Parallel Dynamic Tree Contraction via Self-Adjusting Computation

Umut A. Acar[*†] Vitaly Aksenov[†‡] Sam Westrick[*]
[*]Carnegie Mellon University, USA
[†]Inria, France
[‡]ITMO University, Saint-Petersburg, Russia

Abstract

Dynamic algorithms are used to maintain a property of some data while the data undergoes changes over time. For example, a dynamic algorithm on forests may maintain connectivity information as edges are inserted into (or deleted from) the forest. For many problems, dynamic algorithms can achieve sublinear, usually poly-logarithmic, update times for simple small changes such as insertion/deletion of an edge. While many dynamic algorithms have been designed, analyzed, and implemented, nearly all of them are sequential.

In this paper we present a parallel and dynamic algorithm for the dynamic trees problem, which requires computing a property of a forest as the forest undergoes changes by insertions and deletions of edges and nodes. Our algorithm allows insertion and/or deletion of both vertices and edges anywhere in the input (as long as these changes still result in a forest). For a forest of size n, our algorithm applies m changes in expected $O(m \log \frac{n+m}{m})$ work and $O(\log(n+m)C(m))$ parallel time where $C(m)$ is the time of compaction on m elements (typically logarithmic or less, depending on the model). Thus, when m is a small constant the algorithm performs $O(\log n)$ work; as m approaches n, work approaches $O(n)$, which is optimal.

We obtain our algorithm by applying a design technique, called self-adjusting computation, which enables dynamizing static algorithms, to the classic algorithm of Miller and Reif for tree contraction. One advantage of self-adjusting computation is that it can dramatically simplify the design and implementation of a dynamic algorithm, sometimes even allowing for automation of the dynamization process. We take advantage of this and specify the algorithm precisely in the PRAM model and implement it on modern multicore machines. Our empirical evaluation shows that the algorithm performs well in practice, yielding sublinear time updates and scaling well with increasing numbers of processors (cores).

Keywords dynamic; parallel; self-adjusting computation; change propagation; tree; forest; contraction; rake; compress; PRAM; fork-join

1. Introduction

In many applications, algorithms operate on trees that change dynamically over time. For example, an algorithm may compute the heaviest subtree in a edge-weighted tree and may be required to update the result as the tree undergoes changes, e.g., as nodes or edges are inserted and/or deleted. Algorithms that allow efficient computation on such dynamically changing trees, called *dynamic tree algorithms,* have been studied extensively since the early '80s. The proposed algorithms include Link-Cut Trees [35, 36], Euler-Tour Trees [21, 39], Topology Trees [17], RC-Trees [2, 4], and, more recently, Top Trees [10, 38, 40].

All of this prior work, however, considers *sequential* dynamic algorithms. There is relatively little work on *parallel dynamic algorithms*, which would allow updating the result of a computation efficiently under changes to the input data while also taking advantage of parallelism. For example, a parallel dynamic algorithm can perform updates by performing work that is sub-linear in the size of the input and also achieve fast, usually sub-linear *span*, i.e., parallel time, in the size of the input as well as the size of the input-change applied.

Historically, dynamic and parallel algorithms have mostly been studied separately. We can identify at least two challenges that make the design, analysis, and the implementation of parallel dynamic algorithms nontrivial.

- Dynamic algorithms are traditionally designed to handle small changes to the input. Small changes, however, do not generate sufficient parallelism. Larger *batches of changes* can generate parallelism but this requires generalizing the algorithms.
- Dynamic algorithms and parallel algorithms on their own are usually quite complex to design, analyze, and implement. Since parallel dynamic algorithms combine the features of both, their implementation can become a significant hurdle.

We can see the challenges in existing sequential dynamic tree algorithms [2, 4, 10, 17, 21, 35, 36, 38–40]. These algorithms all allow the insertion/deletion of a single edge in logarithmic time (some in expectation, some amortized). A batch of multiple edges can be inserted by iterating over the edges in the batch but this is not parallel. The iterative approach also performs more work than the optimal algorithm by as much as $\Omega(\lg n)$-factor, and thus is not *work efficient*. On the practical side, some of these algorithms have been implemented and evaluated in practice [4, 40] but all implementations are sequential.

Prior work by Reif and Tate [33] made some progress on the parallel dynamic trees problem. Reif and Tate proposed an algorithm that, for a set of m changes on a forest of n nodes, requires $O\left(m \log n\right)$ total work and $O\left(\log m + \log\log n\right)$ span or parallel time. Their algorithm, however, leaves open several questions:

1. the algorithm is not fully general: 1) it only allows changes at the leaves of the tree; 2) it does not specify how to perform deletions, leaving it to future work;
2. the algorithm is not work efficient, because it requires $\Omega(n \lg n)$ work for large changes;
3. the algorithm is specified informally without a detailed specification or implementation, leaving open the question of whether it can be implemented efficiently.

In this paper, we present an algorithm for parallel dynamic trees that overcomes these limitations. Our algorithm is fully general: it allows insertion and deletion of any number of nodes or edges anywhere in the input forest (as long as no cycles are created). It is work efficient and highly parallel: for a forest with n nodes and a set of m changes (insertion/deletion of nodes/edges), the algorithm performs, in expectation, $O(m \log \frac{n+m}{m})$ work in $O(\log(n+m)C(m))$ span (parallel time), where $C(m)$ is the parallel time of compaction on m elements. These bounds mean that our algorithm requires work that is sub-linear in the size of the input n and requires parallel time that is sub-linear in the size of both the input n and the input change m. The algorithm is work efficient, because as m approaches n the work approaches $O(n)$, which is optimal. When m is constant, the bound is $O(\log n)$. The total work of the algorithm thus increases gracefully as m increases and converges to the work-optimal bound as m approaches n. We specify our algorithm precisely and reasonably succinctly in the work-time framework and in the CREW PRAM model.

Our algorithm is also practical: following the specification, we have implemented the algorithm on modern multicore machines by using a fork-join parallelism library in C++ [5] with capabilities similar to Cilk [18]. The results confirm our theoretical bounds and show that the algorithm can work very well in practice by combining the complementary benefits of dynamic and parallel algorithms. The algorithm handles small batches of changes efficiently and quickly yielding orders of magnitude speedups over an optimized static algorithm. As the size of the input changes increases, the total work increases but so does parallelism, continuing to result in speedups. Our experiments show overall that the dynamic parallel algorithm is able to perform quite well, realizing sub-linear time updates and parallelism for a set of composite changes.

Our approach is based on the technique of *self-adjusting computation* for dynamizing static algorithms. The idea behind this technique is to represent the execution of an algorithm by using a *computation graph*, which captures important data and control dependencies in the execution. When the input data is changed, a *change-propagation* technique is used to

update the computation (including the result) by identifying the pieces of the computation affected by the change and re-building them. Change propagation can be viewed as selectively re-executing the static algorithm while re-using results unaffected by the changes made. Prior work showed that this technique can be used to solve a number of algorithmic problems. For example, applying self-adjusting computation to Miller and Reif's sequentialized version of tree-contraction algorithm yields an efficient data structure for dynamic trees [2, 4] but this data structure is sequential. Other applications include algorithmic problems in computational geometry [6–8], and machine learning [9, 37].

Our approach consists of two algorithms, called construction and dynamic update. The *construction algorithm* performs a randomized tree contraction on an input forest to construct a computation graph which we call the *contraction data structure*, by applying Miller and Reif's [29] tree contraction algorithm. The *dynamic-update algorithm*, which uses the change-propagation technique, takes as input a contraction data structure $\mathcal{C}$ and a change set $\mathcal{M}$, and produces an updated contraction data structure $\mathcal{C}'$ that is equivalent to one that would be obtained by re-executing the construction algorithm on the changed input. To this end, dynamic update mimics the execution of the construction algorithm on the whole input, but does so efficiently by only identifying parts of $\mathcal{C}'$ that are affected by the input change $\mathcal{M}$. Since it is behaviorally equivalent to re-computing from scratch, dynamic update can be iterated as desired to perform any sequence of changes.

Due to space restrictions, we give details of some proofs and additional experiments in the appendix.

2. Tree Contraction

2.1 Overview

Tree contraction is the process of shrinking a tree down to a single vertex. In this paper, we will be focusing on a technique introduced by Miller and Reif ([28]) which makes use of two operations: *rake* and *compress*. The former removes all leaves from the tree, while the latter removes an independent set of vertices which lie on a chain. One *round* of tree contraction consists of the simultaneous application of rake and compress across the tree.

