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

Daniel Anderson
Carnegie Mellon University
dlanders@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Abstract

We present an $O(m \log^2(n))$ work, $O(\text{polylog}(n))$ depth parallel algorithm for minimum cut. This algorithm matches the work of a recent sequential algorithm by Gawrychowski, Mozes, and Weimann [ICALP'20, (2020), 57:1-57:15], and improves on the previously best known parallel algorithm by Geissmann and Gianinazzi [SPAA'18, (2018), pp. 1–11] 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 for dynamic trees that solves mixed batches of queries and weight updates in low depth. 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 a work-efficient parallel algorithm for solving the minimum 2-respecting cut problem.

1 Introduction

The minimum cut problem is one of the classic problems in graph theory and algorithms. The problem is to find, given an undirected weighted graph $G = (V, E)$, a 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 [15, 16]. Several algorithms followed which were based on edge contraction [30, 31, 20, 25]. Karger first observed that tree packings [32] can be used to find minimum cuts [22]. In particular, for a graph with n vertices and m edges Karger showed how 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. The second ingredient is then an $O(m \log^2(n))$ time algorithm to find the so-called minimum *2-respecting* cut of each of these spanning trees, yielding an $O(m \log^3(n))$ time algorithm for minimum cut. Karger [22] also describes a parallel algorithm for finding the minimum 2-respecting cut in $O(n^2)$ work in $O(\log^3(n))$ 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 [13] 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 simultaneously in low depth. Their algorithm performs just a factor of $O(\log(n))$ additional work than Karger's sequential algorithm, but substantially improves on the work of Karger's parallel algorithm.

Even more recently, a breakthrough from Gawrychowski, Mozes, and Weimann [11] gave an $O(m \log^2(n))$ algorithm for minimum cuts. Their algorithm is also based on Karger's algorithm, and 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.

To generate the $O(\log n)$ spanning trees, Karger used a combination of random sampling [20] and a modification of a tree packing algorithm of Gabow [10]. The random sampling requires a constant approximation to the minimum cut, which is the most challenging part to parallelize. Karger and Motwani give a parallel algorithm for approximating the cut that runs with $O(m^2/n)$ work in polylogarithmic depth [24].

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

Theorem 1. *Given a weighted, undirected graph G, there exists a parallel algorithm that, with high probability, computes the minimum cut of G 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 and in low depth. This algorithm is based on parallel tree contraction [28] and 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 2. *Given a constant-degree RC tree of size n, and a simple operation set, after $O(n)$ work and $O(\log n)$ depth preprocessing, every following batch of k operations from the operation-set,*

can be processed in $O(k \log(k+n))$ 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 previous results by Geissmann et. al., 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 preserving an approximate minimum cut.

Lastly, we show how to solve the minimum 2-respecting cut problem work-efficiently in parallel, using a combination of our new parallel dynamic tree algorithms combined with the use of RC trees to efficiently perform a divide-and-conquer search over the edges of the 2-constraining trees (Section 5)

Theorem 3. *There exists an algorithm that given a weighted, undirected graph G and a rooted spanning tree T, computes the minimum 2-respecting cut of G with respect to T, in $O(m \log(n))$ work and $O(\text{polylog}(n))$ depth w.h.p.*

Application to the unweighted problem. The unweighted minimum cut problem, or edge connectivity problem was recently improved by Ghafarri, Nowicki, and Thorup [14] who give an $O(m \log(n) + n \log^4(n))$ work and $O(\log^3(n))$ 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 for unweighted minimum cut 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-style parallelism. A procedure can *fork* off 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 [5]. The model can work-efficiently cross simulate the classic CRCW PRAM model [5], and the more recent Binary Forking model [6] 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 [21], 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. We can assume that 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 bound. Note that it is not always possible to convert a Monte Carlo algorithm into a Las Vegas one, unless a fast algorithm for verifying a solution is available, which is not the case for minimum cuts.

Tree contraction. Parallel tree contraction is a framework for producing dynamic tree algorithms, introduced by Miller and Reif [29]. Tree contraction works by performing a sequence of rounds, each applying two operations, rake and compress, in parallel across every vertex of the tree, to produce a sequence of smaller (contracted) trees. 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. For a rooted tree the root is never removed, and is the final surviving vertex. The technique of Miller and Reif produces a sequence of $O(\log(n))$ trees w.h.p., with $O(n)$ vertices in total across all of the contracted trees w.h.p. Their algorithm applies to bounded-degree trees, but arbitrary-degree trees be handled by converting them into equivalent bounded-degree trees.

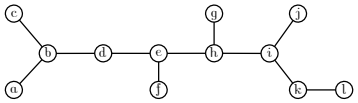
A powerful application of tree contraction is that it can be used to produce a recursive clustering of the given tree with attractive properties. From the resulting tree contraction, a recursive clustering can be produced that consists of $O(n)$ clusters with recursive height $O(\log(n))$ w.h.p. Such a clustering can be represented as a so-called *rake-compress tree* (RC tree) [2].

Rake-compress trees. The RC tree 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. Figure 1 illustrates a recursive clustering, and its corresponding RC tree. 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 always the disjoint union of its children (subclusters). The leaf clusters of the RC tree are the vertices and edges of the original tree. Note that all non-leaf clusters have exactly one vertex (leaf) cluster as a child. This vertex is that cluster's *representative* vertex. Clusters 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.

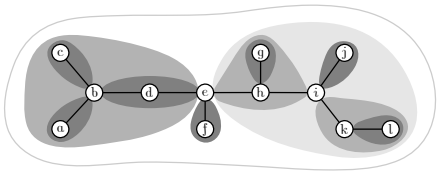
In this paper we will be considering rooted trees. In this case the root of the tree is also the representative of the top level nullary cluster of the RC-tree. All binary clusters have a binary subcluster whose path is above the representative vertex, which we will refer to as the *top cluster*, and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

RC trees are similar to top trees [3], which are also based on a recursive clustering strategy. Both data structures support a wide variety of queries. Compared to top trees, however, RC trees are somewhat simpler (fewer cases) and, importantly for us, it is well understood how to construct them in parallel [29, 12]. We refer the reader to [1] and [2] for a more in-depth explanation of RC trees and their properties.

