Parallel Batch-Dynamic Trees via Change Propagation

Umut A. Acar
Carnegie Mellon University, Pittsburgh, PA, USA
umut@cs.cmu.edu

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

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

Laxman Dhulipala
Carnegie Mellon University, Pittsburgh, PA, USA
ldhulipa@cs.cmu.edu

Sam Westrick
Carnegie Mellon University, Pittsburgh, PA, USA
swestric@cs.cmu.edu

Abstract

The dynamic trees problem is to maintain a forest subject to edge insertions and deletions while facilitating queries such as connectivity, path weights, and subtree weights. Dynamic trees are a fundamental building block of a large number of graph algorithms. Although traditionally studied in the single-update setting, dynamic algorithms capable of supporting batches of updates are increasingly relevant today due to the emergence of rapidly evolving dynamic datasets. Since processing updates on a single processor is often unrealistic for large batches of updates, designing parallel batch-dynamic algorithms that achieve provably low span is important for many applications.

In this work, we design the first work-efficient parallel batch-dynamic algorithm for dynamic trees that is capable of supporting both path queries and subtree queries, as well as a variety of nonlocal queries. Previous work-efficient dynamic trees of Tseng et al. were only capable of handling subtree queries [*ALENEX'19*, (2019), pp. 92–106]. To achieve this, we propose a framework for algorithmically dynamizing static round-synchronous algorithms to obtain parallel batch-dynamic algorithms. In our framework, the algorithm designer can apply the technique to any suitably defined static algorithm. We then obtain theoretical guarantees for algorithms in our framework by defining the notion of a computation distance between two executions of the underlying algorithm.

Our dynamic trees algorithm is obtained by applying our dynamization framework to the parallel tree contraction algorithm of Miller and Reif [*FOCS'85*, (1985), pp. 478–489], and then performing a novel analysis of the computation distance of this algorithm under batch updates. We show that k updates can be performed in $O(k \log(1 + n/k))$ work in expectation, which matches the algorithm of Tseng et al. while providing support for a substantially larger number of queries and applications.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms; Theory of computation → Shared memory algorithms

Keywords and phrases Dynamic trees, Graph algorithms, Parallel algorithms, Dynamic algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.2

Related Version A full version is available at [2], `https://arxiv.org/abs/2002.05129`.

Funding This work was supported by NSF grants CCF-1901381, CCF-1910030 and CCF-1919223.

Acknowledgements The authors would like to thank Ticha Sethapakdi for helping with the figures.

1 Introduction

The dynamic trees problem, first posed by Sleator and Tarjan [27] is to maintain a forest of trees subject to the insertion and deletion of edges, also known as *links* and *cuts*. Dynamic trees are used as a building block in a multitude of applications, including maximum flows [27], dynamic connectivity and minimum spanning trees [11], and minimum cuts [19], making them a fruitful line of work with a rich history. There are a number of established sequential dynamic tree algorithms, including link-cut trees [27], top trees [28], Euler-tour trees [15], and rake-compress trees [5], all of which achieve $O(\log(n))$ time per operation.

Since they already perform such little work, there is often little to gain by processing single updates in parallel, hence parallel applications often process *batches* of updates. We are therefore concerned with the design of parallel *batch-dynamic* algorithms. Parallel batch-dynamic algorithms have been developed for several graph problems including incremental connectivity [26], Euler-Tour trees [29], and for fully dynamic connectivity [1]. Parallel batch-dynamic algorithms have also been recently studied in the MPC model [16, 10].

By applying batches it is often possible to obtain significant parallelism while preserving work efficiency. However, designing and implementing dynamic algorithms is difficult, and arguably even more so in the parallel setting.

The goals of this paper are twofold. First and foremost, we are interested in designing a parallel batch-dynamic algorithm for dynamic trees that supports a wide range of applications. On another level, based on the observation that parallel dynamic algorithms are usually quite complex and difficult to design, we are also interested in easing the design process of parallel batch-dynamic algorithms as a whole. To this end, we propose a framework for algorithmically dynamizing static parallel algorithms to obtain efficient parallel batch-dynamic algorithms. We then define a cost model that captures the *computation distance* between two executions of the static algorithm which allows us to bound the runtime of dynamic updates. There are several benefits of using algorithmic dynamization, some more theoretical some practical:

1. Proving correctness of a batch dynamized algorithm relies simply on the correctness of the parallel algorithm, which presumably has already been proven.
2. It is easy to implement different classes of updates. For example, for dynamic trees, in addition to links and cuts, it is very easy to update edge weights or vertex weights for supporting queries such as path length, subtree sums, or weighted diameter. One need only change the values of the weights and propagate.
3. Due to the simplicity of our approach, we believe it is likely to make it easier to program parallel batch-dynamic algorithms, and also result in practical implementations.

Using our algorithmic dynamization framework, we obtain a parallel batch-dynamic algorithm for rake-compress trees that generalizes the sequential data structure work efficiently without loss of generality. Specifically, our main contribution is the following theorem.

▶ **Theorem 1.** *The following operations can be supported on a bounded-degree dynamic tree of size n using the CRCW* PRAM*:*

- *Batch insertions and deletions of k edges in $O(k \log(1 + n/k))$ work in expectation and $O(\log(n) \log^*(n))$ span w.h.p.*
- *Batch connectivity, subtree sum, and path sum queries for batches of size k in $O(k \log(1 + n/k))$ work in expectation and $O(\log(n))$ span w.h.p.*
- *Independent parallel connectivity, subtree sum, path sum, diameter, lowest common ancestor, center, and median queries in $O(\log n)$ time per query w.h.p.*

Arbitrary-degree trees can be handled by transforming them into bounded degree trees using known techniques. We compare the capabilities of parallel rake-compress trees with other dynamic tree algorithms in Table 1. In summary, they support more operations than existing

Table 1 The known capabilities of various dynamic tree algorithms. Nonlocal queries are operations such as computing centers and medians. Our work extends rake-compress trees [5], which were previously only sequential, to also support parallel operations.

	Parallel Operations		Queries Supported		
	Updates	Queries	Path	Subtree	Nonlocal
Link-cut trees [27]			✓		
(Parallel) Euler-tour trees [15, 29]	✓	✓		✓	
Top trees [28]		✓	✓	✓	✓
Rake-compress trees [5]		✓	✓	✓	✓
Parallel rake-compress trees (this paper)	✓	✓	✓	✓	✓

parallel data structures, and support the same broad set of operations as existing non-parallel data structures. Theorem 1 is obtained by dynamizing the parallel tree contraction algorithm of Miller and Reif [20] and performing a novel analysis of the computation distance.

Standalone algorithms for dynamic parallel tree contraction have previously been proposed, but are inefficient and not fully general. In particular, Reif and Tate [25] give an algorithm for parallel dynamic tree contraction that can process a batch of k leaf insertions or deletions in $O(k\log(n))$ work. Unlike our algorithm, theirs is not work efficient, as it performs $\Omega(n\log(n))$ work for batches of size $\Omega(n)$, and it can only modify the tree at the leaves.

Lastly, as some evidence of the applicability of algorithmic dynamization, in the full version of this paper [2], we demonstrate two other applications of the technique. Specifically, we consider map-reduce based computations, and dynamic sequences with splitting and joining. To summarize, the main contributions of this paper are:

1. An algorithmic framework for dynamizing round-synchronous parallel algorithms, and a cost model for analyzing the performance of algorithms resulting from the framework
2. An analysis of the computation distance of Miller and Reif's tree contraction algorithm under batch edge insertions and deletions, which shows that it can be efficiently dynamized
3. The first work-efficient parallel algorithm for batch-dynamic trees that supports subtree queries, path queries, and nonlocal queries such as centers and medians.

