POSTER: Unexpected Scaling in Path Copying Trees

Vitaly Aksenov
ITMO University
Russia

Trevor Brown
University of Waterloo
Canada

Alexander Fedorov
IST Austria
Austria

Ilya Kokorin
ITMO University
Russia

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**.

Keywords: concurrency, persistent trees, path copying

1 Introduction

Although a wide variety of handcrafted concurrent data structures have been proposed, there is considerable interest in universal approaches (*Universal Constructions* or UCs) for building concurrent data structures. UCs (semi-)automatically convert a sequential data structure into a concurrent one. The simplest approach uses locks [3, 6] that protect a sequential data structure and allow only one process to access it at a time. However, the resulting data structure is blocking. Most work on UCs instead focuses on obtaining non-blocking progress guarantees such as *obstruction-freedom*, *lock-freedom* or *wait-freedom*. Many non-blocking UCs have appeared. Key examples include the seminal wait-free UC [2] by Herlihy, a NUMA-aware UC [10] by Yi et al., and an efficient UC for large objects [1] by Fatourou et al.

In this work, we consider the simpler problem of implementing *persistent* data structures which preserve the old version whenever the data structure is modified [7]. Usually this entails copying a part of the data structure, for example, the path from the root to a modified node in a tree [4], so that none of the existing nodes need to be changed directly.

We borrow ideas from persistent data structures and multi version concurrency control (MVCC) [9], most notably path copying, and use them to implement concurrent versions of sequential persistent data structures. Data structures implemented this way can be highly efficient for searches, but we expect them not to scale in write-heavy workloads. Surprisingly, we found that a concurrent treap implemented in this way obtained up to 2.4x speedup compared to a sequential treap [8] with 4 processes in a write-heavy workload. We present this effect experimentally, and analyze it in a model with private per-processor caches: informally, as the number of processes grows large, speedup in our treap of size N tends to $\Omega(\log N)$.

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577512

2 Straightforward Synchronization for Persistent Data Structures

In the following discussion, we focus on *rooted* data structures, but one could imagine generalizing these ideas by adding a level of indirection in data structures with more than one *entry point* (e.g., one could add a dummy root node containing all entry points).

We store a pointer to the current version of the persistent data structure (e.g., to the root of the current version of a persistent tree) in a `Read/CAS` register called `Root_Ptr`.

Read-only operations (queries) read the current version and then execute sequentially on the obtained version. Note that no other process can modify this version, so the sequential operation is trivially atomic.

Modifying operations are implemented in the following way: 1) read the current version; 2) obtain the new version by applying the sequential modification using path copying (i.e., by copying the root, and copying each visited node); 3) try to atomically replace the current version with the new one using `CAS`; if the `CAS` succeeds, return: the modifying operation has been successfully applied; otherwise, the data structure has been modified by some concurrent process: retry the execution from step (1). This approach trivially results in a lock-free linearizable data structure.

We expect read-only operations to scale extremely well. Indeed, two processes may concurrently read the current version of the persistent data structure and execute read-only persistent operations in parallel.

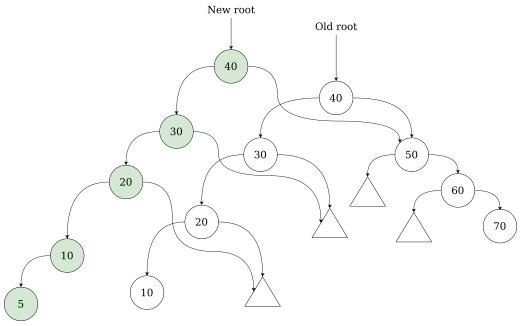
However, modification operations seemingly afford no opportunity for scaling. When multiple modifications contend, only one can finish successfully, and the others must retry. For example, consider concurrent modification operations on a set: 1) process `P` calls `insert(2)` and fetches the current pointer `RP`; 2) process `Q` calls `remove(5)` and fetches the current pointer `RP`; 3) `P` constructs a new version $\texttt{RP}_\texttt{P}$ with key 2; 4) `Q` constructs a new version $\texttt{RP}_\texttt{Q}$ without key 5; 5) `P` successfully executes `CAS(&Set.Root_Pointer, RP, RP`$_\texttt{P}$`)`; 6) `Q` executes `CAS` from `RP` to $\texttt{RP}_\texttt{Q}$ but fails; thus, `Q` must retry.

Successful modifications are applied sequentially, one after another. Intuitively, this should not scale at all in a workload where all operations must perform successful modifications. As we will see in Section 4, this intuition would be incorrect.

3 Analysis

The key insight is that failed attempts to perform updates load data into processor caches that may be useful on future attempts. To better understand, consider the binary search tree modification depicted in Fig. 1. Suppose we want to

Figure 1. The new version (green) of the tree shares its nodes with the old version (white)



insert two keys: 5 and 75. We compare how these insertions are performed sequentially and concurrently.

At first, we consider the sequential execution. We insert key 5 into the tree. It should be inserted as a left child of 10. Thus, we traverse the tree from the root to the leaf 10. On the way, we fetch nodes {40, 30, 20, 10} into the processor's cache. Note this operation performs four uncached loads.