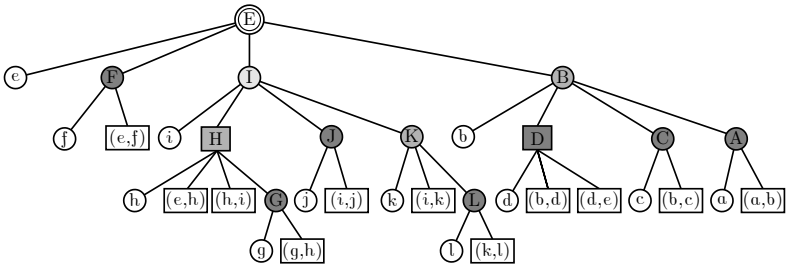
Compressed path trees. For a weighted (unrooted) tree T and a set of *marked* vertices $V \subset T$, the compressed path tree is a minimal tree T_c on V and some additional "steiner vertices" from T such that for every pair $(u, v) \in V$, the lightest edge on the path from u to v is the same in T and T_c. Alternatively, the compressed path tree is the tree T with all unmarked vertices of degree less than



(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. (Non-base) unary clusters are shown as circles, binary clusters as rectangles, and the finalize (nullary) cluster at the root with two concentric circles. The base clusters (the leaves) 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].

three spliced out. It is not hard to show that T_c has size less than $2|V|$. Compressed path trees are described in [4], 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. [11] 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 [22] is based on the notion of *k-respecting cuts*. Given a weighted, undirected graph G and a spanning tree T, a cut of G k-respects T if at most k edges of T cross the cut. 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 spanning trees with weights assigned to the edges such that for each edge in G, its total weight in all of the spanning trees is no more than its weight in G. Since the underlying tree packing algorithms used by Karger have running time proportional to the size of the minimum cut, random sampling is used to produce a sparsified graph, or *skeleton*, such that the resulting tree packing still has the desired property w.h.p. This allows the tree packing algorithms to run sufficiently fast. Given the sparsified graph, Karger gives two algorithms for producing tree packings of size $O(\log(n))$ such that w.h.p., the minimum cut 2-respects one of them. The first approach uses a tree packing algorithm of Gabow [10]. 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 one 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. Here 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 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 [36, 3, 2] 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.

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 weights for the subtree rooted at v