Technical overview

A *round-synchronous* algorithm consists of a sequence of rounds, where a round executes in parallel across a set of processes, and each process runs a sequencial *round computation* reading and writing from shared memory and doing local computation. The round synchronous model is similar to Valiant's well-known Bulk Synchronous Parallel (BSP) model [30], except that communication is done via shared memory. Algorithmic dynamization works by running the round-synchronous algorithm while tracking all write-read dependences – i.e., a dependence from a write in one round to a read in a later round. Then, whenever a batch of changes are made to the input, *change propagation* propagates the changes through the original computation, only rerunning round computations if the values they read have changed. We note that depending on the algorithm, changes to the input could drastically change the underlying computation, introducing new dependencies, or invalidating old ones. Part of the novelty of this paper is bounding the work and span of this update process.

The idea of change propagation has been applied in the sequential setting and used to generate efficient dynamic algorithms [3, 4]. The general idea of parallel change propagation has also been used in various practical systems [9, 14, 7, 23, 24] but none of them have been analyzed theoretically.

To capture the cost of running the change propagation algorithm for a particular parallel algorithm and class of updates we define a *computational distance* between two computations, which corresponds to the total work of the round computations that differ in the two computations. The *input configuration* for a computation consists of the input I, stored in shared memory, and an initial set of processes P. We show the following bounds, where the *work* is the sum of the time of all round computations, and *span* is the sum over rounds of the maximum time of any round computation in that round.

▶ **Theorem 2.** *Given a round-synchronous algorithm A that with input configuration (I, P) does W work in R rounds and S span, then, on the CRCW* PRAM*,*

1. *the initial run of the algorithm with tracking takes $O(W)$ work in expectation and $O(S + R \log W)$ time w.h.p.,*
2. *running change propagation from input configuration (I, P) to configuration (I', P') takes $O(W_\Delta + R')$ work in expectation and $O(S' + R' \log W')$ time w.h.p., where W_Δ is the computation distance between the two configurations, and S',R',W' are the maximum span, rounds and work for the two configurations.*

We show that the work can be reduced to $O(W_\Delta)$, and that the $\log W$ and $\log W'$ terms can be reduced to $\log^* W$ when the round-synchronous algorithms have certain restrictions that are satisfied by all of our example algorithms, including our main result on dynamic trees. We also present similar results in other parallel models of computation.

With our dynamization framework and cost model, we develop an algorithm for dynamic trees that support a broad set of queries including subtree sums, path queries, lowest common ancestors, diameter, center, and median queries. This significantly improves over previous work on batch-dynamic Euler tour trees [29], which only support subtree sums.

Our dynamic trees algorithm is a parallel version of the sequential rake-compress tree (RC tree) data structure. Previous work showed that in the sequential setting, one can generate an RC tree (or forest) as a byproduct of Miller and Reif's tree contraction process, which supports the wide collection of queries mentioned above, all in logarithmic time, w.h.p. [5]. Our approach generalizes this sequential algorithm to allow for batches of edge insertions or deletions, work efficiently in parallel. The challenge is in analyzing the computation distance incurred by batch updates in the parallel batch-dynamic setting. In Section 4 we do just that, and obtain the following result:

▶ **Theorem 3.** *In the round synchronous model, Miller and Reif's tree contraction algorithm does $O(n)$ work in expectation and has $O(\log n)$ rounds and span w.h.p. Furthermore, given forests T with n vertices, and T' with k modifications to the edge list of T, the computation distance of the algorithm on the two inputs is $O(k \log(1 + n/k))$ in expectation.*

The bounds can then be plugged into Theorem 2 to show that a set of k edges can be inserted or deleted in a batch in $O(k \log(1 + n/k))$ work in expectation and $O(\log^2 n)$ span w.h.p. We show that the span can be improved to $O(\log n \log^* n)$ w.h.p. on the CRCW PRAM model. The last step in obtaining our dynamic trees framework is to plug the dynamized tree contraction algorithm into the RC trees framework [5] (see Section 5).

2 Preliminaries

2.1 Parallel Models

The parallel random access machine (PRAM) model is a classic parallel model with p processors that work in lock-step, connected by a parallel shared-memory [17]. In this paper we primarily consider the Concurrent-Read Concurrent-Write model (CRCW PRAM), where memory locations are allowed to be concurrently read and concurrently written to. If multiple writers write to the same location concurrently, we assume that an arbitrary writer wins. We analyze algorithms on the CRCW PRAM in terms of their *work* and *span*. The span (or parallel time) of an algorithm is the minimum running time achievable when arbitrarily many processors are available. The work is the product of the span and the number of processors.

The threaded random access machine (TRAM) is closely related to the PRAM, but more closely models current machines and programming paradigms [8]. In the binary forking TRAM (binary forking model for short), a process can *fork* another process to run in parallel, and can *join* to wait for all forked calls to complete. In the binary forking model, the *work* of an algorithm is the total number of instructions it performs, and the *span* is the longest chain of sequentially dependent instructions.

2.2 Parallel Primitives

The following parallel procedures are used throughout the paper. *Scan* takes as input an array A of length n, an associative binary operator $\oplus$, and an identity element $\perp$ such that $\perp \oplus x = x$ for any x, and returns the array $(\perp, \perp \oplus A[0], \perp \oplus A[0] \oplus A[1], \ldots, \perp \oplus_{i=0}^{n-2} A[i])$ as well as the overall sum, $\perp \oplus_{i=0}^{n-1} A[i]$. Scan takes $O(n)$ work and $O(\log n)$ span (assuming $\oplus$ takes $O(1)$ work) [17] on the CRCW PRAM, and in the binary forking model.

Filter takes an array A and a predicate f and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order as in A. Filter can be done in $O(n)$ work and $O(\log n)$ span on the CRCW PRAM (assuming f takes $O(1)$ work) [17], and in the binary forking model. The *Approximate Compaction* problem is similar to a Filter. It takes an array A and a predicate f and returns a new array containing $a \in A$ for which $f(a)$ is true where some of the entries in the returned array can have a null value. The total size of the returned array is at most a constant factor larger than the number of non-null elements. Gil et al. [12] describe a parallel approximate compaction algorithm that uses linear space and achieves $O(n)$ work and $O(\log^*(n))$ span w.h.p. on the CRCW PRAM.

A *semisort* takes an input array of elements, where each element has an associated key and reorders the elements so that elements with equal keys are contiguous. The purpose is to collect equal keys together, rather than sort them. Semisorting a sequence of length n can be performed in $O(n)$ expected work and $O(\log n)$ depth w.h.p. on the CRCW PRAM [13] and in the binary forking model [8], assuming access to a uniformly random hash function mapping keys to integers in the range $[1, n^{O(1)}]$.

3 Dynamization Framework

3.1 Round-synchronous algorithms

In this framework, we consider dynamizing algorithms that are *round synchronous*. The round synchronous framework encompasses a range of classic BSP [30] and PRAM algorithms. A round-synchronous algorithm consists of M *processes*, with process IDs bounded by $O(M)$. The algorithm performs sequential rounds in which each active process executes, in parallel, a *round computation*. At the end of a round, any processes can decide to *retire*, in which case they will no longer execute in any future round. The algorithm terminates once there are no remaining active processes – i.e., they have all retired. Given a fixed input, round-synchronous algorithms must perform deterministically. Note that this does not preclude us from implementing randomized algorithms (indeed, our dynamic trees algorithm is randomized), it just requires that we provide the source of randomness as an input to the algorithm, so that its behavior is identical if re-executed. An algorithm in the round synchronous framework is defined in terms of a procedure ComputeRound(r, p), which performs the computation of process p in round r. The initial run of a round-synchronous algorithm must specify the set P of initial process IDs.

Memory model

Processes in a round-synchronous algorithm may read and write to local memory that is not persisted across rounds. They also have access to a *shared memory.* The input to a round-synchronous algorithm is the initial contents of the shared memory. Round computations can read and write to shared memory with the condition that writes do not become visible until the end of the round. Reads can only access shared locations that have been written to, and shared locations can only be written to once, hence concurrent writes are not permitted. The contents of the shared memory at termination is considered to be the algorithm's output. Change propagation is driven by tracking all reads and writes to shared memory.

