Brief Announcement: Parallel Dynamic Tree Contraction via Self-Adjusting Computation

Umut A. Acar, Vitalii Aksenov, Sam Westrick

To cite this version:


HAL Id: hal-01664903
https://inria.hal.science/hal-01664903
Submitted on 15 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
ABSTRACT
Dynamic algorithms are used to compute a property of some data while the data undergoes changes over time. Many dynamic algorithms have been proposed but nearly all are sequential. In this paper, we present our ongoing work on designing a parallel algorithm for the dynamic trees problem, which requires computing a property of a forest as the forest undergoes changes. Our algorithm allows insertion and/or deletion of both vertices and edges anywhere in the input and performs updates in parallel. We obtain our algorithm by applying a dynamization technique called self-adjusting computation to the classic algorithm of Miller and Reif for tree contraction.

1 INTRODUCTION
In many applications, algorithms operate on data that changes dynamically over time. For example, an algorithm may compute the heaviest subtree in an edge-weighted tree and may be required to update the result as the tree undergoes changes, e.g., as vertices or edges are inserted and/or deleted. Dynamic algorithms have been studied extensively; several papers review prior work [16, 17, 33]. Nearly all of the prior work on dynamic algorithms considers sequential dynamic algorithms. There is relatively little work on parallel dynamic algorithms, which would take advantage of parallelism when performing updates.

As an example dynamic problem, consider the classic problem of dynamic trees. This problem requires computing various properties of a forest of trees as edges and vertices are inserted and deleted [37]. Algorithms and data structures for dynamic trees have been studied extensively since the early '80s, including Link-Cut Trees [37, 38], Euler-Tour Trees [24, 41], Topology Trees [20], RC-Trees [3, 4], and, more recently, Top Trees [10, 40, 42]. These algorithms are work efficient: they allow the insertion/deletion of a single edge in logarithmic time (some in expectation, some amortized). Some of these algorithms have also been implemented [4, 42] and have been shown to perform well in practice. The algorithms and implementations, however, are all sequential. In prior work, Reif and Tate [35] give a parallel algorithm for dynamic trees but their algorithm is not fully dynamic: it allows changes only at the leaves of a tree and does not support deletions, leaving it to future work.

We are interested in designing a parallel algorithm for the dynamic trees problem. There are at least two challenges here.

- 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 believe that it is possible to overcome these challenges by using a technique called dynamization. The basic idea is to "dynamize" a static (non-dynamic) algorithm by recording carefully chosen intermediate results computed by the static algorithm and re-using these results when the data changes as a result of dynamic updates. In sequential algorithms, dynamization has been used for a variety of problems, e.g., by Bentley and Saxe [11], Overmars [31], Mumeley [30], and many others. We believe that parallel algorithms are particularly amenable to dynamization, because they minimize dependencies between subcomputations. In this paper, we outline a parallel algorithm for dynamic trees by dynamizing Miller and Reif’s tree contraction algorithm [28, 29] by using self-adjusting computation [1, 2, 27], which, for the purposes of this paper, can be viewed a dynamization technique. The resulting dynamic parallel algorithm allows insertion and deletion of any number of vertices or edges anywhere in the input forest (as long as no cycles are created) and supports parallel updates.

2 THE ALGORITHM
Our approach is based on the technique of self-adjusting computation for dynamizing static algorithms. The idea behind this technique is to use a construction algorithm to build a computation graph, which captures important data and control dependencies in the execution of the static algorithm. When the input data is changed, a change-propagation algorithm is used to update the computation 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.

Construction algorithm. The construction algorithm performs randomized tree contraction on an input forest \( F \) and produces a computation graph \( C_F \). At a high level, it follows Miller and Reif's algorithm [28] by proceeding in rounds of contraction. Each round takes a forest as input and produces a smaller forest for the next round by applying the rake and compress operations. The rake operation deletes all leaves; the compress operation deletes certain
vertices that have one child. Using tree contraction to compute a property of a tree requires specifying application-specific data and how such data is handled during rake and compress [4, 28, 29]. Because this can be done orthogonally, we don’t consider application-specific data in this paper.