Algorithm 1 The SUBTREESUM query. The query starts at the leaf for v and goes up the RC tree keeping track of the total weight on the bottom side of v. Note that x will never be a unary cluster, so if not the representative or top subcluster of p (Line 5), it is the bottom subcluster with nothing below it in this cluster.

```
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. Leaves in the RC tree start with zero weight. This satisfies the first condition since the sums take constant time. An addWeight(v, w) adds weight w to the vertex v (which is a leaf in the 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, which 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 the function requires constant space, satisfying the second condition. The operations therefore has a simple RC implementation.

We are interested in implementing batches of of operations from an an operation-set on trees with a simple RC implementation. In particular, we prove Theorem 2.

Proof of Theorem 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], or in the binary forking model [6], or deterministically [12]. 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 position.
2. Collect all operations by their associated leaf, and sort within each leaf by timestamp. This can be implemented with a single sort.
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.
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 operation we have its value, and for each query we also have its partial evaluation based on the value. We prepend the initial value. We call this the *operation list*. An operation list is *non-trivial* if it has more than just the initial value.
5. Sequentially for each level of the cluster 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 by timestamp.
   (b) Calculate for each element the latest value of each child at or before the timestamp. This can be implemented by prefix sums.

(c) For each element in the list 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.

Note that 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, adjacent duplicates that share the same parent can be combined.

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 work 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 $O(k)$. □

Note that we could maintain operation lists at each cluster for all operations on the source tree (along with links to the child nodes) across all batches. This woud allow arbitrary queries back in time in $O(\log n)$ work per query. However it would not satisfy the desired space bounds.

3.1 Path Updates and Path/Subtree Queries

We now consider implementing mixed operations consisting of updating paths, and querying both paths and subtres. We will use these in batch in Sections 3.2 and 5. In particular we wish to maintain, given a rooted tree $T = (V, E)$ a weight $w(e)$ for each $e \in E$, a data structure that supports the following operations.

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

We also consider AddPath'(v, x), which adds x 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 from these with a constant number of calls given the LCA in the original tree for AddPath and in the RC tree for QueryPath.

The interface can be implemented in $O(\log(n))$ time by using top trees [3]. Here we describe a simple RC implementation to allow efficient batching.

Lemma 1. *The* AddPath*',* QuerySubtree*,* QueryPath*', and* QueryEdge *operations on bounded degree trees can be supported with a simple RC implementation.*

Algorithm 2 A simple RC implementation of AddPath and QuerySubtree.

procedure $f_{\text{UNARY}}(w_v, (m_t, l_t, w_t), U)$
  $w' \leftarrow w_v + \sum_{(m,w)\in U} w$
  $m_u \leftarrow \min_{(m,w)\in U} m$
  return $(\min(m_t, l_t + w', m_u), w_t + w')$
procedure $f_{\text{BINARY}}(w_v, (m_t, l_t, w_t), (m_b, l_b, w_b), U)$
  $w' \leftarrow w_v + w_b + \sum_{(m,w)\in U} w$
  $m_u \leftarrow \min_{(m,w)\in U} m$
  return $(\min(m_t, m_b, m_u), \min(l_t + w', l_b), w_t + w')$
procedure AddPath'(v, w)
  $v{\rightarrow}\text{value} \leftarrow v{\rightarrow}\text{value} + w$
  Reevaluate the $f(\cdot)$ on path to root.
procedure QuerySubtree(v)
  $w \leftarrow \infty$; $l \leftarrow \infty$
  $x \leftarrow v$; $p \leftarrow x{\rightarrow}p$
  while binary p **do**
    if $(x = p{\rightarrow}t)$ or $(x = p{\rightarrow}v)$ **then**
      $w' \leftarrow p{\rightarrow}b{\rightarrow}w + p{\rightarrow}v{\rightarrow}w + \sum_{u\in p\not\rightarrow U} u{\rightarrow}w$
      $l \leftarrow \min(l + w', p{\rightarrow}b{\rightarrow}l)$
      $m \leftarrow \min(m, p{\rightarrow}b{\rightarrow}m, \min_{u\in p\not\rightarrow U} u{\rightarrow}m)$
    $x \leftarrow p$; $p \leftarrow x{\rightarrow}p$
  $w' \leftarrow p{\rightarrow}v{\rightarrow}w + \sum_{u\in p\not\rightarrow U} u{\rightarrow}w$
  return $\min(l + w', m, \min_{u\in p\not\rightarrow U} u{\rightarrow}m)$

Proof. Our simple RC implementation for combining values, AddPath, and QuerySubtree is given in Algorithm 2. The other two operations can be found in Appendix E. The value of each vertex (leaf) in the cluster is the the total weight added to that vertex by AddPath. The value for each unary cluster consists of: m, the minimum weight edge in the cluster, and w, the total weigh 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 (which are binary clusters) 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., $\min(m_t, l_t + w', m_y)$). This is correct since all paths from those clusters to the root go through the binary edge, 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 cluster vertex goes through them. The minimum weight is therefore correct. The total AddPath weight is trivially 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'). 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.

The QuerySubtree accumulates the appropriate minimum weights within a subtree as it goes up the RC tree. 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) 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 1. *Given a bounded-degree tree of size n, any sequence of m* AddPath, QuerySubtree, QueryPath, *and* QueryEdge *operations can be evaluated in $O(n + m \log n)$ work, $O(\log^2(n + m))$ depth and $O(n + m)$ space.*

Proof. The LCAs required to convert AddPath to AddPath' and QueryPath to QueryPath' can be computed in $O(n)$ work, $O(\log(n))$ depth, and $O(n)$ space [35] . The rest follows from Theorem 2 and Lemma 1. □

3.2 Improving Previous Results

Using our batched mixed operations, 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 (GG) algorithm [13], and we can parallelize Lovett and Sandlund's (LS) algorithm [26].

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 set in $O(m \log n)$ work and $O(\log n)$ depth [13, Lemma 12]. 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 1 improves their results to $O(m \log^2 n)$ work.

Lovett and Sandlund significantly simplify Kargers' 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 [13, Lemma 7], and the results of Corollary 1 again gives an $O(m \log^2 n)$ work, $O(\log^2 n)$ 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 a constant-factor approximation to the 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

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. To save work, we can use the trick mentioned by Gawrychowski et al. [11]. Since the sampled graph is a multigraph sampled from a graph with m edges, the MST algorithm need only know 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 and $O(\log(n))$ depth MST algorithm [8] 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 a constant-factor approximation to the minimum cut. In the sequential setting, Matula's algorithm can be used, which runs in linear time on unweighted graphs, and can be extended to weighted graphs to run 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, which is far too much for our purposes. In Appendix A, we derive a faster version of the approximation algorithm that runs in $O(m \log^2(n))$ work and $O(\log^3(n))$ depth. Taking all of this together, we have the following theorem.

Theorem 4. *Given a weighted, undirected graph, a set of $O(\log(n))$ spanning trees can be produced in $O(m \log^2(n))$ work and $O(\log^3(n))$ depth such that w.h.p., the minimum cut two respects at least one of them*

5 Finding Minimum 2-respecting Cuts

We are given a graph G and a set of $O(\log(n))$ trees such that, w.h.p., the minimum cut of G 2-respects at least one of the trees. In this section, we will give an algorithm that finds the minimum 2-respecting cut of G with respect to a tree T in $O(m \log(n))$ work and $O(\text{polylog}(n))$ depth.

Our faster $O(m \log(n))$ work 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. We assume we are given an undirected weighted graph $G = (V, E)$ with maximum degree three. Note that any arbitrary degree graph can easily be *ternarized* by replacing high-degree vertices with paths of infinite weight edges, resulting in a graph of maximum degree three with the same minimum cut, and only a constant-factor larger size.

5.1 Descendant edges

We now present our algorithm for minimum 2-respecting cut 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 binary since G is a connected graph with bounded degree three.

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 $u - v$ path in T contains the edge e. Then the weight of the cut induced by a pair of edges $\{e, e'\}$ 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 the expression

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

To do so, it traverses T from the root while maintaining dynamic 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 $e' \in$ 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 mixed-batch evaluation algorithm (Corollary 1).

The operation sequence can be generated as follows. First, the weights $w(F_e)$ for each edge can be computed using the batch evaluation algorithm (Corollary 1) 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 [35] 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 1, we arrive at the following result.

Theorem 5. *There exists an algorithm that given a weighted, undirected graph G and a rooted spanning tree T, computes the minimum 2-respecting cut of G with respect to T such that one of the cut edges is a descendant of the other 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. [11], 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 [17], 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, so we never have to propagate anything up the original possibly unbalanced tree.

Definition 2 (The bipartite problem)**.** *Given two weighted rooted trees T_1 and T_2 and a set of weighted edges that cross from one tree to the other, i.e. $L = \{(u, v) : u \in T_1, v \in T_2\}$, 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 underneath e_1 and v_2 is in the subtree underneath e_2. The size of a bipartite problem is the size of L plus the sizes of T_1 and T_2.*

Gawrychowski et al. observe that if T_1 and T_2 are disjoint subtrees of T, then, assigning weights of $-2w(F_e)$ to each edge, 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 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 $e \in E(G), e \notin T$, 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.

Lemma 2. *There exists an algorithm that can generate the bipartite problems in $O(m \log(n))$ work and $O(\log^2(n))$ depth w.h.p.*

Proof. The edge weights can be computed using the batch evaluation algorithm in $O(m \log(n))$ work and $O(\log^2(n))$ depth in the same way as before. LCAs can be computed using the parallel LCA algorithm of Schieber and Vishkin [35] 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 noting 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). See Figure 2. A unary cluster is the disjoint union of exactly one binary cluster at the top, zero to two unary clusters at the bottom, and one leaf cluster corresponding to the representative vertex joining them in the middle. A binary cluster is the disjoint union of two binary clusters, one on top and one below; zero or one unary clusters at the bottom; and the leaf cluster corresponding to the representative vertex joining them in the middle. All non-leaf clusters, except the root cluster are either unary or binary clusters. 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

(a) A unary cluster consisting of one binary subcluster and two unary subclusters

(b) A binary cluster consisting of two binary subclusters and one unary subcluster

Figure 2: Unary clusters and binary clusters

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 [4], 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 copy of the tree.

Remark 1 (Identifying vertices)**.** *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. Since our bounds are already randomized, using hashing is okay.*

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 using ideas similar to standard treefix sum algorithms (See Appendix D for details). It remains to show that the BIPARTITE procedure runs in low work and depth.

Theorem 6. *A bipartite problem of size m can be solved in $O(m \log(m))$ work and $O(\log^3(m))$ depth w.h.p.*

Algorithm 3 Parallel bipartite problem algorithm