Pseudocode

We describe round-synchronous algorithms using the following primitives:
1. The **read** instruction reads the given shared memory locations and returns their values,
2. The **write** instruction writes the given value to the given shared memory location.
3. Processes may retire by invoking the **retire process** instruction.

Measures

The following measures will help us to analyse the efficiency of round-synchronous algorithms. For convenience, we define the *input configuration* of a round-synchronous algorithm as the pair (I, P), where I is the input to the algorithm (i.e. the initial state of shared memory) and P is the set of initial process IDs.

▶ **Definition 4** (Initial work, Round complexity, and Span)**.** *The* initial work *of a round-synchronous algorithm on some input configuration (I, P) is the sum of the work performed by all of the computations of each processes over all rounds when given that input. Its* round complexity *is the number of rounds that it performs, and its* span *is the sum of the maximum costs per round of the computations performed by each process.*

3.2 Change propagation

Given a round-synchronous algorithm, a *dynamic update* consists of a change to the input configuration, i.e. changing the contents of shared memory, and/or adding or deleting processes. The initial run and change propagation algorithms maintain the following data:
1. $R_{r,p}$, the memory locations read by process p in round r
2. $W_{r,p}$, the memory locations written by process p in round r
3. S_m, the set of round, process pairs that read memory location m
4. $X_{r,p}$, which is **true** if process p retired in round r

Algorithm 1 depicts the procedure for executing the initial run of a round-synchronous algorithm before making any dynamic updates.

To help formalize change propagation, we define the notion of an *affected computation*. The task of change propagation is to identify the affected computations and rerun them.

Algorithm 1 Initial run.

```
1: procedure Run($P$)
2:    local $r \leftarrow 0$
3:    while $P \neq \emptyset$ do
4:       for each process $p \in P$ do in parallel
5:          ComputeRound($r, p$)
6:          $R_{r,p} \leftarrow$ {memory locations read by $p$ in round $r$}
7:          $W_{r,p} \leftarrow$ {memory locations written to by $p$ in round $r$}
8:          $X_{r,p} \leftarrow$ (true if $p$ retired in round $r$ else false)
9:       for each $m \in \cup_{p \in P} R_{r,p}$ do in parallel
10:         $S_m \leftarrow S_m \cup \{(r,p) \mid m \in R_{r,p} \wedge p \in P\}$
11:      $P \leftarrow P \setminus \{p \in P : X_{r,p} = \textbf{true}\}$
12:      $r \leftarrow r + 1$
```

▶ **Definition 5** (Affected computation). *Given a round-synchronous algorithm A and two input configurations (I, P) and (I', P'), the* affected computations *are the round and process pairs (r, p) such that either:*

1. *process p runs in round r on one input configuration but not the other*
2. *process p runs in round r on both input configurations, but reads a variable from shared memory that has a different value in one configuration than the other*

The change propagation algorithm is depicted in Algorithm 2. It works by maintaining the affected computations as three disjoint sets, P, the set of processes that read a memory location that was rewritten, L, processes that outlived their previous self, i.e. that retired the last time they ran, but did not retire when re-executed, and D, processes that retired earlier than their previous self. First, at each round, the algorithm determines the set of computations that should become affected because of shared memory locations that were rewritten in the previous round (Lines 12–14). These are used to determine P, the set of affected computations to rerun this round (Line 15). To ensure correctness, the algorithm must then reset the reads that were performed by the computations that are no longer alive, or that will be reran, since the set of locations that they read may differ from last time (Lines 18–19). Lines 22–26 perform the re-execution of all processes that read a changed memory location, or that lived longer (did not retire) than in the previous configuration. The algorithm then subscribes the reads of these computations to the memory locations that they read (Lines 28–29). Finally, on Lines 32–36, the algorithm updates the set of changed memory locations (U), the set of computations that lived longer than their previous self (L) and the set of computations that retired earlier then their previous self (D).

3.3 Correctness

In this section, we sketch a proof of correctness of the change propagation algorithm (Algorithm 2). Intuitively, correctness is assured because of the write-once condition on global shared memory, which ensures that computations can not have their output overwritten, and hence do not need to be re-executed unless data that they depend on is modified.

▶ **Lemma 6.** *Given a dynamic update, re-executing only the affected computations for each round will result in the same output as re-executing all computations on the new input.*

Proof. Since by definition they read the same values, computations that are not affected, if re-executed, would produce the same output as they did the first time. Since all shared memory locations can only be written to once, values written by processes that are not

Algorithm 2 Change propagation.

```
// $U$ = sequence of memory locations that have been modified
// $P^+$ = sequence of new process IDs to create
// $P^-$ = sequence of process IDs to remove
procedure PROPAGATE($U$, $P^+$, $P^-$)
  local $D \leftarrow P^-$                          // Processes that died earlier than before
  local $L \leftarrow P^+$                          // Processes that lived longer than before
  local $A \leftarrow \emptyset$                    // Affected computations at each round
  local $r \leftarrow 0$
  while $U \neq \emptyset \vee D \neq \emptyset \vee L \neq \emptyset \vee \exists r' \geq r : (A_{r'} \neq \emptyset)$ do
    // Determine the computations that become affected
    // due to the newly updated memory locations $U$
    local $A' \leftarrow \cup_{m \in U} S_m$
    for each $r' \in \cup_{(r',p) \in A'} \{r'\}$ do in parallel
      $A_{r'} \leftarrow A_{r'} \cup \{p \mid (r', p) \in A'\}$
    local $P \leftarrow A_r \setminus D$                              // Processes to rerun
    // Forget the prior reads of all processes that are
    // now dead or will be rerun on this round
    for each $m \in \cup_{p \in P \cup D} R_{r,p}$ do in parallel
      $S_m \leftarrow S_m \setminus \{(r, p) \mid m \in R_{r,p} \wedge p \in P \cup D\}$
    local $X^{\text{prev}} = \{p \mapsto X_{r,p} \mid p \in P\}$
    // (Re)run all changed or newly live processes
    for each process $p$ in $P \cup L$ do in parallel
      COMPUTEROUND$(r, p)$
      $R_{r,p} \leftarrow$ {memory locations read by $p$ in round $r$}
      $W_{r,p} \leftarrow$ {memory locations written to by $p$ in round $r$}
      $X_{r,p} \leftarrow$ (true if $p$ retired in round $r$ else false)
    // Remember the reads performed by processes on this round
    for each $m \in \cup_{p \in P \cup L} R_{r,p}$ do in parallel
      $S_m \leftarrow S_m \cup \{(r, p) \mid m \in R_{r,p} \wedge p \in P \cup L\}$
    // Update the sets of changed memory locations,
    // newly live processes, and newly dead processes
    $U \leftarrow \cup_{p \in (P \cup L)} W_{r,p}$
    $L' \leftarrow \{p \in P \mid X^{\text{prev}}_p = \textbf{true} \wedge X_{r,p} = \textbf{false}\}$
    $L \leftarrow L \cup L' \setminus \{p \in L \mid X_{r,p} = \textbf{true}\}$
    $D' \leftarrow \{p \in P \mid X^{\text{prev}}_p = \textbf{false} \wedge X_{r,p} = \textbf{true}\}$
    $D \leftarrow D \cup D' \setminus \{p \in D \mid X^{\text{prev}}_p = \textbf{true}\}$
    $r \leftarrow r + 1$
```

re-executed can not have been overwritten, and hence it is safe to not re-execute them, as their output is preserved. Therefore re-executing only the affected computations will produce the same output as re-executing all computations. ◀

▶ **Theorem 7** (Consistency). *Given a dynamic update, change propagation correctly updates the output of the algorithm.*

Proof sketch. Follows from Lemma 6 and the fact that all reads and writes to global shared memory are tracked in Algorithm 2, and since global shared memory is the only method by which processes communicate, all affected computations are identified. ◀

3.4 Cost analysis

To analyze the work of change propagation, we need to formalize a notion of *computation distance*. Intuitively, the computation distance between two computations is the work performed by one and not the other. We then show that change propagation can efficiently re-execute the affected computations in work proportional to the computation distance.

