the anticipated number of random words in parallel at the beginning of our algorithms.

Our algorithms are Monte Carlo, i.e., correct w.h.p. but run in a deterministic amount of time. We can use Las Vegas algorithms, which are fast w.h.p. but always correct, as subroutines, because any Las Vegas algorithm can be converted into a Monte Carlo algorithm by halting and returning an arbitrary answer after the desired time.

Tree contraction. Parallel tree contraction is a technique developed to efficiently apply various operations over trees in logarithmic parallel depth [29], and was also later applied to dynamic trees [2]. Tree contration consists of a set of rake and compress operations. The *rake* operation removes a leaf vertex and merges it with its parent. The *compress* operation removes a vertex of degree two and replaces its two incident edges with a single edge joining its neighbors. Miller and Reif [29] observed that rakes and compresses can be applied in parallel as long as they are applied to an independent set of vertices. They describe a random-mate technique that ensures that any tree contracts to a single vertex in $O(\log n)$ rounds w.h.p., and using a total of $O(n)$ work in expectation. Gazit, Miller, and Teng [13] give a deterministic version with the same bounds, and Blelloch et al. [7] give a version that works in the binary-forking model. Miller and Reif's algorithm applies to bounded-degree trees, but arbitrary-degree trees can typically be handled by converting them into bounded-degree trees. For a rooted tree, the root is never removed, and is the final surviving vertex.

Rake-compress trees. The RC tree [1, 2] of a tree T encodes a recursive clustering of T corresponding to the result of tree contraction, where each cluster corresponds to a rake or compress (see Figure 1). A cluster is defined to be a connected subset of vertices and edges of the original tree. Importantly, a cluster can contain an edge without containing its endpoints. The *boundary vertices* of a cluster C are the vertices $v \notin C$ such that an edge $e \in C$ has v as one of its endpoints. All of the clusters in an RC tree have at most two boundary vertices. A cluster with no boundary vertices is called a *nullary cluster* (generated at the top-level *root* cluster), a cluster with one boundary is a *unary cluster* (generated by the rake operation) and a cluster with two boundaries is *binary cluster* (generated by the compress operation). The *cluster path* of a binary cluster is the path in T between its boundary vertices. Nodes in an RC tree correspond to clusters, such that a node is the disjoint union of its children.

The leaf clusters of the RC tree are the vertices and edges of the original tree, which are nullary and binary clusters respectively. Note that all non-leaf clusters have exactly one vertex (leaf) cluster as a child. This vertex is that cluster's *representative* vertex. The recursive clustering is then defined by the following simple rule: Each rake or compress operation corresponds to a cluster, such that the operation that deletes vertex v from the tree defines a cluster with representative vertex v whose non-leaf subclusters are all of the clusters that have v as a boundary vertex. Clusters therefore have the useful property that the constituent clusters of

a parent cluster C share a single boundary vertex in common—the representative of C, and their remaining boundary vertices become the boundary vertices of C.

In this paper we will be considering rooted trees. In this case the root of the tree is also the representative of the top level nullary cluster of the RC-tree, e.g. vertex e in Figure 1. Non-leaf binary clusters have a binary subcluster whose cluster path is above the representative vertex in the input tree, which we will refer to as the *top cluster*, and a binary subcluster whose cluster path is below the representative vertex, which we call the *bottom cluster*. We will also refer to the binary subcluster of a unary cluster as the top cluster as its cluster path is also above the representative vertex. In our pseudocode, we will use the following notation. For a cluster x: $x.v$ is the representative vertex, $x.t$ is the top subcluster, $x.b$ is the bottom subcluster, $x.U$ is a list of unary subclusters, and $x.p$ is the parent cluster.

Compressed path trees. For a weighted (unrooted) tree T and a set of *marked* vertices $V \subset V(T)$, the compressed path tree is a weighted tree T_c on some subset of the vertices of T including V with the following property: for every pair of vertices $(u, v) \in V \times V$, the weight of the lightest edge on the path from u to v is the same in T and T_c. The compressed path three T_c is defined as the smallest such tree. Alternatively, the compressed path tree is the tree T with all unmarked vertices of degree less than three spliced out, where each spliced-out path is replaced by an edge whose weight is the lightest of the weights on the path it replaced. It is not hard to show that T_c has size less than $2|V|$. Compressed path trees are described in [5], where it is shown that given an RC tree for the tree T and a set of k marked vertices, the compressed path tree can be produced in $O(k \log(1 + n/k))$ work and $O(\log^2 n)$ depth w.h.p. Gawrychowski et al. [12] define a similar notion which they call "topologically induced trees", but their algorithm is sequential and requires $O(k \log n)$ work (time).

Karger's minimum cut algorithm. Karger's algorithm for minimum cuts [23] is based on the notion of *k-respecting cuts*. Karger's algorithm is the following two-step process.

(1) Find $O(\log n)$ spanning trees of G such that w.h.p., the minimum cut 2-respects at least one of them
(2) Find, for each of the aforementioned spanning trees, the minimum 2-respecting cut in G

Karger solves the first step using a combination of random sampling and *tree packing*. Given a weighted graph G, a tree packing of G is a set of weighted spanning trees of G such that for each edge in G, its total weight across all of the spanning trees is no more than its weight in G. The underlying tree packing algorithms used by Karger have running time proportional to the size of the minimum cut, so random sampling is first used to produce a sparsified graph, or *skeleton*, where the minimum cut has size $\Theta(\log n)$ w.h.p. The sampling process is carefully crafted such that the resulting tree packing still has the desired property w.h.p.

Given the skeleton graph, Karger gives two algorithms for producing tree packings such that sampling $\Theta(\log n)$ trees from them guarantees that, w.h.p., the minimum cut 2-respects one of them. The first approach uses a tree packing algorithm of Gabow [11]. The second is based on the packing algorithm of Plotkin et al. [33], and is much more amenable to parallelism. It works by performing $O(\log^2 n)$ minimum spanning tree computations. In total, Step 1 of the algorithm takes $O(m + n \log^3 n)$ time.

For the second step, Karger develops an algorithm to find, given a graph G and a spanning tree T, the minimum cut of G that 2-respects T. The algorithm works by arbitrarily rooting the tree, and considering two cases: when the two cut edges are on the same root-to-leaf path, and when they are not. Both cases use a similar technique; They consider each edge e in the tree and try to find the best matching e' to minimize the weight of the cut induced by the edges $\{e, e'\}$. This is achieved by using a dynamic tree data structure to maintain, for each candidate e', the value that the cut would have if e' were selected as the second cutting edge, while iterating over the possibilities of e and updating the dynamic tree. Karger shows that this step can be implemented sequentially in $O(m \log^2 n)$ time, which results in a total runtime of $O(m \log^3 n)$ when applied to the $O(\log n)$ spanning trees.

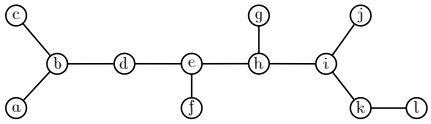
3 BATCHED MIXED OPERATIONS ON TREES

The batched mixed operation problem is to take an off-line sequence of mixed operations on a data structure, usually a mix of queries and updates, and process them as a batch. The primary reason for batch processing is to allow for parallelism on what would otherwise be a sequential execution of the operations. We use the term *operation-set* to refer to the set of operations that can be applied among the mixed operations. We are interested in operations on trees, and our results apply to operation-sets that can be implemented on an RC tree in a particular way, defined as follows.