Various versions of tree contraction have been proposed depending on how the independent set is selected for the compress operation [28]. We use the randomized approach where coin flips are used to break symmetry. Miller and Reif showed that it takes $O(\log n)$ rounds w.h.p. to fully contract a tree of n vertices in this manner.

Tree contraction has a number of useful applications, studied extensively in [4, 29, 30]. It can be used to perform various computations by associating data with edges and vertices and defining how data is accumulated during rake and compress operations. In our presentation, we leave these operations to be user-defined.

2.2 Preliminaries

The algorithms described here operate on rooted forests $F = (V, E)$, where V is a set of vertices, and E is a set of directed edges. Each $(u, v) \in E$ indicates that u is a *child* of v, or equivalently that v is the *parent* of u. If either $(u, v) \in E$ or $(v, u) \in E$, we say that u and v are *adjacent*, or equivalently that they are *neighbors*. A vertex with no parent is a *root*.

Note that the choice of edge direction (edges pointing from child to parent) is arbitrary. Our algorithms can be trivially modified to support forests with edges pointing from parent to child.

For analysis later, we will assume that the forests given as input have bounded degree. That is, there exists some constant t such that each vertex has at most t children.

The algorithms in this paper work in rounds, each of which take a forest from the previous round as input and produce a new forest for the next round. When describing the algorithms, we use the term "at round i" to refer to the beginning of round i, and the term "in round i" to refer to some computation that occurs during round i. We count rounds starting at 0. For an input forest F, we write $F^i = (V^i, E^i)$ to refer to the forest at round i, and thus $F = F^0$.

We say that a vertex v is *alive* at round i if $v \in V^i$, and is *dead* at round i if $v \notin V^i$. If $v \in V^i$ but $v \notin V^{i+1}$ then we say that v *contracts* in round i. There are three ways for a vertex to contract: it either *finalizes*, *rakes*, or *compresses*. For some v at round i, we denote its parent with $p^i(v)$, its set of children with $c^i(v)$, and its degree with $\delta^i(v) = \left|c^i(v)\right|$. If v is a root at round i, then $p^i(v) = v$. A vertex is *isolated* at round i if it is a root and $\delta^i(v) = 0$.

When multiple forests are in play, it will be necessary to disambiguate which is in focus. For this, we will use subscripts: for example, $\delta^i_F(v)$ is the degree of v in the forest F^i, and E^i_G is the set of edges in the forest G^i.

2.2.1 Work-Time Framework

We present our algorithms in pseudocode with a single parallel construct: the parallel **for** loop. This is essentially the same as the Work-Time framework [22]. Algorithms written in this style can easily be adapted into more concrete models. For example, we can derive a PRAM algorithm via a scheduling principle such as the one originally described by Brent [12]. We can also derive a fork-join algorithm by implementing each parallel **for** loop as a balanced binary tree of forks where each leaf corresponds to one instance of the body of the loop.

We measure the performance of our algorithms in terms of work and parallel time. Work corresponds to the total number of operations performed. Parallel time (also referred to as *depth* and *span* in other models) is the longest chain of sequential dependencies. An algorithm with work W and time T can then be run on a p-processor PRAM in time $O(W/p + T)$.

For ease of presentation, we use concurrent sets with atomic operations for insertion and deletion, written "insert x into S" and "delete x from S". We discuss how to implement these operations efficiently in Section 2.6. We also use parallel **for** loops of the form "**for** $x \in S$ **where** $p(x)$ **do in parallel**," with the understanding that we can index into S in constant time, and that the body of the loop is executed only if the predicate $p(x)$ is satisfied.

2.3 Contraction Data Structure

The algorithms in this paper manipulate a common data structure which serves as a record of the contraction process. A *contraction data structure* consists of a triple (P, C, D), where $P[i][v]$ is the parent of v at round i, $C[i][v]$ is the set of children of v at round i, and $D[v]$ is the "duration" of v, meaning the number of rounds that v remains alive.

A contraction data structure is *valid* for a forest F if for every v, $D[v]$ is the minimum i such that $v \notin V_F^i$, and for every $i < D[v]$, we have $P[i][v] = p_F^i(v)$ and $C[i][v] = c_F^i(v)$.

2.4 Construction Algorithm

Figure 1 shows pseudocode for FORESTCONTRACTION, which takes as input a forest (V, E) and uses Miller and Reif's tree contraction algorithm to construct a contraction data structure (P, C, D). For ease of presentation, we treat P, C, and D as global variables.

The algorithm begins by initializing P, C, and D for round 0. It then proceeds with a **while** loop, each iteration of which performs one round of contraction by calling RANDOMIZEDCONTRACT. The algorithm terminates as soon as all vertices are dead.

The procedure RANDOMIZEDCONTRACT takes in the set of alive vertices at round i, constructs a new forest for round $i + 1$ by making changes to $P[i + 1]$, $C[i + 1]$, and D, and produces the set of alive vertices for the next round.

Figure 2 contains definitions of auxiliary functions, used in both the construction algorithm and elsewhere. We now briefly describe these functions.

The procedure PROMOTEEDGES determines which edges are present in the next round and adds them by writing to $P[i + 1]$ and $C[i + 1]$. An edge remains if both of its endpoints do not contract. New edges are added only in the case of a compression. PROMOTEEDGES also performs application-specific data manipulation via the user-defined functions DOFINALIZE, DORAKE, and DOCOMPRESS. These operations are essentially the same as in Miller and Reif's original algorithm, except for the finalize operation, which is useful in certain contexts such as constructing an RC-Tree [4].

The functions FINS, RAKES, COMPS, and CONTRACTS specify when a vertex contracts. If a vertex is isolated, then it finalizes; if it has no children, then it rakes; if it has one child, then it may or may not compress. Coin flips are used to break symmetry in the compress case, since it is necessary that the vertices chosen for compression at each round form an independent set.

Coin flips are generated by the function HEADS, which hashes a vertex v at round i to $\{0, 1\}$ and returns whether or not it hashed to 1. We use a family H of 2-wise independent hash functions mapping V to $\{0, 1\}$. We randomly select one member of H for each round of contraction.

2.5 Dynamic Update Algorithm

Figure 3 shows pseudocode for our dynamic tree contraction algorithm, MODIFYCONTRACTION. The input is given as $((V^-, E^-), (V^+, E^+))$, indicating that the vertices V^- and edges E^- should be deleted, while the vertices V^+ and edges E^+ should be added. The goal of MODIFYCONTRACTION is to edit the contraction data structure produced by FORESTCONTRACTION(V, E) such that, afterwards, it is valid for the forest $((V \setminus V^-) \cup V^+, (E \setminus E^-) \cup E^+)$. For simplicity, we assume that $V^- \subseteq V$, and that V^+ is disjoint from V (and thus by extension, V^+ is also disjoint from V^-). We assume the same on E, E^- and E^+.

MODIFYCONTRACTION makes initial changes based on the input and then propagates these changes via repeated calls to our change propagation algorithm, PROPAGATE, as shown in Fig. 4. The design of change propagation is driven by the observation that, when a vertex locally makes a choice about whether or not to contract, it only needs to know who its neighbors are, and whether or not any of its children are leaves. This motivates the definition of the *configuration* of a vertex v at round i, denoted $\kappa_F^i(v)$, defined as

$$\kappa_F^i(v) = \begin{cases} (p_F^i(v), \{(u, \ell_F^i(u)) : u \in c_F^i(v)\}), & \text{if } v \in V_F^i \\ \text{dead}, & \text{if } v \notin V_F^i, \end{cases}$$

```
1  global P, C, D
2
3  procedure ForestContraction(V, E)
4    for v ∈ V do in parallel
5      C[0][v] ← {u : (u, v) ∈ E}
6      if ∃p.(v, p) ∈ E then P[0][v] ← p
7      else P[0][v] ← v
8    local L ← V
9    local i ← 0
10   while |L| > 0 do
11     L ← RandomizedContract(i, L)
12     i ← i + 1
13
14 procedure RandomizedContract(i, L)
15   for v ∈ L do in parallel
16     P[i + 1][v] ← v
17     C[i + 1][v] ← ∅
18   PromoteEdges(i, L)
19   for v ∈ L where Contracts(i, v) do in parallel
20     D[v] ← i + 1
21   return {v ∈ L | ¬Contracts(i, v)}
```

Figure 1. Construction algorithm

where $\ell_F^i(u)$ indicates whether or not $\delta_F^i(u) = 0$ (the "leaf" status of u).