▶ **Definition 8** (Computation distance). *Given a round-synchronous algorithm A and two input configurations, the* computation distance W_Δ *between them is the sum of the work performed by all of the affected computations with respect to both input configurations.*

▶ **Theorem 9.** *Given a round-synchronous algorithm A with input configuration (I, P) that does W work in R rounds and S span, then*

1. *the initial run of the algorithm with tracking takes $O(W)$ work in expectation and $O(S + R \cdot \log(W))$ span w.h.p.,*
2. *running change propagation on a dynamic update to the input configuration (I', P') takes $O(W_\Delta + R')$ work in expectation and $O(S' + R' \log(W'))$ span w.h.p., where S', R', W' are the maximum span, rounds, and work of the algorithm on the two input configurations,*

These bounds hold on the CRCW PRAM *and in the binary forking* TRAM *model.*

Proof. We begin by analyzing the initial run. By definition, all executions of the round computations, ComputeRound, take $O(W)$ work and $O(S)$ span in total, with at most an additional $O(\log(M)) = O(\log(W))$ span to perform the parallel for loop. We will show that all additional work can be charged to the round computations, and that at most an additional $O(\log(W))$ span overhead is incurred.

We observe that $R_{r,p}, W_{r,p}$ and $X_{r,p}$ are at most the size of the work performed by the corresponding computations, hence the cost of Lines 6 – 8 can be charged to the computation. The reader sets S_m can be implemented as dynamic arrays with lazy deletion (this will be discussed during change propagation). To append new elements to S_m (Line 10), we can use a semisort performing linear work in expectation to first bucket the shared memory locations in $\cup_{p \in P} R_{r,p}$, whose work can be charged to the corresponding computations that performed the reads. This adds an additional $O(\log(W))$ span w.h.p. since the number of reads is no more than W in total. Finally, removing retired computations from P (Line 11) requires a compaction operation. Since compaction takes linear work, it can be charged to the execution of the corresponding processes. The span of compaction is at most $O(\log(W))$.

Summing up, we showed that all additional work can be charged to the round computations, and the algorithm incurs at most $O(\log(W))$ additional span per round w.h.p. Hence the cost of the initial run is $O(W)$ work in expectation and $O(S + R \cdot \log(W))$ span w.h.p.

We now analyze the change propagation procedure (Algorithm 2). The core of the work is the re-execution of the affected readers on Line 23, which, by definition takes $O(W_\Delta)$ work, and $O(S')$ span, with at most $O(\log(W'))$ additional span to perform the parallel for loop. Since some rounds may have no affected computations, the algorithm could perform up to $O(R')$ additional work to process these rounds. We will show that all additional work can be charged to the affected computations, incurring at most an additional $O(\log(W'))$ span.

Lines 12 – 14 bucket the newly affected computations by round. This can be achieved with an expected linear work semisort and by maintaining the A_r sets as dynamic arrays. The work is chargeable to the affected computations and the span is at most $O(\log(W'))$ w.h.p. Computing the current set of affected computations (Line 15) requires a filter/compaction operation, whose work is charged to the affected computations and span is at most $O(\log(W'))$.

Updating the reader sets S_m (Line 19) can be done as follows. We maintain S_m as dynamic arrays with lazy deletion, meaning that we delete by marking the corresponding slot as empty. When more than half of the slots have been marked empty, we perform compaction, whose work is charged to the updates and whose span is at most $O(\log(W'))$. In order to perform deletions in constant time, we augment the set $R_{r,p}$ so that it remembers, for each entry m, the location of (r, p) in S_m. Therefore these updates take constant amortized work each (using a dynamic array), charged to the corresponding affected computations, and at most $O(\log(W'))$ span if a resize/compaction is triggered.

X^{prev} can be implemented as an array of size $|P|$, with work charged to the affected computations in P. As in the initial run, the cost of updating $R_{r,p}, W_{r,p}$ and $X_{r,p}$ can also be charged to the work performed by the affected computations.

Updating the reader sets S_m (Line 29) is a matter of appending to dynamic arrays, and, as mentioned earlier, remembering for each $m \in R_{r,p}$, the location of (r, p) in S_m. The work can be charged to the affected computations, and the span is at most $O(\log(W'))$.

Collecting the updated locations U (Line 32) can similarly be charged to the affected computations, and incurs no more than $O(\log(W'))$ span. On Lines 33 – 36, the sets L' and D' are computed by a compaction over P, whose work is charged to the affected computations in P. Updating L and D correspondingly requires a compaction operation, whose work is charged to the affected computations in L and D respectively. Each of these compactions costs $O(\log(W'))$ span.

We can finally conclude that all additional work performed by change propagation can be charged to the affected computations, and hence to the computation distance W_Δ, while incurring at most $O(\log(W'))$ additional span per round w.h.p. Therefore the total work performed by change propagation is $O(W_\Delta + R')$ in expectation and the span is $O(S' + R' \cdot \log(W'))$ w.h.p. ◀

We now show that for a special class of round-synchronous algorithms, the span overhead can be reduced. Our dynamic trees algorithm falls into this special case.

▶ **Definition 10.** *A* restricted round-synchronous *algorithm is a round-synchronous algorithm such that each round computation performs only a constant number of reads and writes, and each shared memory location is read only by a constant number of computations, and only in the round directly after it was written.*

▶ **Theorem 11.** *Given a restricted round-synchronous algorithm A with input configuration (I, P) that does W work in R rounds and S span, then*

1. *the initial run of the algorithm with tracking takes $O(W)$ work and $O(S + R\log^*(W))$ span w.h.p. on the CRCW PRAM and $O(S + R\log(W))$ span in the binary forking model,*
2. *change propagation on a dynamic update to the input configuration (I', P') takes $O(W_\Delta)$ work (in expectation on the CRCW PRAM), and $O(S' + R'\log^*(W'))$ span w.h.p. on the CRCW PRAM and $O(S' + R'\log(W'))$ span in the binary forking model, where S', R', W' are the maximum span, rounds, and work of the algorithm on the two input configurations.*

Proof sketch. Rather than recreate the entirety of the proof of Theorem 9, we simply sketch the differences. In essence, we obtain the result by removing the uses of scans, and semisorts, which were the main cause of the $O(\log(W'))$ span overhead and the randomized work. Instead, we rely only on (possibly approximate) compaction, which is only randomized on the CRCW PRAM. We also lose the R' term in the work since computations can only read from locations written in the previous round, and hence the set of rounds on which there exists an affected computation must be contiguous.

The main technique that we will make use of is the sparse array plus compaction technique. In situations where we wish to collect a set of items from each executed process, we would, in the unrestricted model, require a scan, which costs $O(\log(W'))$ span on the CRCW PRAM.

If each executed process, however, only produces a constant number of these items, we can allocate an array that is a constant size larger than the number of processes, and each process can write its set of items to a designated offset. We can then perform (possibly approximate) compaction on this array to obtain the desired set, with at most a constant factor additional blank entries. This takes $O(\log^*(W'))$ span w.h.p. on the CRCW PRAM, and $O(\log(W'))$ span in the binary forking model.

Maintaining S_m in the initial run and during change propagation is the first bottleneck, originally requiring a semisort. Since each computation performs a constant number of writes, we can collect the writes using the sparse array plus compaction technique. Since, in the restricted model, each modifiable will only be read by a constant number of readers, we can update S_m in constant time.

To compute the affected computations A_r also originally required a semisort, but in the restricted model, since all reads happen on the round directly after the write, no semisort is needed, since they will all have the same value of r. Collecting the affected computations from the written modifiables can also be achieved using the sparse array and compaction technique, using the fact that each computation wrote to a constant number of modifiables, and each modifiable is subsequently read by a constant number of computations. Additionally, A_r will be empty at the beginning of round r, so computing P requires only a compaction.