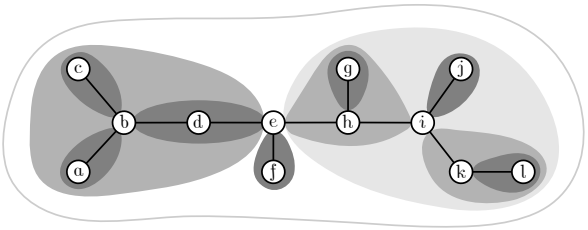
Definition 3.1. An implementation of an operation-set on trees is a *simple RC implementation* if it uses an RC representation of the trees and satisfies the following conditions.

(1) The implementation maintains a value at every RC cluster that can be calculated in constant time from the values of the children of the cluster,
(2) every query operation is implemented by traversing from a leaf to the root examining values at the visited clusters and their children taking contant time per value examined, and using constant space, and
(3) every update operation involves updating the value of a leaf using an associative constant-time operation, and then reevaluating the values on each cluster on the path from the leaf to the root.

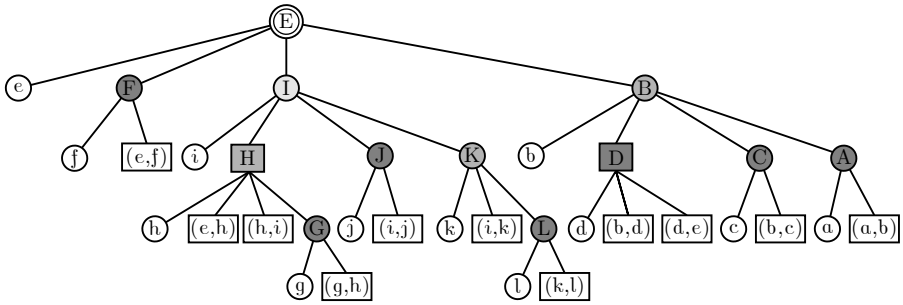
Note that every operation has an *associated leaf* (either an edge or vertex). Also note that setting (i.e., overwriting) a value is an associative operation (just return the second of the arguments). For simple RC implementations, all operations take time (work) proportional to the depth of the RC tree since they only follow a path to the root taking constant time at each cluster. Although the simple RC restriction may seem contrived, most operations on trees studied in previous work [2, 3, 35] can be implemented in this form, including most path and subtree operations. This is because of a useful property of RC trees, that all paths and subtrees in the source tree can be decomposed into clusters that are children of a single path in the RC tree, and typically operations need just update or collect a contribution from each such cluster.



(a) A tree



(b) A recursive clustering of the tree produced by tree contraction. Clusters produced in earlier rounds are depicted in a darker color.



(c) The corresponding RC tree. Unary clusters (from rakes) are shown as filled circles, binary clusters as rectangles, and the finalize (nullary) cluster at the root with two concentric circles. The leaf clusters are labeled in lowercase, and the composite clusters are labeled with the uppercase of their representative.

Figure 1: A tree, a clustering, and the corresponding RC tree [1].

Example. As an example, consider the following two operations on a rooted tree (the first an update, and the second a query):

- addWeight(v, w) : adds weight w to a vertex v
- subtreeSum(v) : returns the sum of the weights of all of the vertices in the subtree rooted at v

Algorithm 1 The subtreeSum query.

```
1: procedure subtreeSum(v)
2:     w ← 0
3:     x ← v; p ← x.p
4:     while p is binary do
5:         if (x = p.t) or (x = p.v) then
6:             w ← w + p.b.w + p.v.w + Σ_{u∈p.U} u.w
7:         x ← p; p ← x.p
8:     return w + p.v.w + Σ_{u∈p.U} u.w
```

These operations can use a simple RC implementation by keeping as the value of each cluster the sum of values of all its children. This satisfies the first condition since the sums take constant time. Single-edge clusters in the RC tree start with the initial weight of the edge, while single-vertex clusters start with zero weight. An addWeight(v, w) adds weight w to the vertex v (which is a leaf in the RC tree) and updates the sums up to the root cluster. This satisfies the third condition since addition is associative and takes constant time. The query can be implemented as in Algorithm 1, where $x.w$ is the weight stored on the cluster x. It starts at the leaf for v and goes up the RC tree keeping track of the total weight underneath v. Note that x will never be a unary cluster, so if not the representative or top subcluster of p, it is the bottom subcluster with nothing below it in this cluster. Observe that subtreeSum only examines values on a path from the start vertex to the root and the children along that path. Each step takes constant time and requires constant space, satisfying the second condition. The operations-set therefore has a simple RC implementation.

3.1 Batched mixed operations algorithm

We are interested in evaluating batches of operations from an operation-set on trees with a simple RC implementation. In particular, we prove Theorem 1.2.

Proof sketch of Theorem 1.2. The preprocessing just builds an RC tree on the source tree, and sets the values for each cluster based on the initial values on the leaves. This can be implemented with the Miller-Reif algorithm [29], in the binary forking model [7], or deterministically [13]. All take linear work and logarithmic depth (w.h.p for the randomized versions). Our algorithm for each batch is then implemented as follows:

(1) Timestamp the operations by their order in the sequence.
(2) Collect all operations by their associated leaf, and sort within each leaf by timestamp. This can be implemented with a single sort on the leaf identifier and timestamp.
(3) For each leaf use a prefix sum on the update values to calculate the value of the leaf after each operation, starting from the initial value on the leaf.

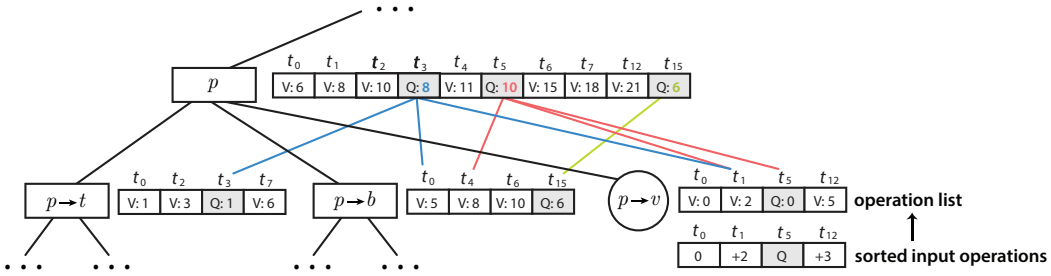


Figure 2: Merging the operation lists for a binary cluster consisting of AddWeight and subtreeSum operations. Values in the operation sequence, denoted V : v, are computed by aggregating the latest values of the children at the given timestamp. For example, at t_6 in p, the algorithm adds 3 from $p.t$ at t_2, 10 from $p.b$ at t_6, and 2 from $p.v$ at t_1. Queries, denoted Q : q, are updated at each level by using the latest values of the children. For example, to update the query at t_3, it takes the current value of 1 from $p.t$ at t_3, then adds the weight of 5 from $p.b$ at t_0, and the weight of 2 from $p.v$ at t_1, as per Algorithm 1.