Consider some input forest $F = (V, E)$, and let $G = ((V \setminus V^-) \cup V^+, (E \setminus E^-) \cup E^+)$ be the newly desired input. We say that a vertex v is *affected* at round i if $\kappa_F^i(v) \neq \kappa_G^i(v)$. Vertices which are not affected at round i have nice properties, as illustrated by Lemmas 1 and 2.

Lemma 1. *If v is unaffected at round i, then either v is dead at round i in both F and G, or v is adjacent to the same set of vertices in both.*

Proof. Follows directly from $\kappa_F^i(v) = \kappa_G^i(v)$. □

Lemma 2. *If v is unaffected at round i, then v contracts in round i of F if and only if v also contracts in round i of G, and in the same manner (finalize, rake, or compress).*

Proof. Follows from $\kappa_F^i(v) = \kappa_G^i(v)$ and the code of Contracts and RandomizedContract, as shown in Figure 1. □

Due to Lemma 2, at each round, all unaffected vertices do not need to be updated. For the affected vertices, we need to rerun the contraction process. This is accomplished by first deleting all edges which are incident upon an affected vertex, and then rebuilding this portion of the forest by promoting edges. Note that in addition to promoting edges incident upon affected vertices, we also have to promote edges incident upon any neighbor of an affected vertex. This is due to vertices editing their neighbors' state rather than their own.

2.5.1 Becoming Affected

If a vertex v is not affected at round i but is affected at round $i + 1$, then we say that v *becomes affected in round i*. A vertex can become affected in a couple different ways, as enumerated in Lemma 3.

Lemma 3. *If v becomes affected in round i, then at least one of the following holds:*

1. *v has an affected neighbor u at round i which contracts in that round in either F^i or G^i.*
2. *v has an affected child u at round $i + 1$ where $\ell_F^{i+1}(u) \neq \ell_G^{i+1}(u)$.*

Proof. First, note that since v becomes affected, we know v does not contract, and furthermore that v has at least one child at round i. (If v were to contract, then by Lemma 2 it would do so in both forests, leading it to being dead in both forests at

```
1  procedure PROMOTEEDGES(i, L)
2    for v ∈ L where ¬CONTRACTS(i, v) do in parallel
3      if P[i][v] ≠ v then insert v into C[i + 1][P[i][v]]
4      for u ∈ C[i][v] do P[i + 1][u] ← v
5    for v ∈ L where CONTRACTS(i, v) do in parallel
6      if FINS(i, v) then DOFINALIZE(i, v)
7      if RAKES(i, v) then DORAKE(i, v)
8      if COMPS(i, v) then
9        DOCOMPRESS(i, v)
10       let {u} = C[i][v]
11       insert u into C[i + 1][P[i][v]]
12       P[i + 1][u] ← P[i][v]
13
14 function FINS(i, v) // finalizes at round i?
15   return P[i][v] = v ∧ C[i][v] = ∅
16
17 function RAKES(i, v) // rakes at round i?
18   return P[i][v] ≠ v ∧ C[i][v] = ∅
19
20 function COMPS(i, v) // compresses at round i?
21   if ∃u.C[i][v] = {u} then
22     let p = P[i][v]
23     return C[i][u] ≠ ∅ ∧ ¬HEADS(i, p) ∧ HEADS(i, v)
24   else return false
25
26 function CONTRACTS(i, v)
27   return FINS(i, v) ∨ RAKES(i, v) ∨ COMPS(i, v)
28
29 function HEADS(i, v) // did v flip heads on round i?
```

Figure 2. Auxiliary Functions

the next round and therefore unaffected. If v were to have no children, then v would rake, but we just argued that v cannot contract).

Suppose that the only neighbors of v which contract in round i are unaffected at round i. Then v's set of children in round $i+1$ is the same in both forests. If all of these are unaffected at round $i+1$, then their leaf statuses are also the same in both forests at round $i+1$, and hence v is unaffected, which is a contradiction. Thus case 2 of the lemma must hold. In any other scenario, case 1 of the lemma holds. □

Lemma 4. *If v does not contract in either forest in round i and $\ell_F^{i+1}(v) \neq \ell_G^{i+1}(v)$, then v is affected at round i.*

Proof. Suppose v is not affected at round i. If none of v's neighbors contract in this round in either forest, then $\ell_F^{i+1}(v) = \ell_G^{i+1}(v)$: contradiction. Otherwise, if the only neighbors which contract do so via a compression, since compression preserves the degree of its endpoints, we will also have $\ell_F^{i+1}(v) = \ell_G^{i+1}(v)$ and thus a contradiction. So, we consider the case of one of v's children raking. However, since v is unaffected, we know $\ell_F^i(u) = \ell_G^i(u)$ for each child u of v. Thus if one of them rakes in round i in one forest, it will also do so in the other, and we will have $\ell_F^{i+1}(v) = \ell_G^{i+1}(v)$. Therefore we conclude that v must be affected at round i. □

Lemmas 3 and 4 tell us how to construct a set of affected vertices for a consecutive round of contraction: each affected vertex which contracts affects its neighbors, and each affected vertex whose leaf status is different in the two forests at the next round affects its parent. This strategy actually overestimates which vertices are affected, since case 1 of Lemma 3 does not imply that v is necessarily affected at the next round.

We make an additional assumption that *vertices which become affected stay affected until dead in both forests*. As a result of this assumption and the overestimation implied by the previous lemmas, the set of affected vertices used by our algorithm at each round may also contain vertices which are unaffected. This has no impact on the correctness of the

```
1  procedure ModifyContraction((V⁻, E⁻), (V⁺, E⁺))
2    let U = ⋃_{(u,v)∈(E⁻∪E⁺)} {u, v}
3    local L ← (U \ V⁻) ∪ V⁺
4    for v ∈ U \ V⁻ do in parallel
5      if ∃p.(v, p) ∈ E⁻ then P[0][v] ← v
6      if ∃p.(v, p) ∈ E⁺ then P[0][v] ← p
7      let ℓ = (C[0][v] = ∅)
8      C[0][v] ← C[0][v] \ {x : (x, y) ∈ E⁻ | v = y}
9      C[0][v] ← C[0][v] ∪ {x : (x, y) ∈ E⁺ | v = y}
10     let ℓ′ = (C[0][v] = ∅)
11     if ℓ′ ≠ ℓ then insert P[0][v] into L
12   for v ∈ V⁺ do in parallel
13     D[v] ← 0 // pretend that v was previously dead
14     C[0][v] ← {u : (u, v) ∈ E⁺}
15     if ∃p.(v, p) ∈ E⁺ then P[0][v] ← p
16     else P[0][v] ← v
17   local i ← 0
18   local X ← {(v, 0) : v ∈ V⁻}
19   while |L| > 0 ∨ |X| > 0 do
20     (L, X) ← Propagate(i, L, X)
21     i ← i + 1
```

Figure 3. Dynamic update algorithm

algorithm because it is always safe to re-evaluate unaffected vertices. We will see later that these assumptions also do not impact cost bounds.

2.5.2 Algorithm Description

On input $((V^-, E^-), (V^+, E^+))$, ModifyContraction begins by deleting E^- and inserting E^+ into $P[0]$ and $C[0]$. In the process, it identifies the set of affected vertices at round 0. Vertices in V^- are included in X to indicate that they are alive in F^0 but dead in G^0. Vertices v in V^+ are marked with $D[v] = 0$ as though they had always existed but were previously dead. ModifyContraction then runs change propagation, updating P, C, and D in rounds until no affected vertices remain.

At each round i, change propagation takes as input a set L containing all affected vertices which are alive in G at round i and a set X containing pairs (v, j) indicating that v is affected and first dead in G at round j, where $j \leq i$. It then updates $P[i+1]$, $C[i+1]$, and D appropriately, and produces new sets of affected vertices (alive and dead) for the next round. Specifically, lines 10-11 of Propagate delete all edges incident on affected vertices, and lines 14-16 calculate the sets of affected vertices for the next round.

Affected vertices v which contract in round i are added to X as the pair $(v, i+1)$, indicating that v is first dead in G at round $i+1$. Any vertices in X which are dead in both forests are removed (line 16), at which point their durations are updated (lines 17-18).

The function LeafStatuses is used to check whether or not the leaf status of affected vertices is different at round $i+1$ in F and G. It suffices to only consider vertices which do not contract in G at round i and are alive in F at round $i+1$, because if a vertex contracts in either forest at round i, then all of its neighbors will already be affected.

The function Spread identifies vertices which become affected or remain affected and alive.