Lastly, collecting the updated locations U can also be performed using the sparse array and compaction technique. In summary, we can replace all originally $O(\log(W'))$ span operations with (approximate) compaction in the restricted setting, and hence we obtain the given span bounds since this takes $O(\log^*(W'))$ span w.h.p. on the CRCW PRAM, and $O(\log(W'))$ span in the binary forking model. ◀

▶ Remark 12 (Space usage). We do not formally specify an implementation of the memory model, but one simple way to achieve good space bounds is to use hashtables to implement global shared memory. Each write to a particular global shared memory location maps to an entry in the hashtable. When a round computation is invalidated during a dynamic update, its writes can be purged from the hashtable to free up space, preventing unbounded space blow up. Since the algorithm must also track the reads of each global shared memory location, using this implementation, the space usage is proportional to the number of shared memory reads and writes. In the restricted round-synchronous model, the number of reads must be proportional to the number of writes, and hence the space usage is proportional to the number of writes.

4 Dynamizing Tree Contraction

In this section, we show how to obtain a dynamic tree contraction algorithm by applying our dynamization technique to the static tree contraction algorithm of Miller and Reif [20]. In Section 5, we will show how to use this to obtain a parallel batch-dynamic trees framework.

Tree contraction

Tree contraction is the process of shrinking a tree down to a single vertex by repeatedly performing local contractions. Each local contraction deletes a vertex and merges its adjacent edges if it had degree two. Tree contraction has a number of useful applications, studied extensively in [21, 22, 5]. It can be used to perform various computations by associating data with edges and vertices and defining how data is accumulated during local contractions.

Various versions of tree contraction have been proposed depending on the specifics of local contractions. We consider an undirected variant of the randomized version proposed by Miller and Reif [20], which makes use of two operations: *rake* and *compress*. The former removes all nodes of degree one from the tree, except in the case of a pair of adjacent degree one vertices, in which case only one of them is removed by tiebreaking on the vertex IDs. The latter operation, compress, removes an independent set of vertices of degree two that are not adjacent to any vertex of degree one. Compressions are randomized with coin flips 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.

Input forests

The algorithms described here operate on undirected forests $F = (V, E)$, where V is a set of vertices, and E is a set of undirected edges. If $(u, v) \in E$, we say that u and v are *adjacent*, or that they are *neighbors*. A vertex with no neighbors is said to be *isolated*, and a vertex with one neighbor is called a *leaf*.

We 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 neighbors. We will explain how to handle arbitrary-degree trees momentarily.

The static algorithm

The static tree contraction algorithm (Algorithm 3) works in rounds, each of which takes a forest from the previous round as input and produces a new forest for the next round. On each round, some vertices may be *deleted*, in which case they are removed from the forest and are not present in all remaining rounds. Let $F^i = (V^i, E^i)$ be the forest after i rounds of contraction, and thus $F^0 = F$ is the input forest. 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 v was deleted in round i. There are three ways for a vertex to be deleted: it either *finalizes* (Line 32), *rakes* (Line 21), or *compresses* (Line 26). Finalization removes isolated vertices. Rake removes all leaves from the tree, with one special exception. If two leaves are adjacent, then to break symmetry and ensure that only one of them rakes, the one with the lower identifier rakes into the other (Line 8). Finally, compression removes an independent set of degree two vertices that are not adjacent to any degree one vertices, as in Miller and Reif's algorithm. The choice of which vertices are deleted in each round is made locally for each vertex based upon its own degree, the degrees of its neighbors, and coin flips for itself and its neighbors (Line 13). For coin flips, we assume a function $\textsc{Heads}(i, v)$ which indicates whether or not vertex v flipped heaps on round i. It is important that $\textsc{Heads}(i, v)$ is a function of both the vertex and the round number, as coin flips must be repeatable for change propagation to be correct.

The algorithm produces a *contraction data structure* which serves as a record of the contraction process. The contraction data structure is a tuple, (A, D), where $A[i][u]$ is a list of pairs containing the vertices adjacent to u in round i, and the positions of u in the adjacency lists of the adjacent vertices. $D[u]$ stores the round on which vertex u contracted. The algorithm also records $\text{leaf}[i][u]$, which is true if vertex u is a leaf at round i. An implementation of the tree contraction algorithm in our framework is shown in Algorithm 3.

Updates

We consider update operations that implement the interface of a batch-dynamic tree data structure. This requires supporting batches of links and cuts. A *link (insertion)* connects two trees in the forest by a newly inserted edge. A *cut (deletion)* deletes an edge from the forest, separating a single tree into two trees. We formally specify the interface for batch-dynamic trees and give a sample implementation of their operations in terms of the tree contraction data structure in the full version of this paper [2].

Algorithm 3 Tree contraction algorithm.

```
procedure ComputeRound($i, u$)
  local $((v_1, p_1), ..., (v_t, p_t))$, $\ell \leftarrow$ read($A[i][u]$, leaf[$i$][$u$])
  if $v_i = \perp \; \forall i$ then                         // A vertex with no neighbors finalizes
    DoFinalize($i, u$)
  else if $\ell$ then                                        // A leaf vertex rakes if its neighbor is
    local $(v, p) \leftarrow (v_i, p_i)$ such that $v_i \neq \perp$       // not a leaf, or if it has the lower ID
    local $\ell' \leftarrow$ read(leaf[$i$][$v$])
    if $\neg\ell' \vee u < v$ then DoRake($i, u, (v, p)$)
    else DoAlive($i, u, ((v_1, p_1), ..., (v_t, p_t))$)
  else                                                       // If the vertex has exactly two
    if $\exists (v, p), (v', p') : \{v_1, ..., v_t\} \setminus \{\perp\} = \{v, v'\}$ then   // neighbors, it will compress
      local $\ell', \ell'' \leftarrow$ read(leaf[$i$][$v$], leaf[$i$][$v'$])      // if neither of them are
      local $c \leftarrow$ Heads($i, u$) $\wedge$ $\neg$Heads($i, v$) $\wedge$ $\neg$Heads($i, v'$)   // leaves and it flips heads
      if $(\neg\ell' \wedge \neg\ell'' \wedge c)$ then              // and they both flip tails
        DoCompress($i, u, (v, p), (v', p')$)
      else
        DoAlive($i, u, ((v_1, p_1), ..., (v_t, p_t))$)
    else
      DoAlive($i, u, ((v_1, p_1), ..., (v_t, p_t))$)

procedure DoRake($i, u, (v, p)$)                             // When a vertex rakes, it replaces itself with
  write($A[i+1][v][p]$, $\perp$)                             // null ($\perp$) in its neighbor's adjacency list in
  write($D[u]$, $i$)                                         // in the next round
  retire process

procedure DoCompress($i, u, (v, p), (v', p')$)               // When a vertex compresses, it replaces itself
  write($A[i+1][v][p]$, $(v', p')$)                          // with its opposite neighbors in each neighbor's
  write($A[i+1][v'][p']$, $(v, p)$)                          // adjacency list in the next round
  write($D[u]$, $i$)
  retire process

procedure DoFinalize($i, u$)
  write($D[u]$, $i$)
  retire process

procedure DoAlive($i, u, ((v_1, p_1), ..., (v_t, p_t))$)     // If a vertex remains alive, it writes itself into
  local nonleaves $\leftarrow 0$                             // its neighbors' adjacency lists in the next
  for $j \leftarrow 1$ to $t$ do                             // round. It must also determine whether it
    if $v_j \neq \perp$ then                                 // it will be a leaf in the next round
      write($A[i+1][v_j][p_j]$, $(u, j)$)
      nonleaves += 1 - read(leaf[$i$][$v_j$])
    else
      write($A[i+1][u][j]$, $\perp$)
  write(leaf[$i+1$][$u$], nonleaves $= 1$)
```

Handling trees of arbitrary degree

To handle trees of arbitrary degree, we can split each vertex into a path of vertices, one for each of its neighbors. This technique is standard and has been described in [18], for example. This results in a tree of degree 3, with at most $O(n + m)$ vertices and $O(m)$ edges for an initial tree of n vertices and m edges. For edge-weighted trees, the additional edges can be given a suitable identity weight to preserve query values. It is simple to maintain such a transformation dynamically. For a batch insertion, a work-efficient semisort can be used to group each new neighbor by their endpoints, and then for each vertex, an appropriate number of new vertices can be added to the path. Batch deletion can be handled similarly.