(4) Initialize each query using the value it received from the prefix sum. We now have a list of operations on each leaf sorted by timestamp. For each update we have its value, and for each query we also have its partial evaluation based on the value. We prepend the initial value to the list, and call this the *operation list*. An operation list is *non-trivial* if it has more than just the initial value.
(5) For each level of the RC tree starting one above the deepest, and in parallel for every cluster on the level for which at least one child has a non-trivial operation list:
   (a) Merge the operation lists from each child into a single list sorted by timestamp.
   (b) Calculate for each element in the merged operations list, the latest value of each child at or before the timestamp. This can be implemented by prefix sums.
   (c) For each list element, calculate the value at that timestamp from the child values collected in the previous step.
   (d) For queries, use the values and/or child values to update the query.

This algorithm needs to have children with non-trivial operation lists identify parents that need to be processed. This can be implemented by keeping a list of all the clusters at a level with non-trivial operation lists left-to-right in level order. When moving up a level, clusters that share the same parent can be combined. An illustration of the merging process is depicted in Figure 2 using the operations from Algorithm 1.

We first consider why the algorithm is correct. We assume by structural induction (over subtrees) that the operation lists contain the correct values for each timestamped operation in the list. This is true at the leaves since we apply a prefix sum across the associative operation to calculate the value at each update. For internal clusters, assuming the child clusters have correct operation lists (values for each timestamp valid until the next timestamp, and partial result of queries), we properly determine the operation lists for the cluster. In particular for all timestamps that appear in children we promote them to the parent, and for each we calculate the value based on the current value, by timestamp, for each child.

We now consider the costs. The cost of the batch before processing the levels is dominated by the sort which takes $O(k \log k)$ work and $O(\log k)$ depth. The cost at each level is then dominated by the merging and prefix sums which take $O(k)$ work and $O(\log k)$ depth accumulated across all clusters that have a child with a non-trivial operation list. If the RC tree has depth $O(\log n)$ then across all levels the cost is bounded by $O(k \log n)$ work and $O(\log n \log k)$ depth. The total work and depth is therefore as stated. The space for each batch of size k is bounded by the size of the RC tree which is $O(n)$ and the total space of the operation lists at any two adjacent levels, which is $O(k)$. □

3.2 Path updates and path/subtree queries

We now consider implementing mixed operations consisting of updating paths, and querying both paths and subtrees. We will use these in Sections 3.3 and 5. In particular we wish to maintain, given a weighted rooted tree $T = (V, E)$, a data structure that supports the following operations.

- AddPath(u, v, w): For $u, v \in V$ adds w to the weight of all edges on the u to v path.
- QuerySubtree(v): Returns the lightest weight of an edge in the subtree rooted at $v \in V$,
- QueryPath(u, v): For $u, v \in V$, returns the lightest weight of an edge on the u to v path.
- QueryEdge(e): Returns $w(e)$

To implement these, we first implement the simpler operations AddPath'(v, w), which adds weight w to the path from v to the root; and QueryPath'(u, v), which requires that v be the representative vertex of an ancestor of u in the RC tree. The more general forms can be implemented in terms of these with a constant number of calls given the lowest common ancestor (LCA) in the original tree for AddPath and in the RC tree for QueryPath.

Lemma 3.2. *The AddPath', QuerySubtree, QueryPath', and QueryEdge operations on bounded degree trees can be supported with a simple RC implementation.*

Proof sketch. Our simple RC implementation for combining values, AddPath', and QuerySubtree is given in Algorithm 2. The other two operations can be found in the full version of our paper [4]. The value of each vertex (leaf) in the cluster is the total weight added to that vertex by AddPath'. The value for each unary

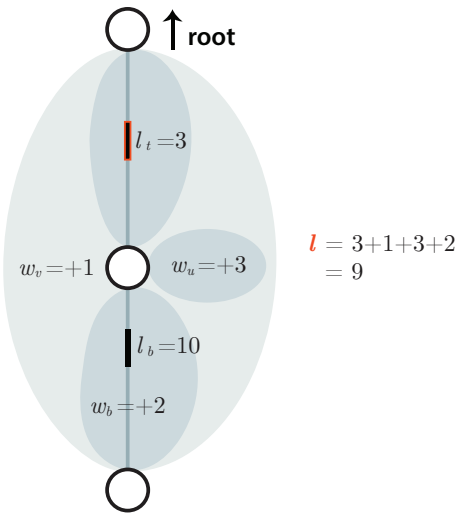


Figure 3: When a binary cluster joins its children, all AddPaths' that originated in the vertex, bottom, or unary subclusters will affect all of the edges in the top cluster path. Here, $w' = w_v + w_b + w_u = 6$ weight is added to edges on the top cluster path due to AddPath operations from below.

Algorithm 2 AddPath' and QuerySubtree.

```
1:  procedure f_UNARY(w_v, (m_t, l_t, w_t), U)
2:      w' ← w_v + Σ_{u∈U} u.w
3:      m_u ← min_{u∈U} u.m
4:      return (min(m_t, l_t + w', m_u), w_t + w')
5:  procedure f_BINARY(w_v, (m_t, l_t, w_t), (m_b, l_b, w_b), U)
6:      w' ← w_v + w_b + Σ_{u∈U} u.w
7:      m_u ← min_{u∈U} u.m
8:      return (min(m_t, m_b, m_u), min(l_t + w', l_b), w_t + w')
9:  procedure AddPath'(v, w)
10:     v.value ← v.value + w
11:     Reevaluate the f(·) on path to root.
12: procedure QuerySubtree(v)
13:     w ← ∞; l ← ∞
14:     x ← v; p ← x.p
15:     while binary p do
16:         if (x = p.t) or (x = p.v) then
17:             w' ← p.b.w + p.v.w + Σ_{u∈p.U} u.w
18:             l ← min(l + w', p.b.l)
19:             m ← min(m, p.b.m, min_{u∈p.U} u.m)
20:         x ← p; p ← x.p
21:     w' ← p.v.w + Σ_{u∈p.U} u.w
22:     return min(l + w', m, min_{u∈p.U} u.m)
```

cluster consists of: m, the minimum weight edge in the cluster; and w, the total weight of AddPaths' originating in the cluster. For each binary cluster we separate the minimum weights on and off the cluster path. In particular, the value of each binary cluster consists of: m, the minimum weight edge not on the cluster path; l, the minimum edge on the cluster path due to all AddPath' originating in the cluster; and w, the total weight of AddPaths' originating in the cluster. The f_{binary} and f_{unary} calculate the values for unary and binary clusters from the values of their children. We initialize each vertex with zero, and each edge e with $(m = 0, l = w(e), w = 0)$.

It is a simple RC implementation since (1) the $f(\cdot)$ can be computed in constant time, (2) the queries just traverse from a leaf on a path to the root (possibly ending early) only examining child values, taking constant time per level and constant space, and (3) the update just sets a leaf using an associative addition, and reevaluates the values to the root.

We argue the implementation is correct. Firstly we argue by structural induction on the RC tree that the values as described in the previous paragraph are maintained correctly by f_{binary} and f_{unary}. In particular assuming the children are correct we show the parent is correct. The values are correct for leaves since we increment the value on vertices with AddPath', and initialize the edges appropriately. To calculate the minimum edge weight of a unary cluster f_{unary} takes the minimum of three quantities: the minimum off-path edge of the child binary cluster, the overall minimum edge of any of the child unary clusters, and, importantly, the minimum edge on the cluster path of the child binary cluster plus the AddPath' weight contributed by the unary clusters and the representative vertex (i.e., $\min(m_t, l_t + w', m_u)$). This is correct since all paths from those clusters to the root go through the cluster path, so it needs to be adjusted. The off-path edges and child unary clusters do not need to be adjusted since no path from the representative vertex goes through them. The minimum weight is therefore correct. The total AddPath' weight is correct since it just adds the contributions.