2.6 Implementation of Concurrent Sets

We now describe how to implement the concurrent set operations used in this section. Using these techniques, our algorithms are valid for a CREW PRAM. We do not make use of concurrent writes, however we leave one subroutine unspecified (compaction). Any compaction algorithm is suitable. For the analysis, we assume that compaction performs $O(n)$ work for an input of size n. Many existing compaction algorithms fit this criteria. Note that the fastest known compaction techniques run on a CRCW PRAM; see Section 5 for more detail.

```
1  procedure PROPAGATE(i, L, X)
2    let ℓ = LEAFSTATUSES(i, L)
3    let NL = ⋃_{v∈L} {v, P[i][v]} ∪ C[i][v]
4    let NLX = NL ∪ {v : (v, _) ∈ X}
5    for v ∈ NL do in parallel
6      if D[v] ≤ i + 1 then
7        P[i + 1][v] ← v
8        C[i + 1][v] ← ∅
9      else
10       if P[i + 1][v] ∈ NLX then P[i + 1][v] ← v
11       C[i + 1][v] ← {u ∈ C[i + 1][v] | u ∉ NLX}
12   PROMOTEEDGES(i, NL)
13   let ℓ′ = LEAFSTATUSES(i, L)
14   let L′ = SPREAD(i, L, ℓ, ℓ′)
15   let X′ = X ∪ {(v, i + 1) : v ∈ L | CONTRACTS(i, v)}
16   let X″ = {(v, j) ∈ X′ | D[v] > i + 1}
17   for (v, j) ∈ X′ where D[v] ≤ i + 1 do in parallel
18     D[v] ← j
19   return (L′, X″)
20
21 function LEAFSTATUSES(i, L)
22   local ℓ
23   for v ∈ L where ¬CONTRACTS(i, v) ∧ D[v] > i + 1 do in parallel
24     ℓ[v] ← (C[i + 1][v] = ∅)
25   return ℓ
26
27 function SPREAD(i, L, ℓ, ℓ′)
28   local L′ ← ∅
29   for v ∈ L where CONTRACTS(i, v) do in parallel
30     if P[i][v] ≠ v then insert P[i][v] into L′
31     for u ∈ C[i][v] do insert u into L′
32   for v ∈ L where ¬CONTRACTS(i, v) do in parallel
33     insert v into L′
34     if D[v] = i + 1 then
35       insert P[i + 1][v] into L′
36       for u ∈ C[i + 1][v] do insert u into L′
37     if D[v] > i + 1 ∧ ℓ′[v] ≠ ℓ[v] then
38       insert P[i + 1][v] into L′
39   return L′
```

Figure 4. Change Propagation

We implement $C[i][v]$ as an array of size t, where t is the maximum number of children of any vertex. Each cell of this array is either empty, or contains the identifier of a vertex. We then associate each parent pointer with an index. For example, for each edge $(u, v) \in E^i$, we would store (v, j) at $P[i][u]$, indicating that the j^{th} cell of $C[i][v]$ contains u.

If u ever edits $C[i][v]$, it will do so by only editing the value stored in the j^{th} cell. If u has some child w and u compresses, then w will inherit the parent pair (v, j), indicating that w is now responsible for updating the j^{th} cell of v's child array on subsequent rounds.

In PROPAGATE, on line 3, we need to build an array containing the unaffected neighbors of affected vertices. We can do this by giving each vertex v a "proposals" array of length $t + 1$, where each of v's neighbors knows a unique index into this array (we could use the same index as in the array of children, with the last proposal cell reserved for the parent).

Each affected vertex v then can propose to an unaffected neighbor u by writing itself at the reserved index in u's proposals array. Each unaffected vertex then accepts a single proposal, such as the first one which appears in its proposal

array. We then allocate an array of length $(t+1)|L|$ and each affected vertex v will attempt to write each of its unaffected neighbors into this array, but only if the neighbor accepted v's proposal. This guarantees that each unaffected neighbor is only written once. We then need to compact the newly created array of unaffected neighbors by eliminating empty cells.

A similar "proposals" technique can be used to construct the new set of affected vertices within the function SPREAD. We represent other sets simply as arrays, with their elements stored in no particular order. To perform a filtering operation (for example, $\{v \in L \mid \neg\textsc{Contracts}(i, v)\}$), we again compact the elements which satisfy the given predicate. Membership tests can be accomplished via a boolean associated with each vertex, indicating whether or not that vertex is in the corresponding set. Union operations on lines 4 and 15 of Figure 3 can be implemented simply as concatenation, since the vertices stored within L and X are guaranteed to be disjoint.

3. Analysis

We analyze the expected work and parallel time of our construction and dynamic update algorithms presented in Sections 2.4 and 2.5. We calculate expectations over all random choices (random hash functions) in our algorithms. Our bounds are given in terms of $C(n)$, the parallel time required to compact an array of length n. We assume that the compaction algorithm used is work-efficient, meaning that the work required to compact n elements is $O(n)$.

Lemma 5. *For any forest (V, E), there exists $\beta \in (0, 1)$ such that $\mathbf{E}\left[\left|V^i\right|\right] \leq \beta^i |V|$, where V^i is the set of vertices remaining after i rounds of contraction.*

Proof. In Appendix. □

Lemma 6. *On a forest of n vertices, after $O(\log n)$ rounds of contraction, there are a constant number of vertices remaining with high probability.*

Proof. For any $c > 0$, consider round $r = (c+1) \cdot \log_{1/\beta}(n)$. By Lemma 5 and Markov's inequality, we have $\mathbf{P}\left[|V^r| \geq 1\right] \leq \beta^r n = n^{-c}$. □

3.1 Construction Algorithm

Theorem 1. *For a forest of n vertices, the work of the construction algorithm is $O(n)$ in expectation, and the parallel time is $O(\log(n)C(n))$ with high probability.*

Proof. At each round, the construction algorithm performs $O\left(\left|V^i\right|\right)$ work, and so the total work is $O\left(\sum_i \mathbf{E}\left[\left|V^i\right|\right]\right)$ in expectation. By Lemma 5, this is $O(|V|) = O(n)$. By Lemma 6, there are $O(\log n)$ rounds with high probability. At each round, the algorithm requires $O(C(n))$ parallel time for compaction. Therefore the parallel time is $O(\log(n)C(n))$ with high probability. □

3.2 Dynamic Update Algorithm

Consider a forest $F = (V, E)$ and the corresponding contraction data structure obtained by running the construction algorithm. We apply the changes $((V^-, E^-), (V^+, E^+))$ in order to obtain a contraction data structure for the forest $G = ((V \setminus V^-) \cup V^+, (E \setminus E^-) \cup E^+)$. We are interested in bounding the work and parallel time of in terms of $n = |V|$ and $m = |V^-| + |V^+| + |E^-| + |E^+|$. Note that the number of vertices in G is upper bounded by $n + m$.

3.2.1 Bounding the Number of Affected Vertices

Let L^i and X^i be the inputs to PROPAGATE at round i. Let $A^i = L^i \cup \{v : (v, \_) \in X^i\}$. This set consists of all vertices which are considered affected by the dynamic update algorithm at round i. We begin by bounding the size of $|A^0|$.

Lemma 7. *We have $|A^0| \leq 3m$.*

Proof. Initally, any vertex incident upon an edge in $E^- \cup E^+$ is considered affected, producing at most $2m$ affected vertices. For each of these, their parent may also be affected due to a leaf-status change. Thus $|A^0| \leq 3m$. □

We say that an affected vertex u *spreads to* v in round i, if v was unaffected at round i and v becomes affected in round i in either of the following ways:

1. v is neighbor of u at round i and u is contracted in round i in either F or G, or
2. v is neighbor of u at round $i+1$ and the leaf status of u changes in round i, i.e., $\ell_F^{i+1}(v) \neq \ell_G^{i+1}(v)$.

Let $k = |A^0|$. For each of F and G, we now inductively construct k disjoint sets for each round i, labeled $A_1^i, A_2^i, \ldots A_k^i$. These sets will form a partition of A^i.

Begin by arbitrarily partitioning A^0 into k singleton sets, and let $A_1^0, \ldots, A_k^0$ be these singleton sets. (In other words, each affected vertex in A^0 is assigned a unique number $1 \leq j \leq k$, and is then placed in A_j^0.)