4.1 Analysis

We now analyse the initial work, round, complexity, span, and computation distance of the tree contraction algorithm. This section is dedicated to proving the following theorem.

▶ **Theorem 13.** *Given a forest of n vertices, the initial work of tree contraction is $O(n)$ in expectation, the round complexity and the span is $O(\log(n))$ w.h.p. and the computation distance induced by updating k edges is $O(k\log(1+n/k))$ in expectation.*

Let $F = (V, E)$ be the set of initial vertices and edges of the input tree, and denote by $F^i = (V^i, E^i)$, the set of remaining (alive) vertices and edges at round i. We use the term *at round i* to denote the beginning of round i, and *in round i* to denote an event that occurs during round i. For some vertex v at round i, we denote the set of its adjacent vertices by $A^i(v)$, and its degree with $\delta^i(v) = |A^i(v)|$. A vertex is *isolated* at round i if $\delta^i(v) = 0$. 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_F is the set of edges in the forest F^i.

4.1.1 Analysis of construction

We first show that the static tree contraction algorithm is efficient. This argument is similar to Miller and Reif's argument in Theorem 2.1 of [21].

▶ **Lemma 14.** *For any forest (V, E), there exists $\beta \in (0,1)$ such that $\mathbf{E}\left[|V^i|\right] \le \beta^i |V|$, where V^i is the set of vertices remaining after i rounds of contraction.*

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 $\mathbf{E}[|V'|] \le \beta |V|$. If $|V| = 1$, then this is trivial since the vertex finalizes (it is deleted with probability 1). For $|V| \ge 2$, Consider the following sets, which partition the vertex set:

$H = \{v : \delta(v) \ge 3\}$
$L = \{v : \delta(v) = 1\}$
$C = \{v : \delta(v) = 2 \wedge \forall u \in A(v), u \notin L\}$
$C' = \{v : \delta(v) = 2\} \setminus C$

Note that at least half of the vertices in L must be deleted, since all leaves are deleted, except those that are adjacent to another leaf, in which case exactly one of the two is deleted. Also, in expectation, 1/8 of the vertices in C are deleted. Vertices in H and C' necessarily do not get deleted. Now, observe that $|C'| \le |L|$, since each vertex in C' is adjacent to a distinct leaf. Finally, we also have $|H| < |L|$, which follows from standard arguments about compact trees. Therefore in expectation,

$$\frac{1}{2}|L| + \frac{1}{8}|C| \ge \frac{1}{4}|L| + \frac{1}{8}|H| + \frac{1}{8}|C'| + \frac{1}{8}|C| \ge \frac{1}{8}|V|$$

vertices are deleted, and hence

$$\mathbf{E}[|V'|] \le \frac{7}{8}|V|.$$

Equivalently, for $\beta = \frac{7}{8}$, for every i, we have $\mathbf{E}\left[\left|V^{i+1}\right| \mid V_i\right] \le \beta \left|V^i\right|$, where V^i is the set of vertices after i rounds of contraction. Therefore $\mathbf{E}\left[\left|V^{i+1}\right|\right] \le \beta\, \mathbf{E}\left[\left|V^i\right|\right]$. Expanding this recurrence, we have $\mathbf{E}\left[\left|V^i\right|\right] \le \beta^i \left|V\right|$. 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. ◀

▶ **Lemma 15.** *On a forest of n vertices, after $O(\log n)$ rounds of contraction, there are no vertices remaining w.h.p.*

Proof. For any $c > 0$, consider round $r = (c+1) \cdot \log_{1/\beta}(n)$. By Lemma 14 and Markov's inequality, we have

$$\mathbf{P}\left[|V^r| \ge 1\right] \le \beta^r n = n^{-c}. \quad ◀$$

Proof of initial work, rounds, and span in Theorem 13

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 14, this is $O(|V|) = O(n)$. The round complexity and the span follow from Lemma 15. ◀

4.1.2 Analysis of dynamic updates

Intuitively, tree contraction is efficiently dynamizable due to the observation that, when a vertex locally makes a choice about whether or not to delete, it only needs to know who its neighbors are, and whether or not its neighbors are leaves. This motivates the definition of the *configuration* of a vertex v at round i, denoted $\kappa^i_F(v)$, defined as

$$\kappa^i_F(v) = \begin{cases} (\{(u, \ell^i_F(u)) : u \in A^i_F(v)\}), & \text{if } v \in V^i_F \\ \text{dead}, & \text{if } v \notin V^i_F, \end{cases}$$

where $\ell^i_F(u)$ indicates whether $\delta^i_F(u) = 1$ (the *leaf status* of u). Consider some input forest $F = (V, E)$, and let $F' = (V, (E \setminus E^-) \cup E^+)$ be the newly desired input after a batch cut with edges E^- and/or a batch-link with edges E^+. We say that a vertex v is *affected* at round i if $\kappa^i_F(v) \ne \kappa^i_{F'}(v)$.

▶ **Lemma 16.** *The execution in the tree contraction algorithm of process p at round r is an affected computation if and only if p is an affected vertex at round r.*

Proof. The code for ComputeRound for tree contraction reads only the neighbors, and corresponding leaf statuses, which are precisely the values encoded by the configuration. Hence if vertex p is alive in both forests the computation p is affected if and only if vertex p is affected. If instead p is dead in one forest but not the other, vertex p is affected, and the process p will have retired in one computation but not the other, and hence it will be an affected computation. Otherwise, if vertex p is dead in both forests, then the process p will have retired in both computations, and hence be unaffected. ◀

This means that we can bound the computation distance by bounding the number of affected vertices. First, we show that vertices that are not affected at round i have nice properties.

▶ **Lemma 17.** *If v is unaffected at round i, then either v is dead at round i in both F and F', or v is adjacent to the same set of vertices in both.*

Proof. Follows directly from $\kappa^i_F(v) = \kappa^i_{F'}(v)$. ◀

▶ **Lemma 18.** *If v is unaffected at round i, then v is deleted in round i of F if and only if v is also deleted in round i of F', and in the same manner (finalize, rake, or compress).*

Proof. Suppose that v is unaffected at round i. Then by definition it has the same neighbors at round i in both F and F'. The contraction process depends only on the neighbors of the vertex, and hence proceeds identically in both cases. ◀

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 many ways.

▶ **Lemma 19.** *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 was deleted in either F^i or $(F')^i$.*
2. *v has an affected neighbor u at round $i+1$ where $\ell_F^{i+1}(u) \neq \ell_{F'}^{i+1}(u)$.*

Proof. First, note that since v becomes affected, we know v does not get deleted, and furthermore that v has at least one child at round i. If v were to be deleted, then by Lemma 18 it would do so in both forests, leading it to being dead in both forests at the next round and therefore unaffected. If v were to have no children, then v would rake, but we just argued that v cannot be deleted.