```
1: procedure BIPARTITE(C : Cluster, T_2 : Tree, T'_2 : Tree, L : Edge list)
2:   if C = {e} then
3:     return w(e) + LIGHTESTEDGE(T'_2)
4:   else
5:     local T_cmp ← T_2.COMPRESS(V_{T_2}(L(C→t)))
6:     local T''_2 ← T'_2.ADDPATHS(L(C) \ L(C→t))
7:     local T''_cmp ← T''_2.COMPRESS(V_{T_2}(L(C→t)))
8:     local ans ← BIPARTITE(C→t, T_cmp, T''_cmp, L(C→t))
9:     for each cluster C' in C→U do
10:      local 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:      local T_cmp ← T_2.COMPRESS(V_{T_2}(L(C→b)))
14:      local 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
```

Proof. First, since all recursive calls are made in parallel and the recursion is on the clusters of T_1, the maximum 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 can be implemented in linear work in the size of T_2 and $O(\log(m))$ depth (Appendix D), 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 gives us the following theorem, which, when combined with Theorem 5, proves Theorem 3.

Theorem 7. *There exists an algorithm that given a weighted, undirected graph G and a rooted spanning tree T, computes the minimum 2-respecting cut of G with respect to T such that the cut edges are independent, in $O(m\log(n))$ work and $O(\log^3(n))$ depth w.h.p.*

Combining Theorem 3 with Theorem 4 concludes our main result (Theorem 1).

6 Conclusion

We present the first work-efficient algorithm for minimum cuts that runs in low depth. That is, the first highly parallel algorithm that performs no more work than the best sequential algorithm. Since our algorithm is work efficient, finding a faster parallel algorithm would entail finding a faster sequential algorithm. Our algorithm is Monte Carlo and it runs in $O(m\log^2(n))$ work and $O(\text{polylog}(n))$ depth. It remains an open problem to find a deterministic algorithm, even a sequential one, that runs in $O(m\,\text{polylog}(n))$ time.

Acknowledgments

This work was supported in part by NSF grants CCF-1408940 and CCF-1629444.

References

[1] U. A. Acar, D. Anderson, G. E. Blelloch, L. Dhulipala, and S. Westrick. Parallel batch-dynamic trees via change propagation. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

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

[3] S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. on Algorithms*, 1(2):243–264, 2005.

[4] D. Anderson, G. E. Blelloch, and K. Tangwongsan. Work-efficient batch-incremental minimum spanning trees with applications to the sliding window model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020. (To appear).

[5] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3), Mar. 1996.

[6] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[7] J. Cheriyan, M.-Y. Kao, and R. Thurimella. Scan-first search and sparse certificates: an improved parallel algorithm for k-vertex connectivity. *SIAM Journal on Computing*, 22(1):157–174, 1993.

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

[9] M. Farach-Colton and M.-T. Tsai. Exact sublinear binomial sampling. *Algorithmica*, 73(4):637–651, 2015.

[10] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, 1995.

[11] P. Gawrychowski, S. Mozes, and O. Weimann. Minimum cut in $O(m \log^2 n)$ time. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*, 2020. (to appear).

[12] H. Gazit, G. L. Miller, and S. Teng. Optimal tree contraction in an EREW model. In S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156, New York, 1988. Plenum Press. Princeton Workshop on Algorithms, Architecture and Technology Issues for Models of Concurrent Computation.

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

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

[15] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

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

[17] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing*, 13(2):338–355, 1984.

[18] L. Hübschle-Schneider and P. Sanders. Parallel weighted random sampling, 2019.

[19] D. Karger. *Random sampling in graph optimization problems*. PhD thesis, stanford university, 1995.

[20] D. R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In *SODA*, volume 93, pages 21–30, 1993.

[21] D. R. Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2):383–413, 1999.

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

[23] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995.

[24] D. R. Karger and R. Motwani. Derandomization through approximation: An nc algorithm for minimum cuts. In *ACM Symposium on Theory of Computing (STOC)*, pages 497–506, 1994.

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

[26] A. M. Lovett and B. Sandlund. A simple algorithm for minimum cuts in near-linear time. *arXiv preprint arXiv:1908.11829*, 2019.

[27] D. W. Matula. A linear time 2+ ε approximation algorithm for edge connectivity. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 500–504. Society for Industrial and Applied Mathematics, 1993.

[28] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. Technical report, HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB, 1985.

[29] G. L. Miller and J. H. Reif. Parallel tree contraction part 1: Fundamentals. In *Randomness and Computation*, volume 5, pages 47–72, 1989.

[30] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.

[31] H. Nagamochi and T. Ibaraki. A linear-time algorithm for finding a sparsek-connected spanning subgraph of ak-connected graph. *Algorithmica*, 7(1-6):583–596, 1992.

[32] C. S. J. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 1(1):445–450, 1961.

[33] S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20(2):257–301, 1995.

[34] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. on Computing*, 18(3), 1989.

[35] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[36] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. of computer and system sciences*, 26(3):362–391, 1983.

A A Parallel Constant-Factor Minimum Cut Approximation

Step one of Karger's procedure for producing a tree packing is to compute a constant-factor approximation to the minimum cut, which is then used to derive the sampling probability for constructing a sparse skeleton. In this section, we will derive an algorithm for a constant-factor approximate minimum cut that runs in $O(m \log^2(n))$ work and $O(\text{polylog}(n))$ depth. Karger and Motwani [24] give an algorithm that runs in $O(m^2/n)$ work and $O(\text{polylog}(n))$ depth. We achieve our bounds by improving Karger's algorithm and speeding up several of the components. Specifically, we use the following combination of ideas, new and old.

1. We extend a k-approximation algorithm of Karger [20] to work in parallel, allowing us to produce an $O(\log(n))$-approximate minimum cut in low work and depth
2. The $\log(n)$-approximate minimum cut allows us to make $O(\log\log(n))$ guesses of the minimum cut such that at least one of them is a constant-factor approximation
3. 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.
4. We show that the parallel sparse k-certificate algorithm of Cheriyan, Kao, and Thurimella [7] for unweighted graphs can be modified to run on weighted graphs
5. We show that Karger and Motwani's parallelization of Matula's algorithm can be modified to run on weighted graphs

We will use the following result due to Karger.

Definition 3 (p-skeleton of a graph)**.** *Given an unweighted graph G and a probability p, the* skeleton $G(p)$ *consists of the vertices of G and a random subset of the edges of G, each sampled with probability p.*

Lemma 3 (Karger [19])**.** *With high probability, if $G(p)$ is constructed and has minimum cut $\hat{c} = \Omega(\log(n)/\varepsilon^2)$ for $\varepsilon \leq 1$, then the minimum cut in G is $(1 \pm \varepsilon)\hat{c}/p$.*

Parallelising the k-approximation algorithm. 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 [20]. Karger's algorithm 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. This relies on the following Lemma.

Lemma 4 (Karger [20])**.** *Given a weighted graph with minimum cut c, with probability $n^{-2/k}$, the meta-vertex with minimum incident weight encountered during a single trial of the contraction algorithm implies a cut of weight at most kc.*

Running $O(n^{2/k} \log(n))$ rounds yields a cut of size at most kc w.h.p. Here we show how to parallelize Karger's algorithm using batched mixed operations yielding the following result:

Lemma 5. *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 polylogarithmic depth.*

Karger shows how to parallelize picking the (weighted) random permutation of the edges with $O(m \log^2(n))$ work. It can easily slightly modified to improve the bounds by a logarithmic factor as follows. The algorithm selects the edges by running a prefix sum over the edge weights. Assuming a total weight of W, it then picks m random integers up to W, and for each uses binary search on the result of the prefix sum to pick an edge. This process, however, might end up picking only the heaviest edges. Karger shows that by removing those edges the total weight W decreases by a constant factor, with high probability. Since the edges can be preprocessed to be polynomial in n (see below), repeating for $\log(n)$ rounds the algorithm will select all edges in the appropriate weighted random order. Each round takes $O(m \log(n))$ work for a total of $O(m \log^2(n))$ work. However, by replacing the binary search with a sort of the random integers and merge into the the result of the prefix sum yields an $O(m \log(n))$ work randomized algorithm. In particular m random numbers uniformly distributed over a range can be sorted in $O(m)$ work and $O(\log(n))$ span by first determining for each number which of m evenly distributed buckets within the range it is in, then sorting by bucket using an integer sort [34] and finally sorting within buckets.