Given sets $A_1^i, \ldots, A_k^i$, we construct sets $A_1^{i+1}, \ldots, A_k^{i+1}$ as follows. Consider some $v \in A^{i+1} \setminus A^i$. By Lemmas 3 and 4, there must exist at least one $u \in A^i$ such that u spreads to v. Since there could be many of these, let $S^i(v)$ be the set of vertices which spread to v in round i. Define

$$j^i(v) = \begin{cases} j, & \text{if } v \in A_j^i \\ \min_{u \in S^i(v)} \left(j \text{ where } u \in A_j^i\right), & \text{o.w.} \end{cases}$$

(In other words, $j^i(v)$ is v's set identifier if v is affected at round i, or otherwise the minimum set identifier j such that a vertex from A_j^i spread to v in round i). We can then produce the following for each $1 \leq j \leq k$:

$$A_j^{i+1} = \{v \in A^{i+1} \mid j^i(v) = j\}$$

Informally, each affected vertex from round i which stays affected also stays in the same place, and each newly affected vertex picks a set to join based on which vertices spread to it.

We say that a vertex v is a *frontier* at round i if v is affected at round i and at least one of its neighbors in either F or G is unaffected at round i. It is easy to show that any frontier at any round is alive in both forests and has the same set of unaffected neighbors in both at that round (thus, the set of frontier vertices at any round is the same in both forests). It is also easy to show that if a vertex v spreads to some other vertex in round i, then v is a frontier at round i. We show next that the number of frontier vertices within each A_j^i is bounded.

Lemma 8. *For any i, j, each of the following statements hold:*

1. *The subforests induced by A_j^i in each of F^i and G^i are trees.*
2. *A_j^i contains at most 2 frontier vertices.*
3. *$|A_j^{i+1} \setminus A_j^i| \leq 2$.*

Proof. Statement 1 follows from rake and compress preserving connectedness, and the fact that if u spreads to v then u and v are neighbors in both forests either at round i or round $i+1$. We prove statement 2 by induction on i, and conclude statement 3 in the process. At round 0, each A_j^0 clearly contains at most 1 frontier.

We now consider some A_j^i. Suppose there is a single frontier vertex v in A_j^i.

- If v compresses in one of the forests, then v will not be a frontier in A_j^{i+1}, but it will spread to at most two newly affected vertices which may be frontiers at round $i+1$. Thus the number of frontiers in A_j^{i+1} will be at most 2, and $|A_j^{i+1} \setminus A_j^i| \leq 2$.
- If v rakes in one of the forests, then we know v must also rake in the other forest (if not, then v could not be a frontier, since its parent would be affected). It spreads to one newly affected vertex (its parent) which may be a frontier at round $i+1$. Thus the number of frontiers in A_j^{i+1} will be at most 1, and $|A_j^{i+1} \setminus A_j^i| \leq 1$.

Now suppose there are two frontiers u and v in A_j^i. Due to statement 1 of the Lemma, each of these must have at least one affected neighbor at round i. Thus if either contracts, it will cease to be a frontier and may add at most one newly affected vertex to A_j^{i+1}, and this newly affected vertex might be a frontier at round $i+1$. The same can be said if either u or v spreads to a neighbor due to a leaf status change. Thus the number of frontiers either remains the same or decreases, and there are at most 2 newly affected vertices. Hence statements 2 and 3 of the Lemma hold. □

Now define $A_{F,j}^i = A_j^i \cap V_F^i$, that is, the set of vertices from A_j^i which are alive in F at round i. We define $A_{G,j}^i$ similarly for forest G.

Lemma 9. *For every i, j, we have $\mathbf{E}\left[\left|A_{F,j}^i\right|\right] \leq \frac{6}{1-\beta}$, and similarly for $A_{G,j}^i$.*

Proof. Consider $F^i_{A^i_{F,j}}$, the subforest induced by $A_{F,j}^i$ in F^i. By Lemma 8, this subforest is a tree, and has at most 2 frontier vertices.

By Lemma 5, if we applied one round of contraction to $F^i_{A^i_{F,j}}$, the expected number of vertices remaining would be at most $\beta \cdot \mathbf{E}\left[\left|A_{F,j}^i\right|\right]$. However, some of the vertices which contract in $F^i_{A^i_{F,j}}$ may not contract in F^i. Specifically, any vertex in $A_{F,j}^i$ which is a frontier or is the parent of a frontier might not contract. There are at most two frontier vertices and two

associated parents. By Lemma 8, two newly affected vertices might also be added. We also have $\left|A^0_{F,j}\right| = 1$. Therefore we conclude the following, which similarly holds for forest G:
$\mathbf{E}\left[\left|A^{i+1}_{F,j}\right|\right] \leq \beta\,\mathbf{E}\left[\left|A^i_{F,j}\right|\right] + 6 \leq 6\sum_{k=0}^{\infty}\beta^k = \frac{6}{1-\beta}$. □

Lemma 10. *For every i, we have $\mathbf{E}\left[\left|A^i\right|\right] \leq \frac{36}{1-\beta}m$.*

Proof. Follows from Lemmas 7 and 9, and $\left|A^i\right| \leq \sum_j \left|A^i_{F,j}\right| + \left|A^i_{G,j}\right|$. □

Theorem 2. *The work of dynamic update when applying m updates to a forest of size n is $O(m\log\frac{n+m}{m})$ in expectation.*

Proof. Let F be the given forest and G be the desired forest. The algorithm does $O\left(\left|A^i\right|\right)$ work at each round i, and therefore if $W = \sum_i \left|A^i\right|$, it suffices to bound $\mathbf{E}[W]$.

Since at least one vertex is either raked or finalized each round, we know that there are at most n rounds. Consider round $r = \log_{1/\beta}((n+m)/m)$, using the β given in Lemma 5. We now split the rounds into two groups: those that come before r and those that come after.

For $i < r$, we bound $\mathbf{E}\left[\left|A^i\right|\right]$ according to Lemma 10, yielding $O(rm) = O\left(m\log\frac{n+m}{m}\right)$ work.

Now consider $r \leq i < n$. For any i we know $\left|A^i\right| \leq \left|V^i_F\right| + \left|V^i_G\right|$, because each affected vertex must be alive in at least one of the two forests at that round. We can then apply the bound given in Lemma 5, and so

$$\begin{aligned}\sum_{r\leq i<n}\mathbf{E}\left[\left|A^i\right|\right] &\leq \sum_{r\leq i<n}\mathbf{E}\left[\left|V^i_F\right|\right] + \mathbf{E}\left[\left|V^i_G\right|\right]\\ &\leq \sum_{r\leq i<n}\beta^i n + \beta^i(n+m)\\ &= O((n+m)\beta^r) = O(m)\end{aligned}$$

Thus $\mathbf{E}[W] = O\left(m\log\frac{n+m}{m}\right) + O(m) = O\left(m\log\frac{n+m}{m}\right)$. □

Theorem 3. *The parallel time of dynamic update when applying m updates to a forest of size n is $O(\log(n+m)C(m))$ in expectation.*

Proof. By Lemma 6, there are $O\left(\log(n+m)\right)$ rounds of propagation with high probability (propagation proceeds until all verices are dead in *both* forests, and G has at most $n+m$ vertices). At each round, by Lemma 10, there are $O(m)$ affected vertices in expectation, and we require $O(C(m))$ time for compaction. Thus the parallel time is $O(\log(n+m)C(m))$ in expectation. □

4. Experimental Evaluation

We present an implementation and empirical evaluation of the algorithm. We use the C++ language and the PASL (Parallel Algorithm Scheduling Library), which provides a state-of-the-art work stealing scheduler [5].

Implementation. The implementation follows the algorithms described in Sections 2.4 and 2.5 but includes many more details.

The contraction data structure—which consists of parent pointers P, children sets C, and durations D—is repesented as a map from vertices v to lists of length $D[v]$. The i^{th} element of each list contains the parent pointer $P[i][v]$ and the set of children $C[i][v]$ for round i. This representation is efficient because our algorithms only require access to $P[i]$, $P[i+1]$, $C[i]$, and $C[i+1]$ in round i. This representation uses $O(n)$ space for a forest of n vertices.

In the algorithm specification, we use a compaction algorithm, which can be implemented in many ways. In practice, asymptotically slower but simpler algorithms fair well because of their smaller constant factors. Our implementation therefore uses a linear-work and logarithmic-time algorithm that relies on parallel prefix sums to determine the target index of each element and a parallel map to copy the input to the result array.

Algorithms compared. For the experimental evaluation, we implemented three algorithms. The first is a highly optimized implementation of the randomized tree-contraction algorithm of Miller and Reif [29]. We refer to this algorithm as *static*, and use it as a baseline in several comparisons. The second and third are our construction algorithm and our dynamic update algorithm, as described above and in Sections 2.4 and 2.5.