Suppose that the only neighbors of v which are deleted 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 20.** *If v is not deleted in either forest in round i and $\ell_F^{i+1}(v) \neq \ell_{F'}^{i+1}(v)$, then v is affected at round i.*

Proof. Suppose v is not affected at round i. If none of v's neighbors are deleted in this round in either forest, then $\ell_F^{i+1}(v) = \ell_{F'}^{i+1}(v)$, a contradiction. Otherwise, if the only neighbors that are deleted do so via a compression, since compression preserves the degree of its endpoints, we will also have $\ell_F^{i+1}(v) = \ell_{F'}^{i+1}(v)$ and thus a contradiction. So, we consider the case of one of v's children raking. However, since v is unaffected, we know $\ell_F^i(u) = \ell_{F'}^i(u)$ 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 $\ell_F^{i+1}(v) = \ell_{F'}^{i+1}(v)$. Therefore v must be affected at round i. ◀

Lemmas 19 and 20 give us tools to bound the number of affected vertices for a consecutive round of contraction: each affected vertex that is deleted affects its neighbors, and each affected vertex whose leaf status is different in the two forests at the next round affects its neighbor. This strategy actually overestimates which vertices are affected, since case 1 of Lemma 19 does not necessarily imply that v is affected at the next round. We wish to show that the number of affected vertices at each round is not large. Intuitively, we will show that the number of affected vertices grows only arithmetically in each round, while shrinking geometrically, which implies that their total number can never grow too large. Let A^i denote the set of affected vertices at round i. We begin by bounding the size of $|A^0|$.

▶ **Lemma 21.** *For a batch update of size k, we have $|A^0| \leq 3k$.*

Proof. The computation for a given vertex u at most reads its neighbors, and if it has a single neighbor, its neighbor's leaf status. Therefore, the addition/deletion of a single edge affects at most 3 vertices at round 0. Hence $|A^0| \leq 3k$. ◀

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

Let $s = |A^0|$. For each of F and F', we now inductively construct s disjoint sets for each round i, labeled $A_1^i, A_2^i, \dots A_s^i$. These sets will form a partition of A^i. First, arbitrarily partition A^0 into s singleton sets, and let $A_1^0, \dots, A_s^0$ be these singleton sets. In other words, each affected vertex in A^0 is assigned a unique number $1 \leq j \leq s$, and is then placed in A_j^0.

Given sets $A_1^i, \dots, A_s^i$, we construct sets $A_1^{i+1}, \dots, A_s^{i+1}$ as follows. Consider some $v \in A^{i+1} \setminus A^i$. By Lemmas 19 and 20, 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

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

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 F' 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, and 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 22.** *For any i, j, each of the following statements hold:*

1. *The subforests induced by A_j^i in each of F^i and $(F')^i$ 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 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 v must also rake in the other forest (if not, then v could not be a frontier, since its neighbor would be affected). It spreads to one newly affected vertex (its neighbor) 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 is deleted, 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 $A_{F',j}^i$ similarly for forest F'.

▶ **Lemma 23.** *For every i, j, we have*

$$\mathbf{E}\left[|A_{F,j}^i|\right] \leq \frac{6}{1-\beta},$$

and similarly for $A_{F',j}^i$.

Proof. Let $F_{A,j}^i$ denote the subforest induced by $A_{F,j}^i$ in F^i. By Lemma 22, this subforest is a tree, and has at most 2 frontier vertices. By Lemma 14, if we applied one round of contraction to $F_{A,j}^i$, the expected number of vertices remaining would be at most $\beta \cdot \mathbf{E}[|A_{F,j}^i|]$. However, some of the vertices that are deleted in $F_{A,j}^i$ may not be deleted in F^i. Specifically, any vertex in $A_{F,j}^i$ which is a frontier or is the neighbor that spread to a frontier might not be deleted. There are at most two frontier vertices and two associated neighbors. By Lemma 22, two newly affected vertices might also be added. We also have $|A_{F,j}^0| = 1$. Therefore we conclude the following, which similarly holds for forest F':

$$\mathbf{E}\left[\left|A_{F,j}^{i+1}\right|\right] \leq \beta\,\mathbf{E}\left[|A_{F,j}^i|\right] + 6 \leq 6\sum_{r=0}^{\infty} \beta^r = \frac{6}{1-\beta}.$$ ◀

▶ **Lemma 24.** *For a batch update of size k, we have for every i,*

$$\mathbf{E}\left[|A^i|\right] \leq \frac{36}{1-\beta}k.$$

Proof. Follows from Lemmas 21 and 23, and the fact that

$$|A^i| \leq \sum_{j=1}^{s} \left(|A_{F,j}^i| + |A_{F',j}^i|\right).$$ ◀

Proof of computation distance in Theorem 13

Proof. Let F be the given forest and F' be the desired forest. Since each process of tree contraction does constant work each round, Lemma 16 implies that the algorithm does $O\left(|A^i|\right)$ work at each round i, so $W_\Delta = \sum_i |A^i|$.

Since at least one vertex is either raked or finalized each round, we know that there are at most n rounds. Consider round $r = \log_{1/\beta}(1 + n/k)$, using the β given in Lemma 14. We now split the rounds into two groups: those that come before r and those that come after.

For $i < r$, we bound $\mathbf{E}\left[|A^i|\right]$ according to Lemma 24, yielding

$$\sum_{i<r} \mathbf{E}\left[|A^i|\right] = O(rk) = O\left(k \log\left(1 + \frac{n}{k}\right)\right)$$

work. Now consider $r \le i < n$. For any i we know $\left|A^i\right| \le \left|V_F^i\right| + \left|V_{F'}^i\right|$, 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 14, and so

$$\begin{aligned}\sum_{r \le i < n} \mathbf{E}\left[\left|A^i\right|\right] &\le \sum_{r \le i < n} \left(\mathbf{E}\left[\left|V_F^i\right|\right] + \mathbf{E}\left[\left|V_{F'}^i\right|\right]\right) \\ &\le \sum_{r \le i < n} \left(\beta^i n + \beta^i n\right) \\ &= O(n\beta^r) \\ &= O\left(\frac{nk}{n+k}\right) \\ &= O(k),\end{aligned}$$

and thus

$$\mathbf{E}\left[W_\Delta\right] = O\left(k \log\left(1 + \frac{n}{k}\right)\right) + O(k) = O\left(k \log\left(1 + \frac{n}{k}\right)\right). \qquad \blacktriangleleft$$

5 Parallel Rake-compress Trees

Dynamic trees typically provide support for dynamic connectivity queries. Most dynamic tree data structures also support some form of augmented value query. For example, Link-cut trees [27] support root-to-vertex path queries, and Euler-tour trees [15] support subtree sum queries. Top trees [28, 6] support both path and subtree queries, as well as nonlocal queries such as centers and medians, but no parallelization of them is known. The only existing parallel batch-dynamic tree data structure is that of Tseng et al. [29], which is based on Euler-tour trees, and hence only handles subtree queries.

Rake-compress trees [5] (RC trees) are another sequential dynamic trees data structure, based on tree contraction, and have also been shown to be capable of handling both path and subtree queries, as well as nonlocal queries, all in $O(\log(n))$ time. In this section, we will explain how our parallel batch-dynamic algorithm for tree contraction can be used to derive a parallel batch-dynamic version of RC trees, leading to the first work-efficient algorithm for batch-dynamic trees that can handle this wide range of queries. We use a slightly different set of definitions than the original presentation of RC trees in [5], which correct some subtle corner cases and simplify the exposition, although the resulting data structure is equivalent. All of the query algorithms for sequential RC trees therefore work on our parallel version.

Contraction and clusters

RC trees are based on the idea that the tree contraction process can be interpreted as a recursive clustering of the original tree. Formally, a *cluster* is a connected subset of vertices and edges of the original tree. Note, importantly, that a cluster may contain an edge without containing both of its endpoints. The *boundary* vertices of a cluster C are the vertices $v \notin C$ that are adjacent to an edge $e \in C$. The *degree* of a cluster is the number of boundary vertices of that cluster. The vertices and edges of the original tree form the base clusters. Clusters are merged using the following simple rule: Whenever a vertex v is deleted, all of the clusters that have v as a boundary vertex are merged with the base cluster containing v. This implies that all clusters formed will have degree at most two. A cluster of degree zero is called a *nullary* cluster, a cluster of degree one a *unary* cluster, and a cluster of degree two a *binary* cluster. All non-base clusters have a unique *representative vertex*, which corresponds to the vertex that was deleted to form it. The full version of this paper [2] provides additional details and some diagrams that explain what each kind of cluster looks like.