For binary clusters we need to separately consider the minimum off- and on-path edges. For the off-path edges the parts that are off the cluster path are the off-path edges from the two binary children, plus all edges from the unary children (i.e., $\min(m_t, m_b, m_u)$). For the on-path edges both the top and bottom binary clusters contribute their on-path edges. The on-path edges from the bottom binary cluster do not need to be adjusted because no vertices in the cluster are below them. The on-path edges from the top binary cluster need to be adjusted by the AddPath' weights from all vertices in the bottom cluster, all vertices in unary child clusters, and the representative vertex since they are all below the path (this sum is given by w'). See Figure 3. The minimum of the resulted adjusted top edge and bottom edge is then returned, which is indeed the minimum edge on the path accounting for AddPaths' on vertices in the cluster.

QuerySubtree(v) accumulates the appropriate minimum weights within a subtree as it goes up the RC tree. It starts at the node for which v is its representative vertex. As with the calculation of values it needs to separate the on-path and off-path minimum weight. Whenever coming as the upper binary cluster to the parent, QuerySubtree needs to add all the contributing AddPath' weights from vertices below it in the parent cluster (the representative vertex, the lower binary cluster and the unary clusters, see Figure 3) to the current minimum on-path weight. A minimum is then taken with the lower on-path minimum edge to calculate the new minimum on-path edge weight (Line 18). The off-path minimum is the minimum of the current off-path minimum, the minimum off-path edge of the bottom cluster and the minimums of the unary clusters

(Line 19). Once we reach a unary cluster we are done since for a unary cluster all subtrees of vertices within the cluster are fully contained within the cluster. The final line therefore just determines the overal minimum for the subtree rooted at v by considering the on-path edges adjusted by AddPath' contributions, the off-path edges, and all edges in child unary clusters. □

Corollary 3.3. *Given a bounded-degree tree of size n, any sequence of k AddPath, QuerySubtree, QueryPath, and QueryEdge operations can be evaluated in $O(n + k\log(nk))$ work, $O(\log n \log k)$ depth and $O(n + k)$ space.*

Proof. The LCAs required to convert AddPath to AddPath' and QueryPath to QueryPath' can be computed in $O(n + m)$ work, $O(\log n)$ depth, and $O(n)$ space [34]. The rest follows from Theorem 1.2 and Lemma 3.2. □

3.3 Improving previous results

Using our batched mixed operations on trees algorithm, we can improve previous results on finding 2-respecting cuts. In particular we can shave off a log factor in the work of Geissmann and Gianinazzi's parallel algorithm [14], and we can parallelise Lovett and Sandlund's sequential algorithm [27].

Geissmann and Gianinazzi find 2-respecting cuts by first finding an $O(m)$ sequence of mixed AddPath and QueryPath operations for each of $O(\log n)$ trees. They show how to find each sequence in $O(m \log n)$ work and $O(\log n)$ depth On each set they then use their own data structure to evaluate the sequence in $O(m \log^2 n)$ work and $O(\log^2 n)$ depth, for a total of $O(m \log^3 n)$ work and $O(\log^2 n)$ depth across the sets. Replacing their data structure with the result of Corollary 3.3 improves their results to $O(m \log^2 n)$ work.

Lovett and Sandlund significantly simplify Karger's algorithm by first finding a heavy-light decomposition—i.e., a vertex disjoint set of paths in a tree such that every path in the tree is covered by at most $O(\log n)$ of them. It then reduces finding the 2-respecting cuts to a sequence of AddPath and QueryPath operations on the decomposed paths induced by each non-tree edge, for a total of $O(m \log n)$ operations. Using Geissmann and Gianinazzi's $O(n \log n)$ work $O(\log^2 n)$ algorithm for finding a heavy-light decomposition [14, Lemma 7], and the result of Corollary 3.3 again gives an $O(m \log^2 n)$ work, $O(\log^2 n)$ depth algorithm.

4 PRODUCING THE TREE PACKING

We follow the general approach used by Karger to produce a set of $O(\log n)$ spanning trees such that w.h.p., the minimum cut 2 respects at least one of them. We have to make several improvements to achieve our desired work and depth bounds. At a high level, Karger's algorithm works as follows.

(1) Compute an $O(1)$-approximate minimum cut c
(2) Sample the edges of G with probability $\Theta(\log n/c)$
(3) Use the tree packing algorithm of Plotkin [33] to generate a packing of $O(\log n)$ trees

In this section, we provide a high-level overview of the tools required to parallelize this algorithm. The details are deferred to the full paper [4].

4.1 A parallel version

Step 2 is trivial to parallelize, as the sampling can be done independently in parallel. The sampling procedure produces an unweighted multigraph with $O(m \log n)$ edges, and takes $O(m \log^2 n)$ work and $O(\log n)$ depth.

In Step 3, Plotkin's algorithm consists of $O(\log^2 n)$ sequential minimum spanning tree (MST) computations on a weighting of the sampled graph, which has $O(m \log n)$ edges. Naively this would require $O(m \log^3 n)$ work, but we can use a trick of Gawrychowski et al. [12]. Since the sampled graph is a multigraph sampled from m edges, the MST algorithm only cares about the lightest of each parallel edge, which can be maintained in $O(1)$ time since the weights change by a fixed amount each iteration. Using Cole, Klein, and Tarjan's linear-work MST algorithm [9] results in a total of $O(m \log^2 n)$ work in $O(\log^3 n)$ depth w.h.p.

The only nontrivial part of parallelizing the tree production is actually Step 1, computing an $O(1)$-approximate minimum cut. In the sequential setting, Matula's algorithm [28] can be used, which runs in linear time on unweighted graphs, and on weighted graphs in $O(m \log^2 n)$ time. To the best of our knowledge, the only known parallelization of Matula's algorithm is due to Karger and Motwani [24], but it takes $O(m^2/n)$ work. We show how to compute an approximate minimum cut in $O(m \log^2 n)$ work and $O(\log^3 n)$ depth, which allows us to prove the following.

Theorem 4.1. *Given a weighted graph, in $O(m \log^2 n)$ work and $O(\log^3 n)$ depth, a set of $O(\log n)$ spanning trees can be produced such that the minimum cut 2-respects at least one of them w.h.p.*

4.2 Parallel $O(1)$-approximate minimum cut

We achieve our bounds by improving Karger's algorithms and speeding up several of the components. We use the following combination of ideas, new and old.

(1) We extend a k-approximation algorithm of Karger [21] to work in parallel, allowing us to produce an $O(\log n)$-approximate minimum cut in low work and depth.
(2) We use a faster sampling technique for producing Karger's skeletons for weighted graphs. This is done by transforming the graph into a graph that maintains an approximate minimum cut but has edge weights each bounded by $O(m \log n)$, and then using binomial random variables to sample all of the multiedges of a particular edge at the same time, instead of separately.
(3) We show that the parallel sparse k-certificate algorithm of Cheriyan, Kao, and Thurimella [8] for unweighted graphs can be modified to run on weighted graphs
(4) We show that Karger and Motwani's parallelization of Matula's algorithm can be generalized to weighted graphs
(5) We use the $\log n$-approximate minimum cut to allow the algorithm to make just $O(\log \log n)$ guesses of the minimum cut such that at least one of them is an $O(1)$ approximation.