Experimental Setup. Our programs are compiled with GCC version 5.2, using `-O2 -march=native`. For the measurements, we considered an Ubuntu Linux machine with kernel v3.13.0-66-generic. Our benchmark machine has four Intel E7-4870 chips and 1Tb of RAM. Each chip has ten cores and shares a 30Mb L3 cache. Each core runs at 2.4Ghz and has 256Kb of L2 cache and 32Kb of L1 cache. Additionally, each core hosts two SMT threads, giving a total of eighty hardware threads. To avoid complications with hyperthreading, however, we did not use more than forty threads. To minimize noise from memory allocation, all of our algorithms preallocate all required memory. For each data point reported, we took averages over 3 runs. All reported execution times are in "seconds."

Input Generation. For our experiments, we generate a variety of trees where the number of children of each node is bounded by 4. As a special case, we also consider perfect binary trees. For generating random trees with varying levels of balance, we used a tree-builder algorithm similar to an algorithm from prior work [4]. The tree builder takes the size n of the input tree, the bound on the number of children t, and a chain factor f, $0 \leq f \leq 1$, and returns a tree where at least $f \cdot n$ of the vertices have degree two as long as $f \leq 1 - 2/n$. The *chain factor* approximately determines the ratio of the nodes with one child. When the chain factor is zero, the algorithm generates a balanced tree where all but possible one internal node has t children. When the chain factor is one, the algorithm generates a chain with only one leaf. In general, the tree becomes more unbalanced as the chain factor increases.

The tree-builder operates in two phases. The first phase builds perfectly balanced tree with $r = \max(n - \lceil nf \rceil, 2)$ vertices where all but possibly one internal node has t children (in our experiments we use $t = 4$). In the second phase, the algorithm adds the remaining $n - r$ vertices to the tree by randomly selecting an edge and "splitting" it into two edges by inserting a vertex in the middle, i.e., the randomly chosen edge (u, v) is split into (u, w) and (w, v). Since all of the vertices added in the second phase of the construction have degree two, the algorithm ensures that at least a fraction f of all vertices have degree two, as long as $f \leq 1 - 2/n$.

Construction Algorithm. Our construction algorithm is similar to the traditional tree-contraction algorithm except that it builds the contraction data structure. Since building the contraction data structure is a memory-intensive task, we expect our construction algorithm to incur some overhead relative to the static algorithm. We measured this overhead for a variety of forests with a range of chain factors, including 0.0, 0.3, 0.6, 1.0, and found it to be less than 2.5 times slower on average than the static algorithm. Our experiments (shown in Figures 10, 11, 12 and 13 in the Appendix) confirm that this factor remains constant over varying input sizes as established by our work efficiency bound (Theorem 1).

Figure 5 shows the speedup for our construction algorithm for inputs of size 4×10^6 with respect to the number of processors. This experiments shows that the algorithm scales reasonably well up to a point, though the growth rate decreases as the number of processors increase. Such "leveling off" of speedups is a classic symptom that many memory-bound parallel algorithms such as our contraction algorithm exhibit on modern architectures where memory bandwidth can become a bottleneck. Our experiments also show that speedups continue to grow reasonably well for inputs with higher chain-factors (e.g. 0.6 and 1.0). We attribute this to the slower contraction rate, which limits the load on the system.

Dynamic-Update Algorithm. Inserting or deleting an isolated vertex (with no incident edges) requires no significant work, we therefore we consider edge insertions and deletions to evaluate our dynamic algorithm. We consider two kinds of experiments.

- **Batch-insert test.** Generate an input forest (V, E) of n vertices, choose k random edges E', and use the dynamic update algorithm to insert the edges in E' to the input forest.
- **Batch-delete test.** Generate an input forest (V, E) of n vertices, choose k random edges E' and use the dynamic-update algorithm to delete edges in E' from the forest.

Figure 6 shows the run-time on 1 processor with respect to size of the inserted set of edges with an initial forest of $n = 10^6$ nodes. This plots shows that the run-time increases consistently with our work bound of $O(m \log \frac{n+m}{m})$ from Theorem 2.

Figure 7 shows the speedup of the dynamic update algorithm with different batch sizes (m). For these experiments, we choose trees with 10^6 nodes and chain-factor of 0.6. Results with other trees were similar. Since the dynamic-update algorithm performs little work for small changes, e.g., for constant number of changes the total work is $O(\lg n)$, and since the span of the algorithm is $\Omega(\lg n)$, for small changes, we don't expect to see speedups. As we increase the number of changes, howewer, speedups should increase. The results shown in Figure 7 are consistent with these outcomes predicted by our analysis. The experiments show that when a batch is small m is in the thousands, the speedups are small.[1] When the batch sizes are larger, we see more significant speedups, which increase with the batch size and the number of processors. As with the construction algorithm, the growth rates of speedups decrease as the number of processors increases because memory becomes a bottleneck.

[1] Several thousand of instructions leave little opportunity for practical parallelism on modern hardware.

An important advantage of a dynamic-parallel algorithm is that it can improve efficiency by using two orthogonal algorithmic techniques: dynamism to reduce work (and thus time) and parallelism to reduce time. Our final experiment compares our dynamic update algorithm to the highly optimized implementation of a static sequential tree contraction algorithm. Figure 8 shows the ratio of the run-time of the static sequential tree contraction algorithm to the dynamic update algorithm as a function of the number of processors for different batches of edges to be inserted. The size of the initial forest is 10^6 with chain factor of 0.6. Each data point (x, y) represents the ratio y of the time for the sequential static algorithm and the time for inserting x edges into the initial forest and performing an update by using our dynamic update algorithm. Ratio's greater that 1 indicate that the dynamic update is faster by that ratio. For small to moderate sized batches, the dynamic update algorithm runs several orders of magnitude faster than the sequential static algorithm (as much as 1000 times). As the batch size increases, the speedups decrease because the impact of dynamism reduces, the speedups, however, are still significant because as the input size increases, the opportunity for parallelism increases. For example, with larger batches consisting of 10^4 edges, the dynamic update algorithm runs approximately from 5 to 10 times faster than the static algorithm, when there is more than one core.

5. Related Work

Tree contraction and dynamic trees. Tree contraction, originally introduced by Miller and Reif [29, 30], has become a crucial technique for computing properties of trees in parallel. It has been studied extensively since its introduction and been used in many applications, e.g., expression evaluation, finding least-common ancestors, common subexpression evaluation, and computing various properties of graphs (e.g., [14, 15, 22, 24, 29, 30, 32, 34]).

Prior work has established a connection between the tree contraction and dynamic trees problem of Sleator and Tarjan [35] by showing that tree contraction can be dynamized to solve the dynamic trees problem [2, 4]. That work considers sequential updates only. In this paper, we show that this connection can be generalized to take advantage of parallelism by using a parallel version of the dynamization technique, called self-adjusting computation, used in the prior work.

Compaction. Compaction is a fundamental technique in parallel algorithm design, particularly for load balancing. It is well known that work-efficient compaction may be accomplished on a CREW PRAM via parallel prefix sums in $O(\log n)$ time. Fich et al show that compaction requires $\Omega(\log \log n)$ time on the CREW PRAM [16].

Linear approximate compaction (LAC), a popular variant on compaction, can achieve faster run times by allowing the output size to be a constant factor larger than necessary. LAC is closely related to hashing, since constructing a hash table of size linear in the number of stored keys effectively solves the LAC problem. Gil and Matias developed hashing techniques for the CRCW PRAM which run in $O(\log^* n \log \log n)$ time using an optimal number of processors [19]. Matias and Vishkin's CRCW PRAM algorithm further improves the bound to $O(\log^* n \log \log^* n)$ expected time [27]. With $O(n)$ processors, they are able to achieve $O(\log^* n)$ expected time. MacKenzie showed a matching lower bound of $\Omega(\log^* n)$ in expectation, proving that this result is tight [26].

Since our algorithm is a CREW algorithm, any of the aforementioned results can be used in our algorithm. In our implementation, we use the $O(\log n)$ time algorithm based on parallel prefix sums because of its simplicity and practical efficiency.

Other parallel dynamic algorithms. Parallel and dynamic algorithms are two important classes of algorithms that have, for the most part, been studied separately. Some exceptions are Jung and Mehlhorn's parallel algorithm for incremental (semi-dynamic) spanning trees [23], which allows inserting one new vertex into the input graph. Pawagi and Kaser propose a parallel (fully) dynamic algorithm that allows insertion and deletion of arbitrary number of vertices and edges as a batch [31]. Acar et al present a parallel dynamic algorithm for well-spaced points sets that allow insertion and deletion of arbitrary number of points simultaneously as a batch [6].