5.1 Building and maintaining RC trees

Given a tree and an execution of the tree contraction algorithm, the RC tree consists of *nodes* which correspond to the clusters formed by the contraction process. The children of a node are the nodes corresponding to the clusters that merged together to form it. An example tree, a clustering, and the corresponding RC tree are depicted in Figure 1. Note that in the case of a disconnected forest, the RC tree will have multiple roots.

We will sketch here how to maintain an RC tree subject to batch-dynamic updates in parallel using our algorithm for parallel batch-dynamic tree contraction. This requires just two simple augmentations to the tree contraction algorithm. Recall that tree contraction (Algorithm 3) maintains an adjacency list for each vertex at each round. Whenever a neighbor u of a vertex v rakes into v, the process u writes a null value into the corresponding position in v's adjacency list. This process can be augmented to also write, in addition to the null value, the identity of the vertex that just raked. Second, when storing the data for a neighboring edge in a vertex's adjacency list, we additionally write the name of the representative vertex if that edge corresponds to a compression, or null if the edge is an edge of the original tree. The RC tree can then be inferred using this augmented data as follows.

1. Given any cluster C with representative v, its unary children can be determined by looking at the vertices that raked into v. The children are precisely the unary clusters represented by these vertices. For the final cluster, these are its only children.
2. Given a binary or unary cluster C with representative v, its binary children can be determined by inspecting v's adjacency list at the moment it was deleted. The binary clusters corresponding to the edges adjacent to v at its time of death are the binary children of the cluster C.

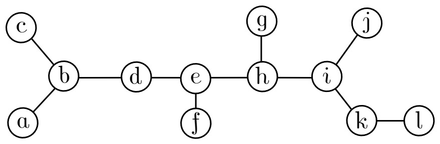
It then suffices to observe that this information about the clusters can be recorded during the contraction process. By employing change propagation, the RC tree can therefore be maintained subject to batch-dynamic updates. Since each cluster consists of a constant amount of information, this can be done in the same work and span bounds as the tree contraction algorithm. We therefore have the following result.

▶ **Theorem 25.** *We can maintain a rake-compress tree of a tree on n vertices subject to batch insertions and batch deletions of size k in $O(k \log(1 + n/k))$ work in expectation and $O(\log^2(n))$ span per update w.h.p. The span can be improved to $O(\log(n) \log^*(n))$ w.h.p. on the CRCW* PRAM.

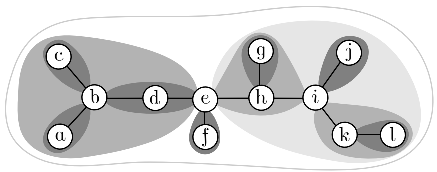
5.2 Applications

Most kinds of queries assume that the vertices and/or edges of the input tree are annotated with data, such as weights or labels. In order to support queries, each cluster is annotated with some additional information. The algorithm must then specify how to combine the data from multiple constituent clusters whenever a set of clusters merge. These annotations are generated during the tree contraction algorithm, and are therefore available for querying immediately after performing an update.

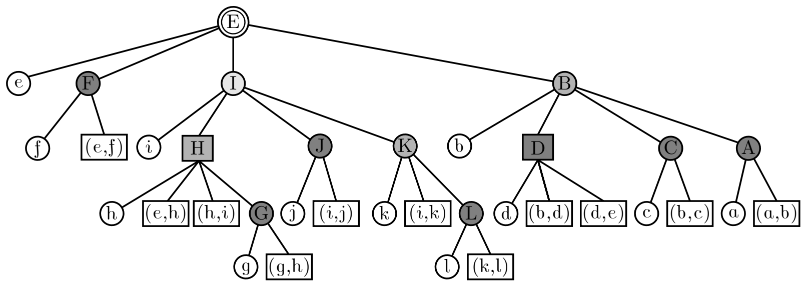
Once the clusters are annotated with the necessary data, the queries themselves typically perform a bottom-up or top-down traversal of the RC tree, or possibly in the case of more complicated queries, a combination of both. A variety of queries is described in [5].



(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 label of their representative vertex.

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

Batch queries

We can also implement *batch queries*, in which we answer k queries simultaneously in $O(k \log(1 + n/k))$ work in expectation and $O(\log(n))$ span w.h.p. This improves upon the work bound of $O(k \log(n))$ obtained by simply running independent queries in parallel. The idea is to detect when multiple traversals would intersect, and to eliminate redundant work that they would perform. An example in which this technique is applicable is finding a representative vertex of a connected component. When traversing upwards, if multiple query paths intersect, then only one proceeds up the tree and subsequently brings the answer back down for the other ones. The following theorem is the main tool that we can use for analyzing batch queries. The proof is similar to that of the computation distance in Theorem 13, and can be found in the full version of this paper [2].

▶ **Theorem 26.** *Given a tree on n vertices and a corresponding RC tree, k root-to-leaf paths in the RC tree touch $O(k \log(1 + n/k))$ distinct RC tree nodes in expectation.*

In the full version of this paper [2], we will show that batch connectivity, subtree sum, and path sum queries given batches of size k can be answered in $O(k \log(1 + n/k))$ work in expectation and $O(\log(n))$ span w.h.p.

6 Conclusion

In this paper we showed that we can obtain work-efficient parallel batch-dynamic algorithms by applying an algorithmic dynamization technique to corresponding static algorithms. Using this technique, we obtained the first work-efficient parallel algorithm for batch-dynamic trees that supports more than just subtree queries. Our framework also demonstrates the broad benefits of algorithmic dynamization; much of the complexity of designing parallel batch-dynamic algorithms by hand is removed, since the static algorithms are usually simpler than their dynamic counterparts. We note that although the round synchronous model captures a very broad class of algorithms, the breadth of algorithms suitable for dynamization is less clear. To be suitable for dynamization, an algorithm additionally needs to have small computational distance between small input changes. As some evidence of broad applicability, however, the practical systems mentioned in the technical overview of the introduction have been applied broadly and successfully – again without any theoretical justification, yet.

References

1 Umut A Acar, Daniel Anderson, Guy E Blelloch, and Laxman Dhulipala. Parallel batch-dynamic graph connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

2 Umut A Acar, Daniel Anderson, Guy E Blelloch, Laxman Dhulipala, and Sam Westrick. Parallel batch-dynamic trees via change propagation. *arXiv preprint*, 2020. `arXiv:2002.05129`.

3 Umut A Acar, Guy E Blelloch, and Robert Harper. Adaptive functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

4 Umut A Acar, Guy E Blelloch, Robert Harper, Jorge L Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

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

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

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

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

9 Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

10 Laxman Dhulipala, David Durfee, Janardhan Kulkarni, Richard Peng, Saurabh Sawlani, and Xiaorui Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2020.

11 Greg N Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

12 Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

13 Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

14 Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in data centers. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

15 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.

16 Giuseppe F Italiano, Silvio Lattanzi, Vahab S Mirrokni, and Nikos Parotsidis. Dynamic algorithms for the massively parallel computation model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

17 Joseph JáJá. *An Introduction to Parallel Algorithms*, volume 17. Addison-Wesley Reading, 1992.

18 Donald B Johnson and Panagiotis Metaxas. Optimal algorithms for the vertex updating problem of a minimum spanning tree. In *International Parallel Processing Symposium (IPPS)*, 1992.

19 David R Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000.

20 Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, October 1985.

21 Gary L. Miller and John H. Reif. Parallel tree contraction part 1: Fundamentals. In *Randomness and Computation*, pages 47–72. JAI Press, Greenwich, Connecticut, 1989. Vol. 5.

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

23 Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

24 Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

25 John H Reif and Stephen R Tate. Dynamic parallel tree contraction. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1994.

26 Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find with applications to incremental graph connectivity. In *European Conference on Parallel Processing (Euro-Par)*, 2016.

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

28 Robert E Tarjan and Renato F Werneck. Self-adjusting top trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005.

29 Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. Batch-parallel Euler tour trees. In *Algorithm Engineering and Experiments (ALENEX)*, 2019.

30 Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.