Parallel k-approximate minimum cut. Karger describes an $O(mn^{2/k} \log n)$ time sequential algorithm for finding a cut in a graph within a factor of k of the optimal cut [21]. It works by randomly selecting edges to contract with probability proportional to their weight until a single vertex remains, and keeping track of the

component with smallest incident weight (not including internal edges) during the contraction.

His analysis shows that in a weighted graph with minimum cut c, with probability $n^{-2/k}$, the component with minimum incident weight encountered during a single trial of the contraction algorithm implies a cut of weight at most kc, and therefore, running $O(n^{2/k}\log n)$ trials yields a cut of size at most kc w.h.p.

Although Karger's contraction algorithm is easy to parallelize using a parallel minimum spanning tree algorithm, keeping track of the incident component weights is trickier. To overcome this problem, we show that we can use our batched mixed operation framework from Section 3 to simulate the sequential contraction process efficiently. Specifically, we show that that the following operations have a simple RC implementation.

- SubtractWeight(v, w): Subtract weight w from vertex v
- JoinEdge(e): Mark the edge e as "joined"
- QueryWeight(v): Return the weight of the connected component containing the vertex v, where the components are induced by the joined edges

With this tool, it is straightforward to simulate the contraction process, then we obtain the following lemma.

Lemma 4.2. *For a weighted graph, a cut within a factor of k of the minimum cut can be found w.h.p. in $O(mn^{2/k}\log^2 n)$ work and $O(\log^2 n)$ depth.*

Setting $k = \log n$ therefore gives a $\log n$ approximation in $O(m\log^2 n)$ work and $O(\log^2 n)$ depth.

Transformation to bounded edge weights. For our algorithm to be efficient, we require that the input graph has small integer weights. Karger [20] gives a transformation that ensures all edge weights of a graph are bounded by $O(n^5)$ without affecting the minimum cut by more than a a constant factor. For our algorithm $O(n^5)$ would be too big, so we design a different transformation that guarantees all edge weights are bounded by $O(m\log n)$, and only affects the weight of the minimum cut by a constant factor.

Lemma 4.3. *There exists a transformation that, given an integer-weighted graph G, produces an integer-weighted graph G' no larger than G, such that G' has edge weights bounded by $O(m\log n)$, and the minimum cut of G' corresponds to an $O(1)$-approximate minimum cut in G.*

Parallel weighted sampling. We combine recent results on sampling binomial random variables [10] and parallel alias table construction [19] to perform samples from $B(n', 1/2)$ in $O(\log n')$ time w.h.p., and from $B(n', p)$ in $O(\log^2 n')$ time w.h.p., for any $n' \le N$ after $O(N^{1/2+\varepsilon})$ work preprocessing and $O(\log N)$ depth. Since we preprocess the graph to have weights at most $O(m\log n)$, this requires no more than $O(m)$ work in preprocessing.

This does not immediately give the desired bounds, since it takes $O(\log^2 n)$ work per edge when sampling from $B(n,p)$, and our algorithm samples the graph $O(\log\log n)$ times. However, only the first sample of the graph needs to be this expensive. In Karger's algorithm, and by extension, our algorithm, subsequent samples always halve p in each iteration, and hence we can use subsampling and only require random variables from $B(n, 1/2)$. This means that we can perform up to $O(\log n)$ rounds of subsampling in $O(m\log^2 n)$ total work, instead of $O(m\log^3 n)$ work.

Sparse certificates. A *sparse k-connectivity certificate* of a graph $G = (V, E)$ is a graph $G' = (V, E' \subset E)$ with at most $O(kn)$ edges, such that every cut in G of weight at most k has the same weight in G'. We show that the sparse certificate algorithm of Cheriyan, Kao, and Thurimella [8] for unweighted graphs can be easily extended to weighted graphs, with the following result.

Lemma 4.4. *A sparse k-connectivity certificate for a weighted, undirected graph can be found in $O(km)$ work and $O(k\log n)$ depth.*

Parallelizing Matula's algorithm. Matula [28] gave a linear time sequential algorithm for $(2+\varepsilon)$-approximate edge connectivity (unweighted minimum cut). It is easy to extend to weighted graphs so that it runs in $O(m\log n\log W)$ time, where W is the total weight of the graph. Using standard transformations to obtain polynomially bounded edge weights, this gives an $O(m\log^2 n)$ algorithm. Karger and Motwani [24] gave a parallel version of Matula's unweighted algorithm that runs in $O(m^2/n)$ work. A slight modification to this algorithm makes it work on weighted graphs in $O(dm\log(W/m))$ work and $O(d\log n\log W)$ depth, where d is the minimum weighted degree of the graph.

Lemma 4.5. *Given a weighted graph with minimum weighted-degree d and total weight W, an $O(1)$-approximate minimum cut can be found in $O(dm\log(W/m))$ work and $O(d\log n\log W)$ depth.*

Parallel $O(1)$-approximate minimum cut. The final ingredient needed to produce the parallel minimum cut approximation is a trick due to Karger. Recall that to produce the skeleton graph, the sampling probability must be inversely proportional to the weight of the minimum cut, which paradoxically is what we are trying to compute. This issue is solve by using *doubling*. The algorithm makes successively larger guesses of the minimum cut and computes the resulting approximation. It can then use Karger's sampling theorem (Lemma 6.3.2 of [20]) to verify whether the guess was too high. To minimize the work, we use Lemma 4.2 to first produce a $O(\log n)$-approximation to the minimum cut, which allows us to make just $O(\log\log n)$ guesses such that one of them will be correct to within a factor two.

Our algorithm proceeds by making these $O(\log\log n)$ guesses in parallel. For each, we sample a corresponding skeleton graph and compute a $\Theta(\log n)$ certificate, since, by the sampling theorem, until we have made the correct guess, the minimum cut in the skeleton will have weight $O(\log n)$ w.h.p. This then guarantees that we can run our version of parallel Matula's algorithm in $O(n\log n\log\log n)$ work, since, after producing the certificate, the total weight of the graph is at most $O(n\log n)$, and the minimum weighted degree is no more than $O(\log n)$. Taking every ingredient together allows us to conclude the following lemma.

Lemma 4.6. *Given a weighted, undirected graph, the weight of an $O(1)$-approximate minimum cut can be computed w.h.p. in $O(m\log^2 n)$ work and $O(\log^3 n)$ depth*

5 FINDING MINIMUM 2-RESPECTING CUTS

We are given a connected, weighted, undirected graph $G = (V, E)$ and a spanning tree T. In this section, we will give an algorithm

that finds the minimum 2-respecting cut of G with respect to T in $O(m \log n)$ work and $O(\log^3 n)$ depth.

Our algorithm, like those that came before it, finds the minimum 2-respecting cut by considering two cases. We assume that the tree T is rooted arbitrarily. In the first case, we assume that the two tree edges of the cut occur along the same root-to-leaf path, i.e. one is a descendant of the other. This is called the *descendant edges* case. In the second case, we assume that the two edges do not occur along the same root-to-leaf path. This is the *independent edges* case.

