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

Umut Acar, Vitalii Aksenov, Sam Westrick

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

Umut A. Acar
Carnegie Mellon University, USA
Inria, France
umut@cs.cmu.edu

Vitaly Aksenov
Inria, France
ITMO University, Russia
aksenov@rain.ifmo.ru

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

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.

SPAA '17, July 24-26, 2017, Washington DC, USA

ACM ISBN 978-1-4503-4593-4/17/07.
https://doi.org/10.1145/3087556.3087595

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 algoritms, dynamization has been used for a variety of problems, e.g., by Bentley and Saxe [11], Overmars [31], Mulmuley [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 $\kappa_F^i(v)$, as the set

$$\kappa_F^i(v) = \left\{(u, \ell_F^i(u)) \,\middle|\, u \text{ is a neighbor of } v \text{ at round } i \text{ in } F\right\}$$

where $\ell_F^i(u)$ 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 $\mathcal{M}$ of deletions and insertions of edges and vertices. Let G be the forest given by applying the changes $\mathcal{M}$ 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 $\mathcal{M}$, 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 $\mathcal{M}$. 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 $\kappa_F^i(v) = \kappa_G^i(v)$.
- *Affected vertices* are those that would be contracted differently in G than in F. A vertex v is affected at round i iff $\kappa_F^i(v) \neq \kappa_G^i(v)$.

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.

Size	Runtime		Self Speedup	Work Improvement	Speedup
m	$T_1^{C(m)}$	$T_{39}^{C(m)}$	$\frac{T_1^{C(m)}}{T_{39}^{C(m)}}$	$\frac{T_1^*}{T_1^{C(m)}}$	$\frac{T_1^*}{T_{39}^{C(m)}}$
10^2	0.004	0.004	1	260	260
10^3	0.04	0.04	1	26	26
10^4	0.28	0.14	2	3.71	7.43
$3 \cdot 10^4$	1.03	0.19	5.42	1.01	5.47

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^* for the time of Miller and Reif's algorithm on 1 processor, and $T_p^{C(m)}$ 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 ~3% 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

[1] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Trans. Prog. Lang. Sys.*, 32(1):3:1–53, 2009.

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

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

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

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

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

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

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

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

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

[11] Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[12] Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Björn B. Brandenburg, and Rodrigo Rodrigues. iThreads: A threading library for parallel incremental computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 645–659, 2015.

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

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

[15] Yan Chen, Umut A. Acar, and Kanat Tangwongsan. Functional programming for dynamic and large data with self-adjusting computation. In *International Conference on Functional Programming (ICFP '14)*, pages 227–240, Sep 2014.

[16] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

[17] Camil Demetrescu, Irene Finocchi, and Giuseppe F. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. CRC Press, 2005.

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

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

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

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

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

[23] Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[24] 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.

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

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

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

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

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

[30] Ketan Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling (extended abstract). In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 180–196, 1991.

[31] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

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

[33] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *Principles of Programming Languages*, pages 502–510, 1993.

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

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

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

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

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

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

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

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

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