The more interesting part to parallelize is identifying the component with smallest incident weight during the contraction process. Identifying the edges that are contracted is easy using a minimum spanning tree over the position on which the edge is selected, but keeping track of the smallest incident weight of a component is somewhat trickier. To achieve this, we use our parallel batch mixed operations framework from Section 3. In Appendix B, we show that that the following operations have a simple RC implementation and therefore can be applied in batch.

- 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, we can simulate the contraction process, and determine the minimum incident weight of a component as follows:

1. Compute an MST with respect to the random edge ordering, where a heavier weight indicates that an edge contracts later
2. For each edge $(u, v) \in G$, determine the heaviest edge in the MST on the unique (u, v) path
3. Construct a vertex-weighted tree from the MST, where the weights are the total incident weight on each vertex in G. For each edge (u, v) in the MST in contraction order:
   - Determine the set of edges in G such that (u, v) is the heaviest edge on its MST path. For each such edge identified, SubtractWeight from each of its endpoints by the weight of the edge
   - Perform JoinEdge on the edge (u, v)
   - Perform QueryWeight on the vertex u

Observe that the weight of a component at the point in time when it is queried is precisely the total weight of incident edges (again, not including internal edges). Taking the minimum over the initial degrees and all query results therefore yields the desired answer.

Step 1 takes $O(m \log(n))$ work and $O(\log^2(n))$ depth to compute the random edge permutation using Karger's technique [20], and $O(m)$ work and $O(\log(n))$ depth to run a parallel MST algorithm [23]. Step 2 takes $O(m \log(n))$ work and $O(\log(n))$ depth using RC trees [2, 1], and Step 3 takes $O(m \log(n))$ work and $O(\log^2(n))$ depth using our batch evaluation framework (Theorem 2).

Based on Lemma 4, trying $O(n^{2/k}\log(n))$ random contractions yields the result of Lemma 5. Setting $k = \log(n)$ then 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 [19] 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 6. *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' is a constant-approximate minimum cut in G.*

Proof. Let G be the input graph and suppose that the true value of the minimum cut is c. First, we use Lemma 5 to obtain a $O(\log(n))$-approximate minimum cut, whose value we denote by $\tilde{c}$ ($c \leq \tilde{c} \leq c\log(n)$). We can contract all edges of the graph with weight greater than $\tilde{c}$ since they can not appear in the minimum cut. Let $s = \tilde{c}/(2m\log(n))$. We delete (not contract) all edges with weight less than s. Since there are at most m edges in any cut, this at most affects the value of a cut by $sm = \tilde{c}/(2\log(n)) \leq c/2$. Therefore the minimum cut in this graph is still a constant factor approximation to the minimum cut in G.

Next, scale all remaining edge weights down by the factor s, rounding down. All edge weights are now integers in the range $[1, 2m\log(n)]$. This is the transformed graph G'. It remains to argue that the value of the minimum cut is a constant-factor approximation. First, note that the scaling process preserves the order of cut values, and hence the true minimum cut in G has the same value in G' as the minimum cut in G'. Consider any cut in G', and scale the weights of the edges back up by a factor s. This introduces a rounding error of at most s per edge. Since any cut has at most m edges, the total rounding error is at most $sm \leq c/2$. Therefore the value of the minimum cut in G' is a constant factor approximation to the value of the minimum cut in G. □

Lastly, observe that this transformation can easily be performed in parallel by using a work-efficient connected components algorithm to perform the edge contractions, as is standard (see e.g. [25]).

Sampling the skeleton from a weighted graph. Note that by definition, the p-skeleton of a graph has $O(pm)$ edges in expectation. For a weighted graph, the p-skeleton is defined as the p-skeleton of the corresponding unweighted multigraph in which an edge of weight w is replaced by w parallel multiedges. The p-skeleton of a weighted graph therefore has $O(pW)$ edges in expectation, where W is the total weight in the graph. Karger gives an algorithm for generating a p-skeleton in $O(pW\log(m))$ work, which relies on performing $O(pW)$ independent random samples with probabilities proportional to the weight of each edge, each of which takes $O(\log(m))$ amortized time. In Karger's algorithm, given a guess of the minimum cut c, he computes p-skeletons for $p = \Theta(\log(n)/c)$, which results in a skeleton of $O(m\log(n))$ edges, and hence takes $O(m\log^2(n))$ work to compute.

Our algorithm instead does the following. For each edge e in the graph, sample a binomial random variable $x \sim B(w(e), p)$. The skeleton then contains the edge e with weight x (conceptually, x unweighted copies of the multiedge e). This results in the same graph as if sampled using Karger's technique. In Appendix C, we show how to use recent results on sampling binomial random variables 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 $n' \leq N$ after $O(N^{1/2+\varepsilon})$ work preprocessing. Since we can preprocess the graph to have edge weights at most $O(m\log(n))$ (Lemma 6), this is no more than $O(m)$ work in preprocessing.

At first glance, this does not improve on Karger's bounds, since we need to perform $O(\log^2(n))$ work per edge when sampling from $B(n,p)$. However, we use the fact that only the first sample of the graph needs to be this expensive. In Karger's algorithm, and by extension, our algorithm, subsequent samples always take p as exactly half of the value of p from last iteration, and hence we can use subsampling to 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 *k-connectivity certificate* of a graph $G=(V,E)$ is a graph $G'=(V,E'\subset E)$ such that every cut in G of weight at most k has the same weight in G'. In other words, a k-connectivity certificate is a subgraph that preserves cuts of weight up to k. A k-connectivity certificate is called *sparse* if it has $O(kn)$ edges.

Cheriyan, Kao, and Thurimella [7] introduce a parallel graph search called *scan-first search*, which they show can be used to generate k-connectivity certificates of undirected graphs. Here, we briefly note that the algorithm can easily be extended to handle weighted graphs. The scan-first search algorithm is implemented as follows

Algorithm 4 Scan-first search [7]

1: **procedure** SFS($G=(V,E)$: ***Graph***, r : ***Vertex***)
2: Find a spanning tree T' rooted at r
3: Find a preorder numbering to the vertices in T'
4: For each vertex $v\in T'$ with $v\neq r$, let $b(v)$ denote the neighbor of v with the smallest preorder number
5: Let T be the tree formed by $\{v,b(v)\}$ for all $v\neq r$

Using a linear work, low depth spanning tree algorithm, scan-first search can easily be implemented in $O(m)$ work and $O(\log(n))$ depth. Cheriyan, Kao, and Thurimella show that if E_i are the edges in a scan-first search forest of the graph $G_{i-1}=(V,E\setminus(E_1\cup ...E_{i-1}))$, then $E_1\cup ...E_k$ is a sparse k-connectivity certificate. A sparse k-connectivity certificate can therefore be found in $O(km)$ work and $O(k\log(n))$ depth by running scan-first search k times.

In the weighted setting, we treat an edge of weight w as w parallel unweighted multiedges. As always, this is only conceptual, the multigraph is never actually generated. To compute certificates in weighted graphs, we therefore use the following simple modification. After computing each scan-first search tree, instead of removing the edges present from G, simply lower their weight by one, and remove them only if their weight becomes zero. It is easy to see that this is equivalent to running the ordinary algorithm on the unweighted multigraph. We therefore have the following.

Lemma 7. *Given a weighted, undirected graph $G=(V,E)$, a sparse k-connectivity certificate can be found in $O(km)$ work and $O(k\log(n))$ depth.*

Parallel Matula's algorithm for weighted graphs. Matula [27] gave a linear time sequential algorithm for a $(2+\varepsilon)$-approximation to 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. We will show that 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 degree of the graph. When $d=O(\log(n))$ and $W=O(m\,\mathrm{polylog}(n))$, this gives a work bound of $O(m\log(n)\log\log(n))$.

Essentially, Karger and Motwani's version of Matula's algorithm does the following steps as indicated in Algorithm 5.

Algorithm 5 Approximate minimum cut