Since we are going to use RC trees, we require that G have bounded degree. Note that any arbitrary degree graph can easily be *ternarized* by replacing high-degree vertices with cycles of infinite weight edges, resulting in a graph of maximum degree three with the same minimum cut, and only a constant-factor larger size in terms of edges, which our bounds depend on.

5.1 Descendant edges

We present our minimum 2-respecting cut algorithm for the descendant edges case. Let T be a spanning tree of a connected graph $G = (V, E)$ of degree at most three, and root T at an arbitrary vertex of degree at most two. The rooted tree is therefore a binary tree.

We use the following fact. For any tree edge $e \in T$, let F_e denote the set of edges $(u, v) \in E$ (tree and non-tree) such that the u to v path in T contains the edge e. Then the weight of the cut induced by a pair of edges $\{e, e'\}$ in T is given by

$$w(F_e \Delta F_{e'}) = w(F_e) + w(F_{e'}) - 2w(F_e \cap F_{e'}),$$

where Δ denotes the symmetric difference between the two sets. For each tree edge e, our algorithm seeks the tree edge e' that minimizes $w(F_e \Delta F_{e'})$, which is equivalent to minimizing

$$w(F_{e'}) - 2w(F_e \cap F_{e'}).$$

To do so, it traverses T from the root while maintaining weights on a tree data structure that satisfies the following invariant:

Invariant 1 (Current subtree invariant). *When visiting $e = (u, v)$, for every edge $e' \in Subtree(v)$, the weight of e' in the dynamic tree is $w(F_{e'}) - 2w(F_e \cap F_{e'})$*

The initial weight of each edge e is therefore $w(F_e)$. Maintaining this invariant as the algorithm traverses the tree can then be achieved with the following observation. When the traversal descends from an edge $p = (w, u)$ to a neighboring child edge $e = (u, v)$, the following hold for all $e' \in \text{Subtree}(v)$:

(1) $(F_e \cap F_{e'}) \supseteq (F_p \cap F_{e'})$, since any path that goes through p and e' must pass through e.
(2) $(F_e \cap F_{e'}) \setminus (F_p \cap F_{e'})$ are the edges $(x, y) \in F_{e'}$ such that e is a *top edge* of the path $x - y$ in T (i.e., e is on the path from x to y in T, but the parent edge of e is not).

Therefore, to maintain the current subtree invariant, when the algorithm visits the edge e, it need only subtract twice the weight of all $x - y$ paths that contain e as a top edge. This can be done efficiently by precomputing the sets of top edges. There are at most two top edges for each path $x - y$, and they can be found from the LCA of x and y in T. We need not consider tree edges since they will never appear in $F_{e'}$. By maintaining the aforementioned invariant, the solution follows by taking the minimum value of $w(F_e)$ + QuerySubtree(v) for all edges $e = (u, v)$ during the traversal. As described, this algorithm is entirely sequential, but it can be parallelized using our batched mixed operations on trees algorithm (Corollary 3.3).

The operation sequence can be generated as follows. First, the weights $w(F_e)$ for each edge can be computed using the batched mixed operations algorithm (Corollary 3.3) where each edge (u, v) of weight w creates an AddPath(u, v, w) operation, followed by a QueryEdge(e) for every edge $e \in T$. This takes $O(m \log n)$ work and $O(\log^2 n)$ depth. The LCAs required to compute the sets of top edges can be computed using the parallel LCA algorithm of Schieber and Vishkin [34] in $O(m)$ work and $O(\log n)$ depth in total. By computing an Euler tour of the tree T (an ordered sequence of visited edges) beginning at the root, the order in which to perform the tree operations can be deduced in $O(n)$ work and $O(\log n)$ depth. Each edge in the Euler tour generates an AddPath operation for each of its top edges, followed by a QuerySubtree operation. Note that each edge is visited twice during the Euler tour. The second visit corresponds to negating the AddPath operations from the first visit. The solution is then the minimum result of all of the QuerySubtree operations. Since there are a constant number of top edges per path, and $O(m)$ paths in total, the operation sequence has length $O(m)$. Using Corollary 3.3, we arrive at the following.

Theorem 5.1. *Given a weighted, undirected graph G and a rooted spanning tree T, the minimum 2-respecting cut of G with respect to T such that one of the cut edges is a descendant of the other can be computed in in $O(m \log n)$ work and $O(\log^2 n)$ depth w.h.p.*

5.2 Independent edges

The independent edge case is where the two cutting edges do not fall on the same root-to-leaf path. To solve the independent edges problem, we use the framework of Gawrychowski et al. [12], which is to decompose the problem into a set of subproblems, which they call *bipartite problems*. The key challenge in parallelizing the solution to the bipartite problem is dealing with the fact that the resulting trees might not be balanced. The algorithm of Gawrychowski et al. relies on performing a biased divide-and-conquer search guided by a heavy-light decomposition [18], and then propagating results up the trees bottom up. Since the trees may be unbalanced, this can not be easily parallelized. Our solution is to use the recursive clustering of RC trees to guide a divide and conquer search in which we can maintain all of the needed information on the clusters.

Definition 5.2 (The bipartite problem). Given two weighted rooted trees T_1 and T_2 and a set of weighted edges that cross from one to the other, $L = \{(u, v) : u \in T_1, v \in T_2\}$, the bipartite problem is to select $e_1 \in T_1$ and $e_2 \in T_2$ with the goal of minimizing the sum of the weight of e_1 and e_2 plus the weights of all edges $(v_1, v_2) \in L$ such that v_1 is in the subtree rooted at the bottom endpoint of e_1 and v_2 is in the subtree rooted at the bottom endpoint of e_2. The size of a bipartite problem is the size of L plus the size of T_1 and T_2.

Gawrychowski et al. observe that if T_1 and T_2 are edge-disjoint subtrees of T, then, assigning weights of $-2w(F_e)$ to each edge in T, the solution to the bipartite problem is the minimum 2-respecting cut such that $e_1 \in T_1$ and $e_2 \in T_2$. The independent edges problem is then solved by reducing it to several instances of the bipartite

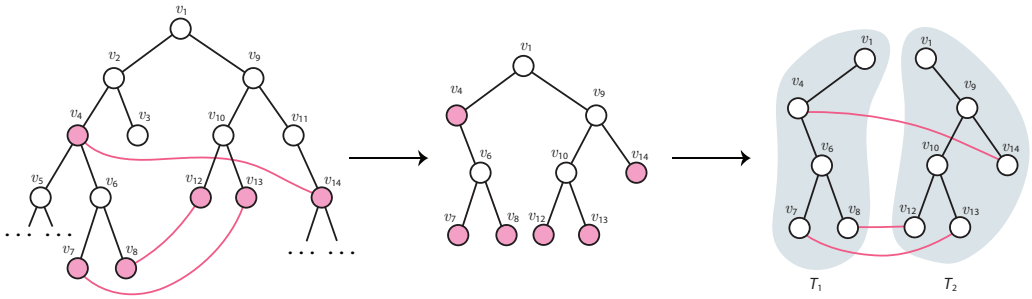


Figure 4: The bipartite problems are generated by compressing the input tree with respect to the endpoints of the edges whose endpoints share an LCA, then splitting the tree into the left and right halves.

problem, and taking the minimum answer among all of them. We will show how to generate the bipartite problems efficiently, and how to solve them efficiently, both in parallel.