The construction algorithm produces the computation graph \( C_F \) by storing a snapshot of the contraction of \( F \) at each round. Each snapshot consists of the configurations of vertices in the forest at that round. We define the configuration of a vertex \( v \) at round \( i \) in forest \( F \), written \( \kappa_i^F(v) \), as the set

\[
\kappa_i^F(v) = \{ (u, \ell_i^F(u)) \mid u \text{ is a neighbor of } v \text{ at round } i \text{ in } F \}
\]

where \( \ell_i^F(u) \) is a boolean indicating whether or not \( u \) is a leaf at round \( i \) in forest \( F \). The configuration of a vertex captures all of the information necessary to specify the treatment of that vertex by Miller and Reif’s algorithm. This information allows us to perform change-propagation efficiently by identifying the parts of the computation that are affected by an input change.

**Change-propagation algorithm.** Consider some input forest \( F \). Executing the construction algorithm on \( F \) yields a computation graph \( C_F \). Suppose we now wish to modify the input forest \( F \) by applying a set \( M \) of deletions and insertions of edges and vertices. Let \( G \) be the forest given by applying the changes \( M \) to \( F \). Instead of redoing the computation on \( G \) (which would require linear work), we provide a change-propagation algorithm that uses the computation graph \( C_F \) to perform the update more efficiently and quickly.

Given a change set \( M \), our change-propagation algorithm edits \( C_F \) and returns the updated computation graph \( C_G \). An important property of change-propagation is that the updated computation graph \( C_G \) is identical to one that would be obtained by running the construction algorithm on forest \( G \). Change-propagation can thus be iterated as many times as desired.

Change-propagation mimics the execution of the construction algorithm, but does so efficiently by only editing the parts of \( C_F \) which are affected by the input change \( M \). As in the construction algorithm, change-propagation proceeds in rounds but distinguishes between two classes of vertices at each round:

- **Unaffected vertices** are those that would be contracted in \( G \) in the same manner as in the contraction of forest \( F \). A vertex \( v \) is unaffected at round \( i \) iff \( \kappa_i^F(v) = \kappa_i^G(v) \).
- **Affected vertices** are those that would be contracted differently in \( G \) than in \( F \). A vertex \( v \) is affected at round \( i \) iff \( \kappa_i^F(v) \neq \kappa_i^G(v) \).

Each round of change-propagation takes a set of affected vertices as input and produces a new set of affected vertices for the next round. It updates the computation graph by deleting all edges which touch an affected vertex before re-running contraction (in parallel) for the affected vertices only. To produce the set of affected vertices for the next round, change-propagation only needs to keep track of what changes it makes to the computation graph.

**Analysis.** In the full version of the paper, we plan to establish the following two results. For a forest of size \( n \) and a batch of \( m \) insertions and/or deletions,

- change-propagation performs \( O(m \log \frac{n + m}{m}) \) work in expectation, and
- change-propagation exposes plenty of parallelism, i.e., its span (parallel time) is poly-logarithmic in \( n \) and \( m \).

<table>
<thead>
<tr>
<th>Size</th>
<th>Runtime 1</th>
<th>Runtime 39</th>
<th>Self Speedup</th>
<th>Work Improvement</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m )</td>
<td>( T_1^{C(m)} )</td>
<td>( T_{39}^{C(m)} )</td>
<td>( \frac{T_1^{C(m)}}{T_{39}^{C(m)}} )</td>
<td>( \frac{T_1}{T_{39}} )</td>
<td>( \frac{T_1^{C(m)}}{T_{39}^{C(m)}} )</td>
</tr>
<tr>
<td>( 10^2 )</td>
<td>0.004</td>
<td>0.004</td>
<td>1</td>
<td>260</td>
<td>260</td>
</tr>
<tr>
<td>( 10^3 )</td>
<td>0.04</td>
<td>0.04</td>
<td>1</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>( 10^4 )</td>
<td>0.28</td>
<td>0.14</td>
<td>2</td>
<td>3.71</td>
<td>7.43</td>
</tr>
<tr>
<td>( 3 \cdot 10^4 )</td>
<td>1.03</td>
<td>0.19</td>
<td>5.42</td>
<td>1.01</td>
<td>5.47</td>
</tr>
</tbody>
</table>