```
1: procedure MATULA(G = (V, E) : Graph )
2:     local d ← minimum degree in G
3:     local k ← d/(2 + ε)
4:     local C ← Compute a sparse k-certificate of G
5:     local G' ← Contract all non-certificate edges of E
6:     return min(d, MATULA(G'))
```

It can be shown that at each iteration, the size of the graph is reduced by a constant factor, and hence there are at most $O(\log(n))$ iterations. Furthermore, the work performed at each step is geometrically decreasing, so the total work, using the sparse certificate algorithm of Cheriyan, Kao, and Thurimella [7] is $O(dm)$ and the depth is $O(d\log^2(n))$.

To extend this to weighted graphs, we can replace the sparse certificate routine with our modified version for weighted graphs, and replace the computation of d with the equivalent weighted degree. By interpreting an edge-weighted graph as a multigraph where each edge of weight w corresponds to w parallel multiedges, we can see that the algorithm is equivalent. To argue the cost bounds, note that like in the original algorithm where the size of the graph decreases by a constant factor each iteration, the total weight of the graph must decrease by a constant factor in each iteration. Because of this, it is no longer true that the work of each iteration is geometrically decreasing. Naively, this gives a work bound of $O(dm\log(W))$, but we can tighten this slightly as follows. Observe that after performing $\log(W/m)$ iterations, the total weight of the graph will have been reduced to $O(m)$, and hence, like in the sequential algorithm, the work must subsequently begin to decrease geometrically. Hence the total work can actually be bounded by $O(dm\log(W/m) + dm) = O(dm\log(W/m))$. We therefore have the following.

Lemma 8. *Given an undirected, weighted graph G with minimum weighted-degree d and total weight W, a constant-factor approximation to the minimum cut can be computed in $O(dm\log(W/m))$ work and $O(d\log(n)\log(W))$ depth.*

A parallel approximation algorithm for minimum cut. The final ingredient needed to produce the parallel minimum cut approximation is a trick due to Karger. Recall that to produce Karger's 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 verify whether the guess was too high by checking whether the minimum cut in the skeleton contained too few edges (Lemma 3). Specifically, Karger's sampling theorem (Lemma 6.3.2 of [19]) says that we will know that we have made the correct guess within a factor two when the skeleton has $\Omega(\log(n))$ edges in its minimum cut. To perform the minimum amount of work, we use Lemma 5 to first produce a $O(\log(n))$-approximation to the minimum cut, which allows us to make just $O(\log\log(n))$ guesses with the guarantee 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 consider the corresponding skeleton graph and compute a $\Theta(\log(n))$ certificate, since, by assumption, until we have made the correct guess, the minimum cut in the skeleton will be less than $O(\log(n))$ 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 (Lemma 8), since, after producing the certificate, the total weight of the graph is at most $O(n\log(n))$, and the minimum degree is no more than $O(\log(n))$. The details are depicted in Algorithm 6. It takes $O(m\log^2(n))$ work to produce the sequence of graph skeletons, $O(m\log(n)\log\log(n))$ work to compute the sparse certificates, and $O(n\log(n)(\log\log(n))^2)$ work to

compute the resulting approximations from Matula's algorithm. All together, the algorithm takes at most $O(m \log^2(n))$ work, and runs in $O(\log^3(n))$ depth.

Algorithm 6 Approximate minimum cut algorithm