5.2.1 Generating the bipartite problems. The following parallel algorithm generates $O(n)$ instances of the bipartite problem with total size at most $O(m)$. For each edge e in T, the algorithm first assigns them a weight equal to $-2w(F_e)$. Now consider all non-tree edges, i.e. all edges $e \in E, e \notin T$, and group them by the LCA of their endpoints in T. This forms a partition of the $O(m)$ edges of G, each group identified by a vertex. Each vertex in T conversely has an associated (possibly empty) list of non-tree edges.

For each vertex v in T with a non-empty associated list of edges, create a compressed path tree of T with respect to the endpoints of the associated edges and v. Finally, for each such compressed path tree, root it at v (the common LCA of the edge endpoints). The bipartite problems are now generated as follows. For each vertex v with a non-empty list of non-tree edges, and the corresponding compressed path tree T_v, consider the children x, y of v in T_v. The bipartite problem consists of T_1, which contains the edge (v, x) and the subtree of T_v rooted at x, and likewise, T_2, which contains the edge (v, y) and the subtree of T_v rooted at y, and L, the associated list of non-tree edges. See Figure 4 for an illustration.

Lemma 5.3. *Given a tree and a set of non-tree edges, the corresponding bipartite problems can be generated in $O(m \log n)$ work and $O(\log^2 n)$ depth w.h.p.*

Proof. The edge weight values can be computed in the same way as before using our batched mixed operations on trees algorithm in $O(m \log n)$ work and $O(\log^2 n)$ depth. LCAs can be computed using the parallel LCA algorithm of Schieber and Vishkin [34] in $O(m)$ work and $O(\log n)$ depth. Grouping the edges by LCA can be achieved using a parallel sorting algorithm in $O(m \log n)$ work and $O(\log n)$ depth. Together, these steps take $O(m \log n)$ work and $O(\log^2 n)$ depth. For each group, computing the compressed path tree takes $O(m_i \log(1 + n/m_i)) \leq O(m_i \log n)$ work and $O(\log^2 n)$ depth w.h.p., where m_i is the number of edges in the group. Performing all compressed path tree computations in parallel and observing that the edge lists of each vertex are a disjoint partition of the edges of G, this takes at most $O(m \log n)$ work and $O(\log^2 n)$ depth in total w.h.p. □

It remains only for us to show that the bipartite problems can be efficiently solved in parallel.

5.2.2 Solving the bipartite problems. Our solution is a recursive algorithm that utilizes the recursive cluster structure of RC trees. Recall that RC trees consist of unary and binary clusters (and the nullary cluster at the root, but this is not needed by our algorithm). Since the bipartite problems are constructed such that trees T_1 and T_2 always have a root with a single child, the root cluster of their RC trees consists of exactly one unary cluster.

High-level idea. Recall that the goal is to select an edge $e_1 \in T_1$ and an edge $e_2 \in T_2$ that minimizes their costs plus the cost of all edges $(u, v) \in L$ such that u is a descendant of e_1 and v is a descendant of e_2. Our algorithm first constructs an RC tree of T_1, and weights the edges in T_1 and T_2 by their cost. At a high level, the algorithm then works as follows. Given a binary cluster c_1 of T_1, the algorithm maintains weights on T_2 such that for each edge $e_2 \in T_2$, its weight is the weight of e_2 in the original tree plus the sum of the weights of all edges $(u, v) \in L$ such that u is a descendant of the bottom boundary of c_1, and v is a descendant of e_2. This implies that for a binary cluster of T_1 consisting of an isolated edge $e_1 \in T_1$, the weights of each $e_2 \in T_2$ are precisely such that $w(e_1) + w(e_2)$ is the value of selecting $\{e_1, e_2\}$ as the solution. This idea leads to a very natural recursive algorithm. We start with the topmost unary cluster of T_1 and proceed recursively down the clusters of T_1, maintaining T_2 with weights as described. When the algorithm recurses into the top binary child of a cluster, it must add the weights of all $(u, v) \in L$ that are descendants of that cluster to the corresponding paths in T_2. If recursing on the bottom binary subcluster of a binary cluster, the weights on T_2 are unchanged. When recursing on a unary cluster, since it has no descendants, the algorithm uses the original weights of T_2. Once the recursion hits a binary cluster that consists of a single edge e_1, it can return the solution $w(e_1) + w(e_2)$, where e_2 is the lightest edge with respect to the current weights on T_2. Lastly, to perform this process efficiently, the algorithm *compresses*, using the compressed path tree algorithm [5], the tree T_2 every time it recurses, keeping only the vertices that are endpoints of the crossing edges that touch the current cluster of T_1.

Implementation. We provide pseudocode for our algorithm in Algorithm 3. Given a bipartite problem (T_1, T_2, L), we use the notation $L(C)$ to denote the edges of L limited to those that are incident

on some vertex in the cluster C. Furthermore, we use $V_{T_2}(L(C))$ to denote the set of vertices given by the endpoints of the edges in $L(C)$ that are in T_2. The pseudocode does not make the parallelism explicit, but all that is required is to run the recursive calls in parallel. The procedure takes as input a cluster C of T_1, a compressed version of T_2 with its original weights, and T_2', the compressed version of T_2 with updated weights. At the top level, it takes the cluster representing all of T_1 for the first argument, and the cluster for all of T_2 for the second and third argument. The Compress function compresses the given tree with respect to the given vertex set and its root, and returns the compressed tree still rooted at the same root. AddPaths(S) takes a set $S \subset L$ of edges and for each one, adds $w(u, v)$ to the root-to-v path, where $v \in T_2$, returning a new tree.

Algorithm 3 Parallel bipartite problem algorithm

```
1:  procedure Bipartite(C, T_2, T_2', L)
2:    if C = {e} then
3:      return w(e) + LightestEdge(T_2')
4:    else
5:      T_cmp ← T_2.Compress(V_{T_2}(L(C.t)))
6:      T_2'' ← T_2'.AddPaths(L(C) \ L(C.t))
7:      T_cmp'' ← T_2''.Compress(V_{T_2}(L(C.t)))
8:      ans ← Bipartite(C.t, T_cmp, T_cmp'', L(C.t))
9:      for each cluster C' in C.U do
10:       T_cmp ← T_2.Compress(V_{T_2}(L(C')))
11:       ans ← min(ans, Bipartite(C', T_cmp, T_cmp, L(C')))
12:     if C is a binary cluster then
13:       T_cmp ← T_2.Compress(V_{T_2}(L(C.b)))
14:       T_cmp' ← T_2'.Compress(V_{T_2}(L(C.b)))
15:       ans ← min(ans, Bipartite(T_cmp, T_cmp', L(C.b)))
16:     return ans
```

Since this algorithm creates many copies of T_2, we must ensure that we can still identify and locate a desired vertex given its label. One simple way to achieve this is to build a static hashtable alongside each copy of T_2 that maps vertex labels to the instance of that vertex in that copy.

An ingredient that we need to achieve low depth is an efficient way to update the weights in T_2 when adding weights to a collection of paths. Although RC trees support batch-adding weights to paths, the standard algorithm does not meet our cost requirements. This is easy to achieve in linear work and $O(\log n)$ depth by propagating the total weight of all updates up the clusters, and then propagating back down the tree, the weight of all updates that are descendants of the current cluster. See the full version [4] for more details. It remains to analyze the cost of the Bipartite procedure.