Self-Adjusting Computation. Our approach is based on the technique of self-adjusting computation for dynamizing static algorithms. Prior work developed the foundations of self-adjusting computation [1, 25] and applied to a number of problems including in computational geometry [3, 7, 8], and machine learning algorithms [9, 37]. All of this prior work assumes a sequential model of computation. There has been recent progress in generalizing self-adjusting computation to support parallelism [11, 13, 20**?**]. These results show that the self-adjusting computation techniques can work well in practical parallel systems but do not establish any bounds. There is only one paper that shows bounds for parallel change propagation on a geometric problem [6]; in this paper, we apply a similar design technique to the dynamic trees problem.

6. Conclusion

We present a fully general, work-efficient, and practical parallel dynamic algorithm for tree contraction, an important problem in parallel computing. Our parallel dynamic algorithm allows the input forest of size n to be modified by insertion or deletion of any number $m \geq 0$ of nodes and edges. It updates the result by performing work that is linear in m and at most logarithmic in n, and in span (parallel time) that is poly-logarithmic in n and m. We show that the algorithm is practical by presenting an implementation and evaluation. Our results suggest that parallel dynamic algorithms could combine the best features of dynamic and parallel algorithms: they allow inputs to be modified in an essentially arbitrary way and can handle such changes by performing sub-linear work in the size of the input and poly-logarithmic span (parallel time) in the size of the input and the changes.

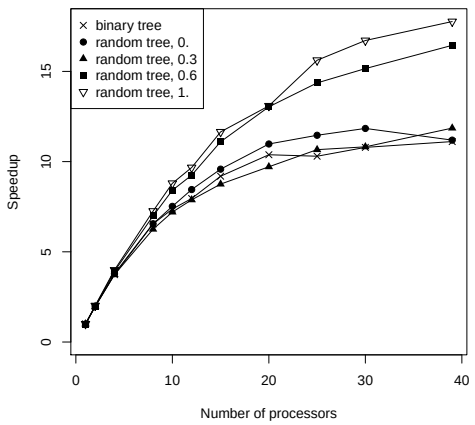

Figure 5. Speedup of our construction algorithm ($n = 4 \times 10^6$).

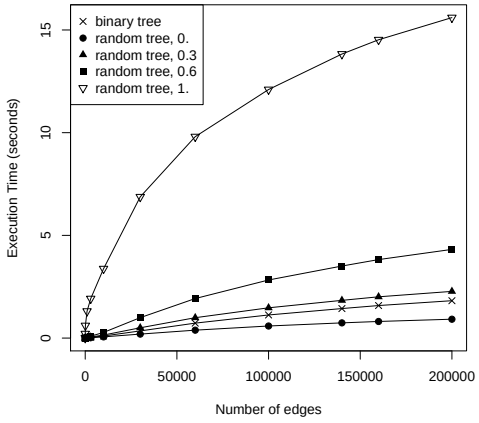

Figure 6. Dynamic-update time with 1 processor with respect to size of a batch of insertions ($n = 10^6$).

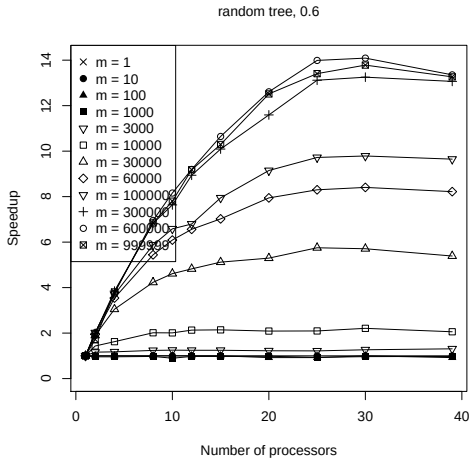

Figure 7. Self-speedup of parallel update algorithm with respect number of insertions on random tree ($n = 10^6$, chain factor 0.6).

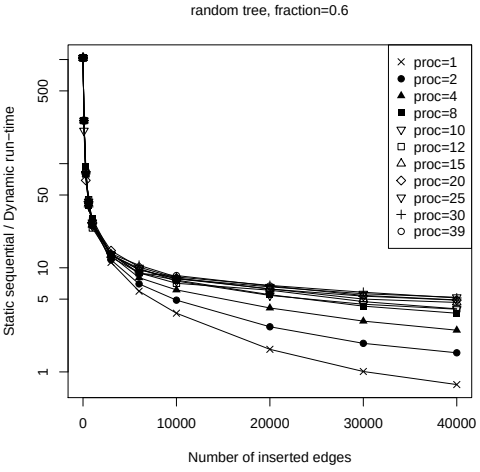

Figure 8. Dynamic update versus construction algorithms comparison with respect number of used processors on random tree ($n = 10^6$, chain factor 0.6).

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Prog. Lang. Sys.*, 28(6):990–1034, 2006.

[2] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 531–540, 2004.

[3] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Duru Türkoğlu. Robust kinetic convex hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, September 2008.

[4] Umut A. Acar, Guy E. Blelloch, and Jorge L. Vittes. An experimental analysis of change propagation in dynamic trees. In *Workshop on Algorithm Engineering and Experimentation*, 2005.

[5] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *PPoPP '13*, 2013.

[6] Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. Parallelism in dynamic well-spaced point sets. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2011.

[7] Umut A. Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoğlu. Dynamic well-spaced point sets. *Journal of Computational Geometry: Theory and Applications*, 2013.

[8] Umut A. Acar, Benoît Hudson, and Duru Türkoğlu. Kinetic mesh-refinement in 2D. In *SCG '11: Proceedings of the 27th Annual Symposium on Computational Geometry*, 2011.

[9] Umut A. Acar, Alexander Ihler, Ramgopal Mettu, and Özgür Sümer. Adaptive Bayesian inference. In *Neural Information Processing Systems (NIPS)*, 2007.

[10] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully-dynamic trees with top trees, 2003. The Computing Research Repository (CoRR)[cs.DS/0310065].

[11] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: MapReduce for incremental computations. In *ACM Symposium on Cloud Computing*, 2011.

[12] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[13] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011.

[14] Krzysztof Diks and Torben Hagerup. More general parallel tree contraction: Register allocation and broadcasting in a tree. *Theoretical Computer Science*, 203(1):3 – 29, 1998.

[15] David Eppstein and Zvi Galil. Parallel algorithmic techniques for combinational computation. *Annual review of computer science*, 3(1):233–283, 1988.

[16] Faith Fich, Mirosław Kowaluk, Mirosław Kutyłowski, Krzysztof Loryś, and Prabhakar Ragde. Retrieval of scattered information by erew, crew, and crcw prams. *computational complexity*, 5(2):113–131, 1995.

[17] Greg N. Frederickson. A data structure for dynamically maintaining rooted trees. *Journal Algorithms*, 24(1):37–65, 1997.

[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[19] Joseph Gil and Yossi Matias. Fast hashing on a pramdesigning by expectation. In *Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 271–280. Society for Industrial and Applied Mathematics, 1991.

[20] Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar Ghuloum. A proposal for parallel self-adjusting computation. In *DAMP '07: Declarative Aspects of Multicore Programming*, 2007.

[21] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[22] Joseph Jaja. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Company, 1992.

[23] Hermann Jung and Kurt Mehlhorn. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees. *Inf. Process. Lett.*, 27:227–236, April 1988.

[24] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 869–942. 1990.

[25] Ruy Ley-Wild, Umut A. Acar, and Matthew Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

[26] Philip D MacKenzie. Load balancing requires &ω (log* n) expected time. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 94–99. Society for Industrial and Applied Mathematics, 1992.

[27] Yossi Matias and Uzi Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 307–316, 1991.

[28] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, pages 487–489, 1985.

[29] Gary L. Miller and John H. Reif. Parallel tree contraction, part I: Fundamentals. *Advances in Computing Research*, 5:47–72, 1989.

[30] Gary L. Miller and John H. Reif. Parallel tree contraction, part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

[31] Shaunak Pawagi and Owen Kaser. Optimal parallel algorithms for multiple updates of minimum spanning trees. *Algorithmica*, 9:357–381, 1993.

[32] S. Rao Kosaraju and Arthur L. Delcher. *Optimal parallel evaluation of tree-structured computations by raking (extended abstract)*, pages 101–110. 1988.

[33] John H. Reif and Stephen R. Tate. Dynamic parallel tree contraction (extended abstract). In *SPAA*, pages 114–121, 1994.

[34] Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, pages 431–448, 2015.

[35] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[36] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[37] Özgür Sümer, Umut A. Acar, Alexander Ihler, and Ramgopal Mettu. Adaptive exact inference in graphical models. *Journal of Machine Learning*, 8:180–186, 2011.