```
1: procedure ApproxMinCut(G = (V, E) : Graph )
2:   local C ← A log(n)-approximation of MinCut(G)
3:   for c ∈ {C/ log(n), 2C/ log(n), ...C} do in parallel
4:     p(c) ← Θ(log(n)/c)
5:     local G_p ← Compute the skeleton graph G'(p)
6:     local G'_p ← Compute a Θ(log(n))-certificate of G_p
7:     ĉ(c) ← Matula(G'_p)
8:   if ĉ(C/ log(n)) ≤ O(log(n)) then
9:     return ĉ(C/ log(n))
10:  else
11:    local c ← min{c | ĉ(c) ≥ O(log(n))}
12:    return ĉ(c)/p(c)
```

To see that the final returned value is correct, we appeal to Karger's sampling theorem, which says that w.h.p., if our guess of the minimum cut is too high by a factor two, the minimum cut of the skeleton will have less than $O(\log(n))$ edges w.h.p. [19], and hence the certificate algorithm does not damage the minimum cut. Once our guess is below the minimum cut by a factor two, the sampling theorem says that the minimum cut of the skeleton exceeds $\Omega(\log(n))$. Provided that we set the constant of the $\Theta(\log(n))$ certificate to be a constant factor larger than this threshold, a Chernoff bound shows us that one of our guesses leads to a skeleton with approximate minimum cut $\hat{c} = \Omega(\log(n))$ that is not damaged by the certificate, and then Lemma 3 says that $\hat{c}/p$ is a constant-factor approximation of the minimum cut w.h.p. This argument works only if the minimum cut of G has size at least $\Omega(\log(n))$, but note that if it does not, the skeleton construction $G'(1)$ (which must occur during the last iteration) and the certificate completely preserve the minimum cut and hence the last iteration of the loop finds a constant factor approximation of the minimum cut in G.

B Mixed Component Weight Operations

Here, we describe a simple RC implementation of the following operations, which is hence amenable to our batched mixed operations framework.

- 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

The values stored in the RC clusters are as follows. Vertices store their weight, and unary clusters store the weight of the component reachable via joined edges from the boundary vertex. A binary cluster is either *joined*, meaning that its boundary vertices are connected by joined edges, in which case it stores a single value, the weight of the component reachable via joined edges from the boundaries, otherwise it is *split*, in which case it stores a pair: the weight of the component reachable via joined edges from the top boundary, and the weight of the component reachable via joined edges from the bottom boundary. We provide pseudocode for the update operations for Illustration in Algorithm 7.

Algorithm 7 A simple RC implementation of SubtractWeight and JoinEdge.