Theorem 5.4. *A bipartite problem of size m can be solved in $O(m \log m)$ work and $O(\log^3 m)$ depth w.h.p.*

Proof. First, since all recursive calls are made in parallel and the recursion is on the clusters of T_1, the number of levels of recursion is $O(\log m)$ w.h.p. We will show that the algorithm performs $O(m)$ work in total at each level, in $O(\log^2 m)$ depth w.h.p. Observe first that at each level of recursion, the edges L for each call are a disjoint partition of the non-tree edges, since each recursive call takes a disjoint subset. We will now argue that each call does work proportional to $|L|$. Since T_2 and T_2' are both compressed with respect to L, their size is proportional to $|L|$. AddPaths takes linear work in the size of T_2 and $O(\log m)$ depth, and hence takes $O(|L|)$ work and $O(\log m)$ depth. Compress(K) takes $O(|K| \log(1 + |T_2|/|K|)) \leq O(|K| + |T_2|)$ work and $O(\log^2 m)$ depth w.h.p.. Since compression is with respect to some subset of L, all of the compress operations take $O(|L|)$ work and $O(\log^2 m)$ depth w.h.p. In total, this is $O(|L|)$ work in $O(\log^2 m)$ depth w.h.p. at each level for each call. Since the Ls at each level are a disjoint partition of the non-tree edges, the total work per level is $O(m)$ w.h.p., and hence the desired bounds follow. □

Since there are $O(n)$ bipartite problems of total size $O(m)$, solving them all in parallel yields the following, which, when combined with Theorem 5.1, proves Theorem 1.3.

Theorem 5.5. *Given a weighted, undirected graph G and a rooted spanning tree T, the minimum 2-respecting cut of G with respect to T such that the cut edges are independent can be computed in $O(m \log n)$ work and $O(\log^3 n)$ depth w.h.p.*

Combining Theorem 4.1 with Theorem 1.3 on each of the $O(\log n)$ trees in parallel proves Theorem 1.1.

6 CONCLUSION

We present a randomized $O(m \log^2 n)$ work, $O(\log^3 n)$ depth parallel algorithm for minimum cut. It is the first parallel minimum cut algorithm to match the work bound of the best sequential algorithm, making it work efficient. Finding a faster parallel algorithm for minimum cut would therefore entail finding a faster sequential algorithm. It remains an open problem to find a deterministic algorithm for minimum cut, even a sequential one, that runs in $O(m \operatorname{polylog} n)$ time.

ACKNOWLEDGMENTS

We thank the anonymous referees for their comments and suggestions, and Phil Gibbons for his feedback on the manuscript. We thank Ticha Sethapakdi for helping with the figures. This research was supported by NSF grants CCF-1901381, CCF-1910030, and CCF-1919223.

REFERENCES

[1] Umut A Acar, Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, and Sam Westrick. 2020. Parallel Batch-dynamic Trees via Change Propagation. In *European Symposium on Algorithms (ESA)*.

[2] Umut A Acar, Guy E. Blelloch, and Jorge L Vittes. 2005. An experimental analysis of change propagation in dynamic trees. In *Algorithm Engineering and Experiments (ALENEX)*.

[3] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2005. Maintaining information in fully dynamic trees with top trees. *ACM Trans. on Algorithms* 1, 2 (2005), 243–264.

[4] Daniel Anderson and Guy E. Blelloch. 2021. Parallel Minimum Cuts in $O(m \log^2 n)$ Work and Low Depth. (2021). arXiv:2102.05301 [cs.DS]

[5] Daniel Anderson, Guy E. Blelloch, and Kanat Tangwongsan. 2020. Work-efficient batch-incremental minimum spanning trees with applications to the sliding window model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[6] Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (March 1996).

[7] 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)*.

[8] Joseph Cheriyan, Ming-Yang Kao, and Ramakrishna Thurimella. 1993. Scan-first search and sparse certificates: an improved parallel algorithm for k-vertex connectivity. *SIAM J. Comput.* 22, 1 (1993), 157–174.

[9] Richard Cole, Philip N Klein, and Robert E Tarjan. 1996. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[10] Martín Farach-Colton and Meng-Tsung Tsai. 2015. Exact sublinear binomial sampling. *Algorithmica* 73, 4 (2015), 637–651.

[11] Harold N Gabow. 1995. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.* 50, 2 (1995), 259–273.

[12] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. 2020. Minimum Cut in $O(m \log^2 n)$ Time. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*.

[13] H. Gazit, Gary L. Miller, and ShangHua Teng. 1988. Optimal Tree Contraction in the EREW Model. In *Concurrent Computations*. Plenum Press, 139–156.

[14] Barbara Geissmann and Lukas Gianinazzi. 2018. Parallel minimum cuts in near-linear work and low depth. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[15] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. 2020. Faster algorithms for edge connectivity via random 2-out contractions. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[16] Ralph E Gomory and Tien Chung Hu. 1961. Multi-terminal network flows. *J. Soc. Indust. Appl. Math.* 9, 4 (1961), 551–570.

[17] JX Hao and James B Orlin. 1994. A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms* 17, 3 (1994), 424–446.

[18] Dov Harel and Robert Endre Tarjan. 1984. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing* 13, 2 (1984), 338–355.

[19] Lorenz Hübschle-Schneider and Peter Sanders. 2019. Parallel Weighted Random Sampling. (2019). arXiv:1903.00227 [cs.DS]

[20] David Karger. 1995. *Random sampling in graph optimization problems.* Ph.D. Dissertation. Stanford University.

[21] David R Karger. 1993. Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm.. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[22] David R Karger. 1999. Random sampling in cut, flow, and network design problems. *Math. Oper. Res.* 24, 2 (1999), 383–413.

[23] David R Karger. 2000. Minimum cuts in near-linear time. *J. ACM* 47, 1 (2000), 46–76.

[24] David R Karger and Rajeev Motwani. 1994. Derandomization through approximation: An NC algorithm for minimum cuts. In *ACM Symposium on Theory of Computing (STOC)*.

[25] David R Karger and Clifford Stein. 1996. A new approach to the minimum cut problem. *J. ACM* 43, 4 (1996), 601–640.

[26] Jason Li. 2021. Deterministic Mincut in Almost-Linear Time. In *ACM Symposium on Theory of Computing (STOC)*. (To appear).

[27] Antonio Molina Lovett and Bryce Sandlund. 2020. A simple algorithm for minimum cuts in near-linear time. In *Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*.

[28] David W Matula. 1993. A linear time 2+ ε approximation algorithm for edge connectivity. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[29] Gary L. Miller and John H. Reif. 1989. Parallel Tree Contraction Part 1: Fundamentals. In *Randomness and Computation*, Vol. 5. 47–72.

[30] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Discrete Math.* 5, 1 (1992), 54–66.

[31] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica* 7, 1-6 (1992), 583–596.

[32] C St JA Nash-Williams. 1961. Edge-disjoint spanning trees of finite graphs. *J. London Math. Soc.* 1, 1 (1961), 445–450.

[33] Serge A Plotkin, David B Shmoys, and Éva Tardos. 1995. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.* 20, 2 (1995), 257–301.

[34] Baruch Schieber and Uzi Vishkin. 1988. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.* 17, 6 (1988), 1253–1262.

[35] Daniel D Sleator and Robert Endre Tarjan. 1983. A data structure for dynamic trees. *J. Comput. Syst. Sci.* 26, 3 (1983), 362–391.