[38] Robert Tarjan and Renato Werneck. Self-adjusting top trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005.

[39] Robert E. Tarjan. Dynamic trees as search trees via euler tours, applied to the network simplex algorithm. *Mathematical Programming*, 78:167–177, 1997.

[40] Robert E. Tarjan and Renato F. Werneck. Dynamic trees in practice. *J. Exp. Algorithmics*, 14:5:4.5–5:4.23, January 2010.

A. Proof of Lemma 5

The following argument resembles that of Theorem 2.1 in [29].

Proof. We begin by considering trees, and then extend the argument to forests.

Given a tree (V, E), consider the set V' of vertices after one round of contraction. We would like to show there exists $\beta \in (0, 1)$ such that $\mathbf{E}[|V'|] \leq \beta |V|$.

Partition V into P and $\{r\}$, where r is the root of the tree. Thus every vertex in P has a parent. Now partition P into P_0, P_1, and P_2 according to whether the vertex has 0 children, 1 child, or 2 or more children, respectively. Then partition P_1 into C_0, C_1, and C_2 similarly according to how many children the child of the vertex has. We now partition V into sets RAKE and COMP as follows:

$$\begin{aligned}\textsc{Rake} &= P_0 \cup C_0 \cup P_2 \cup \{r\},\\ \textsc{Comp} &= C_1 \cup C_2.\end{aligned}$$

Next we bound how many vertices from each of RAKE and COMP contract in one round.

- We have $|\textsc{Rake}| < 3|P_0| + 1 \leq 3|P_0|$ due to $|C_0| \leq |P_0|$ and $|P_2| < |P_0|$. Note that the "+1" accounts for the inclusion of r. Since all vertices in P_0 are raked, we have

$$|V' \cap \textsc{Rake}| = |\textsc{Rake}| - |P_0| \leq \frac{2}{3}|\textsc{Rake}|$$

- Each vertex in COMP compresses with probability $1/4$, therefore

$$\mathbf{E}[|V' \cap \textsc{Comp}|] = \frac{3}{4}|\textsc{Comp}|$$

We can now show $\beta = \frac{3}{4}$ holds for trees:

$$\begin{aligned}\frac{\mathbf{E}[|V'|]}{|V|} &= \frac{|V' \cap \textsc{Rake}| + \mathbf{E}[|V' \cap \textsc{Comp}|]}{|V|}\\ &\leq \frac{3}{4} \cdot \frac{|\textsc{Rake}| + |\textsc{Comp}|}{|V|} = \frac{3}{4}.\end{aligned}$$

We conclude $\mathbf{E}[|V'|] \leq \beta |V|$. Equivalently, for every i, we have $\mathbf{E}\left[\left|V^{i+1}\right|\right] \leq \beta \left|V^i\right|$, where V^i is the set of vertices after i rounds of contraction. Therefore $\mathbf{E}\left[\left|V^{i+1}\right|\right] \leq \beta\, \mathbf{E}\left[\left|V^i\right|\right]$. Expanding this recurrence, we have $\mathbf{E}\left[\left|V^i\right|\right] \leq \beta^i |V|$.

To extend the proof to forests, simply partition the forest into its constituent trees and apply the same argument to each tree individually. Due to linearity of expectation, summing over all trees yields the desired bounds. □

B. Additional Experiments

Figure 9 shows the run-time on 1 processor with respect to size of the deleted set of edges with an initial forest of $n = 10^6$ nodes. We observe that the run-time increases nearly linearly, consistently with our work bound of $O(m \log \frac{n+m}{m})$ from Theorem 2. Comparing against insertion (Figure 6), we see that the time for batch-deletions are significantly faster than batch-insertions. This is because in the case of insertions, the contraction data structure needs to be extended to include the new edges, whereas in batch-deletions, there are only deletions from the contraction graph, which are cheaper.

Figures 10, 11, 12, 13 show running time of static and construction algorithm on different types of trees with varying sizes. If we fix some type of tree, the difference between their running times grows linearly depending on size. This means that running times of static and construction algorithm differs by constant value for fixed type of tree. Thus we confirm the theoretical result (Theorem 1) about the work efficiency of the algorithm. The constant differences are given in the next table depending on the type of the tree:

type of tree	constant multiplier
perfect binary	1.02
chain factor 0.3	1.7
chain factor 0.6	1.9
chain factor 1.0	2.4

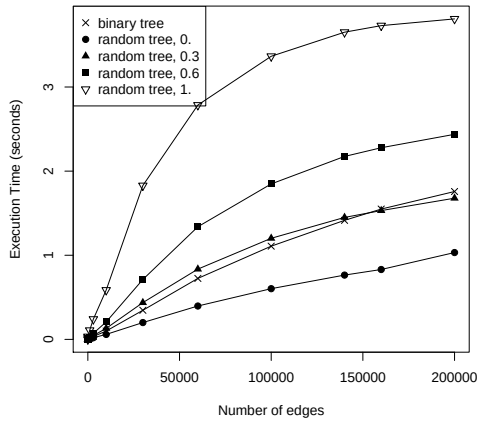

Figure 9. Run-time (y-axis) on 1 processor for a batch of deletions (x-axis) on various input forests with $n = 10^6$ nodes.

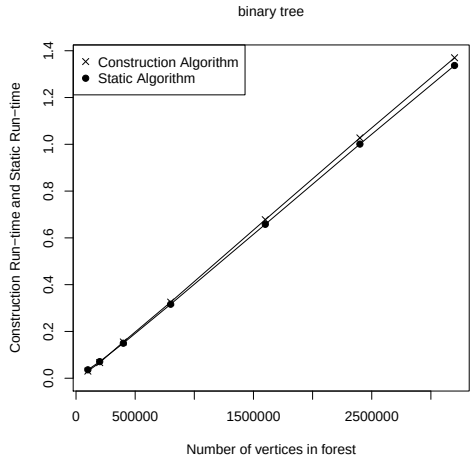

Figure 10. Our construction algorithm and the static algorithm on perfect binary trees of different sizes.

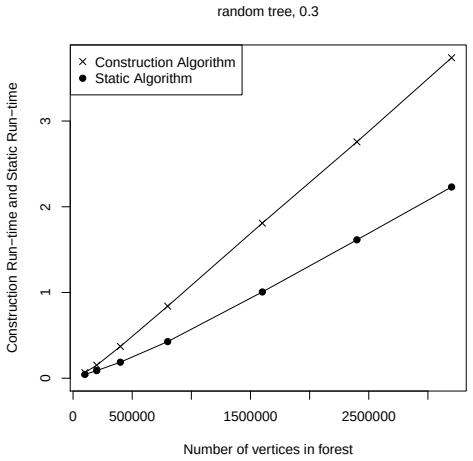

Figure 11. Our construction algorithm and the static algorithm on trees with chain factor 0.3 of different sizes.

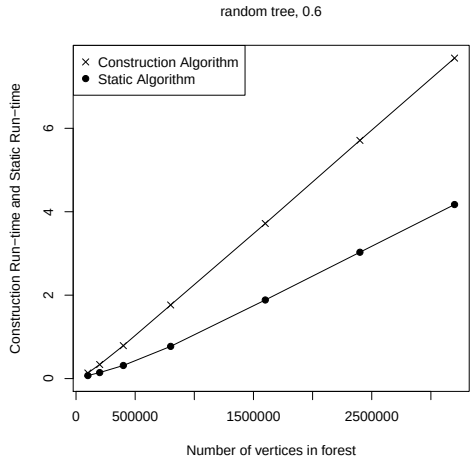

Figure 12. Our construction algorithm and the static algorithm on trees with chain factor 0.6 of different sizes.

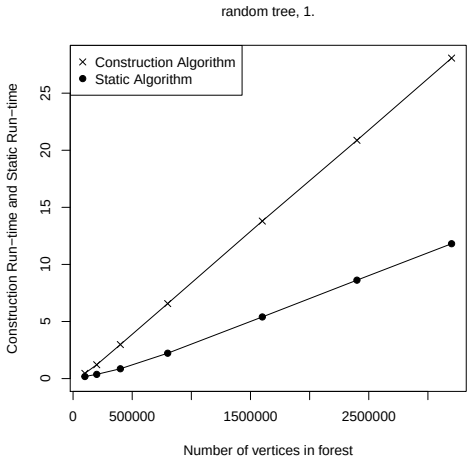

Figure 13. Our construction algorithm and the static algorithm on trees with chain factor 1 of different sizes.