```
1: procedure f_UNARY(v_v, t, U)
2:    if t = (t_v, b_v) then return t_v
3:    else return v_v + t + Σ_{u_v∈U} u_v
4: procedure f_BINARY(v_v, t, b, U)
5:    if t = t_v and b = b_v then
6:       return t_v + b_v + v_v + Σ_{u_v∈U} u_v
7:    else if t = (t_{t_v}, t_{b_v}) and b = b_v then
8:       return (t_{t_v}, t_{b_v} + v_v + b_v + Σ_{u_v∈U} u_v)
9:    else if t = t_v and b = (b_{t_v}, b_{b_v}) then
10:      return (t_v + v_v + b_{t_v} + Σ_{u_v∈U} u_v)
11:   else if t = (t_{t_v}, t_{b_v}) and b = (b_{t_v}, b_{b_v}) then
12:      return (t_{t_v}, b_{b_v})
13: procedure SubtractWeight(v, w)
14:   v→value ← v→value - w
15:   Reevaluate the f(·) on path to root.
16: procedure JoinEdge(e)
17:   e→value ← 0
18:   Reevaluate the f(·) on path to root.
```

The initial value of a vertex is its starting weight. The initial value of an edge is $(0, 0)$, indicating that it is split at the beginning. Note that f_{unary} and f_{binary} can be evaluated in constant time, and the structure of the updates involves setting the value at a leaf using an associative operation and re-evaluating the values of the ancestor clusters.

We can argue that the values are correctly maintained by structural induction. First consider unary clusters. If the top subcluster is split, then the representative vertex and unary subclusters are not reachable via joined edges, and hence the only reachable component is the component reachable inside the top subcluster from its top boundary, whose weight is t_v. If the top subcluster is joined, then the representative vertex is reachable, which is by definition the boundary vertex of the unary subclusters, and hence the reachable component is the union of the reachable components of all of the subclusters, whose weight is as given.

For binary clusters, there are four possible cases, depending on whether the top and bottom subclusters are joined or not. If both are joined, then the representative and hence the boundary of all subclusters is reachable from both boundaries, and hence the cluster is joined and the reachable component is the union of the reachable components of the subclusters. If either subcluster is split, then the reachable component at the corresponding boundary is just the reachable component of the subcluster, whose weight is as given. Lastly, if one of the subclusters is not split, then the corresponding boundary can reach the representative vertex, and hence the reachable components of the unary subclusters, whose weights are as given.

It remains to argue that we can implement QueryWeight with a simple RC implementation. Consider a vertex v whose component weight is desired and consider the parent cluster P of v, i.e., the cluster of which v is the representative. If P has no binary subclusters that are joined, observe that P must contain the entire component of v induced by joined edges, since the only way for a component to exit a cluster is via a boundary which would have to be joined. Answering the query in this situation is therefore easy; the result is the sum of the weights of v, the unary subclusters of P, the bottom boundary weight of the top subcluster (if it exists), and the top bounary weight of the bottom subcluster (if it exists). Suppose instead that P contains a binary subcluster that is joined to some boundary vertex $u \neq v$. Since the subcluster is joined, u is in the same induced

component as v, and hence QueryWeight(v) has the same answer as QueryWeight(u). By standard properties of RC trees, since u is a boundary of P, we also know that the leaf cluster u is the child of some ancestor of P. Since the root cluster has no binary subclusters, this process of jumping to joined boundaries must eventually discover a vertex that falls into the easy case, and since such a vertex u is always the child of some ancestor is P, the algorithm only examines clusters that are on or are children of the root-to-v path in the RC tree, and hence the algorithm is a simple RC implementation.

C Sampling Binomial Random Variables

Our graph sampling procedure makes use of binomial random variables. We will use the following results due to Farach-Colton et al. [9].

Lemma 9 (Farach-Colton et al. [9], Theorem 1)**.** *Given a positive integer n, one can sample a random variate from the binomial distribution $B(n, 1/2)$ in $O(1)$ time with probability $1 - 1/n^{\Omega(1)}$ and in expectation after $O(n^{1/2+\varepsilon})$-time preprocessing for any constant $\varepsilon > 0$, assuming that $O(\log(n))$ bits can be operated on in $O(1)$ time. The preprocessing can be reused for any $n' = O(n)$*

We can also use the following reduction to sample $B(n, p)$ for arbitrary $0 \leq p \leq 1$.

Lemma 10 (Farach-Colton et al. [9], Theorem 2)**.** *Given an algorithm that can draw a sample from $B(n', 1/2)$ in $O(f(n))$ time with probability $1 - 1/n^{\Omega(1)}$ and in expectation for any $n' \leq n$, then drawing a sample from $B(n', p)$ for any real p can be done in $O(f(n) \log(n))$ time with probability $1 - 1/n^{\Omega(n)}$ and in expectation, assuming each bit of p can be obtained in $O(1)$ time*

We note, importantly, that the model used by Farach-Colton et al. assumes that random $\Theta(\log(n))$-size words can be generated in constant time. Since we only assume that we can generate random bits in constant time, we will have to account for this with an extra $O(\log(n))$ factor in the work where appropriate. Note that this does not negatively affect the depth since we can pregenerate as many random words as we anticipate needing, all in parallel at the beginning of our algorithm. Lastly, we also remark that although it might not be clear in their definition, the constants in the algorithm can be configured to control the constant in the $\Omega(1)$ term in the probability, and therefore their algorithms take $O(1)$ time and $O(\log(n))$ time w.h.p.

Preprocessing in parallel. In order to make use of these results, we need to show that the preprocessing of Lemma 9 can be parallelized. Thankfully, it is easy. The preprocessing phase consists of generating n^{ε} alias tables of size $O(\sqrt{n \log(n)})$. Hübschle-Schneider and Sanders [18] give a linear work, $O(\log(n))$ depth parallel algorithm for building alias tables. Building all of them in parallel means we can perform the alias table preprocessing in $O(n^{1/2+\varepsilon})$ work and $O(\log(n))$ depth. The last piece of preprocessing information that needs to be generated is a lookup table for decomposing any integer $n' = O(n)$ into a sum of a constant number of square numbers. This table construction is trivial to parallelize, and hence all preprocessing runs in $O(n^{1/2+\varepsilon})$ work and $O(\log(n))$ depth.

D Bulk Path Updates For RC Trees

Given an RC tree of a tree on n vertices and a set of k path updates of the form (v_i, x_i), denoting that x_i is to be added to the weight of all edges on the path from r to v_i, we can apply all of them in $O(n)$ work and $O(\log(n))$ depth w.h.p. The idea is similar to a standard treefix sum algorithm.

First, the algorithm associates each vertex v_i with its weight x_i. Now the operation is to add to each edge (u, v), where v is a child of u, the weight of all vertices in the subtree rooted at v. The algorithm then proceeds in two steps. First, in a traversal of the RC tree, it computes, for each cluster C, the total weight $W(C)$ of all vertices in it. This step is the same as the first step of our batch evaluation algorithm, and takes $O(n)$ work and $O(\log(n))$ depth w.h.p. Second, for each child cluster of the root, it traverses the RC tree top-down, maintaining the weight w of all vertices that are descendants of the current cluster (initially zero). This is achieved when recursing on $C{\rightarrow}t$, by adding $W(C) - W(C{\rightarrow}t)$ to the accumulated weight w, otherwise keeping w the same. When reaching a base edge cluster, the value w is added to the weight of the edge. This takes $O(n)$ work and $O(\log(n))$ depth w.h.p.

E Additional Mixed Tree Operations

Algorithm 8 A simple RC implementation of QueryEdge and QueryPath'. For QueryPath' the node v must be the representative node of an ancestor of u in the RC tree. It maintains m, the minimum off cluster path, t, the minimum on cluster path above where the path from u meets the cluster path, and b, the minimum on cluster path below that point. Once v is found it picks one of b or t depending on which side v is, or neither if a unary cluster. If a binary cluster it then needs to continue up the tree to add in the additional weights for binary clusters.

```
1:  procedure QueryEdge(e)
2:      w ← w(e)
3:      x ← e; p ← x→p
4:      while binary p do
5:          if x = p→t 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
9:  procedure QueryPath'(u, v)
10:     m ← ∞; t ← ∞; b ← ∞
11:     x ← u; p ← x→p
12:     while not p→v = v do
13:         w' ← p→v→w + Σ_{u∈p→U} u→w
14:         if unary p then
15:             if x = p→t then m ← min(t + w', m)
16:             else m ← min(p→t→l + w', m)
17:             t ← ∞; b ← ∞;
18:         else
19:             w' ← w' + p→b→w
20:             if x = p→t then t ← t + w'; b ← min(b, p→b→l)
21:             else if x = p→b then t ← min(p→t→l + w', t)
22:             else t ← p→t→l + w'; b ← p→t→l
23:         x ← p; p ← x→p
24:     if x = p→t then l ← b
25:     else if x = p→b then l ← t
26:     else return m
27:     while binary p do
28:         w' ← p→v→w + p→b→w + Σ_{u∈p→U} u→w
29:         if (x = p→t) then l ← l + w'
30:     return min(m, l)
```