Figure 1: Execution times (in seconds) and speedups of change-propagation for \( m \) edges inserted into a forest of size \( n = 10^6 \), where \( T_1 = 1.04 \).

Notice that the work bound gives us \( O(\log n) \) work for a single change, and it gracefully approaches \( O(n) \) work as \( m \) approaches \( n \).

**Implementation.** We completed a relatively unoptimized implementation of our algorithm by using a fork-join parallelism library in C++ [5] which is similar to Cilk [21]. We also implemented Miller and Reif’s tree contraction algorithm for comparison.

We generated a random forest of size \( n = 10^6 \) where at least 60% of the vertices lie on a chain (i.e., have exactly 2 neighbors). On this forest, using one processor, Miller and Reif’s algorithm took 1.04 seconds, while our construction algorithm took 2.25 seconds. Since our construction algorithm constructs a computation graph by recording the configuration of vertices, the 2-factor overhead over Miller and Reif’s algorithm seems reasonable. With the same input, our construction algorithm runs in 0.28 seconds on 39 processors, leading to a self-speedup of 8.04.

Figure 1 shows the results for our change-propagation algorithm for inserting \( m \) randomly chosen edges \((10^2 \leq m \leq 3 \cdot 10^4)\). We write \( T_1^{C(m)} \) for the time of Miller and Reif’s algorithm on 1 processor, and \( T_{39}^{C(m)} \) for the time of change-propagation inserting \( m \) edges on \( p \) processors. The work improvement column measures the decrease in work achieved by our algorithm. For small \( m \), the work improvement is significant. As \( m \) increases, the work improvement decreases, converging to the work of the static algorithm at \(-5\%\) of the input size. The speedup captures the cumulative effect of both work improvement and benefits of parallelism. On a small number of changes \((m = 100)\), there is little parallelism but our change-propagation still achieves significant speedup over the static algorithm due to work improvement. As \( m \) increases, work improvement decreases but the amount of parallelism increases, leading to reasonable speedups even with moderately large changes.

### 3 RELATED WORK

**Tree contraction and dynamic trees.** Tree contraction, originally introduced by Miller and Reif, [28, 29] 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., [18, 19, 25, 26, 28, 29, 34, 36]). Prior work has established a connection between the tree contraction and dynamic trees problem of Sleator and Tarjan [37] by showing that
tree contraction can be dynamized to solve the dynamic trees problem [3, 4]. That work considers sequential updates only. In this paper, we outline how this connection can be generalized to take advantage of parallelism.

**Parallel dynamic algorithms.** Historically, parallel and dynamic algorithms have been studied mostly separately, with a few exceptions. Pawagi and Kaser propose a parallel (fully) dynamic algorithm that allows insertion and deletion of arbitrary number of vertices and edges as a batch [32]. 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 [1, 2, 23, 27]. Prior work applied self-adjusting computation to problems in several areas including in dynamic data structures [3, 4], computational geometry [7, 8], large data sets [13, 15], and machine learning algorithms [9, 39]. All of this prior work assumes a sequential model of computation. There has been some progress in generalizing self-adjusting computation to support parallelism [6, 12–14, 22].

**ACKNOWLEDGMENTS**

This research is supported by the National Science Foundation (CCF-1629444, CCF-1320563, and CCF-1408940), European Research Council (ERC-2012-StG-308246), and by Microsoft Research.

**REFERENCES**