Now, we insert 75. It should be inserted as the right child of 70. Our traversal loads four nodes: {40, 50, 60, 70}. Node 40 is already cached, while three other nodes must be loaded from memory. Thus, we perform three uncached loads, for a total of seven uncached loads.

Now, we consider a concurrent execution with two processes, in which P inserts 5 and Q inserts 75. Initially, both processes read Root_Ptr to load the current version. Then, 1) P traverses from the root to 10, loading nodes {40, 30, 20, 10}, and 2) Q traverses from the root to 70, loading nodes {40, 50, 60, 70}.

Each process constructs a new version of the data structure, and tries to replace the root pointer using CAS. Suppose P succeeds and Q fails. Q retries the operation, but on the *new* version (Fig. 1). Note that the new version shares most nodes with the old one.

Q inserts 75 into the new version. Again, the key should be inserted as the right child of 70. Q loads four nodes {40, 50, 60, 70} from the new version of the tree. Crucially, nodes {50, 60, 70} are already cached by Q. This retry only incurs one cache miss!

Thus, there are only five serialized loads in the concurrent execution, compared to seven in the sequential execution.

3.1 High-level analysis

We use a simple model that allows us to analyze this effect. (For the full proof see [5].) In this model, the processes are synchronous, i.e., they perform one primitive operation per tick, and each process has its own cache of size M. We show that for a large number of processes P, the speedup is $\Omega(\log N)$, where N is the size of the tree.

Now, we give the intuition behind the proof. To simplify it, we suppose that the tree is external and balanced, i.e., each operation passes though $\log N$ nodes. We also assume that the workload consists of successful modification operations on keys chosen uniformly at random. We first calculate the cost of an operation for one process: $(\log N - \log M) \cdot R + \log M$ where M is the cache size ($M < N$) and R is the cost of an uncached load. This expression captures the expected behaviour under least-recently-used caching. The process should cache the first $\log M$ levels of the tree, and thus, $\log M$ nodes on a path are in the cache and $\log N - \log M$ are not.

To calculate the throughput in a system with P processes, we suppose that P is quite large ($\approx \Omega(min(R, \log N))$). Thus, each operation performs several unsuccessful attempts, ending with one successful attempt, and all successful attempts (over all operations) are serialized. Since the system is synchronous, each operation attempt A loads the version of the data structure which is the result of a previous successful attempt A'. The nodes evicted since the beginning of A are those created by A'. One can show that in expectation only two nodes on the path to the key are uncached. Finally, the successful attempt of an operation incurs cost $2 \cdot R + (\log N - 2)$. Since successful attempts are serialized, the expected total speedup is $\frac{(\log N - \log M) \cdot R + \log M}{2 \cdot R + (\log N - 2)}$ giving $\Omega(\log N)$ with $R = \Omega(\log N)$ and $M = O(N^{1-\varepsilon})$.

4 Experiments

We implemented a lock-free treap and ran experiments comparing it with a sequential treap in Java on a system with an 18 core Intel Xeon 5220. Each data point is an average of 15 trials. We highlight the following two workloads. (More results appear in [5].)

4.1 Batch inserts and batch removes

Suppose we have P concurrent processes in the system. Initially the set consists of 10^6 random integer keys. Processes operate on mutually disjoint sets of keys. Each process repeatedly: inserts all of its keys, one by one, then removes all of its keys. Since the key sets are disjoint, each operation successfully modifies the treap. We report the *speedup* for our treap over the sequential treap below.

4.2 Random inserts and removes

In this workload, we first insert 10^6 random integers in $[-10^6;\ 10^6]$, then each process repeatedly generates a random key and tries to insert/remove it with equal probability. Some operations do not modify the data structure (e.g., inserting a key that already exists).

Workload	Seq Treap	UC 1p	UC 4p	UC 10p	UC 17p
Batch	451 940	0.89x	1.23x	1.47x	1.47x
Random	419 736	1.48x	2.38x	3.07x	3.19x

Acknowledgments

This work was supported by: the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Program grant: RGPIN-2019-04227, and the Canada Foundation for Innovation John R. Evans Leaders Fund (CFI-JELF) with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund: 38512.

References

[1] Panagiota Fatourou, Nikolaos D Kallimanis, and Eleni Kanellou. 2020. An efficient universal construction for large objects. *arXiv* (2020).

[2] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.

[3] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming.* Newnes.

[4] Haim Kaplan. 2018. Persistent data structures. In *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 511–527.

[5] Ilya Kokorin, Alexander Fedorov, Trevor Brown, and Vitaly Aksenov. 2022. POSTER: Unexpected Scaling in Path Copying Trees. *arXiv preprint arXiv:2212.00521* (2022).

[6] Leslie Lamport. 1987. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)* 5, 1 (1987), 1–11.

[7] Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press.

[8] Raimund Seidel and Cecilia R Aragon. 1996. Randomized search trees. *Algorithmica* 16, 4 (1996), 464–497.

[9] Y Sun, G Blelloch, W Lim, and A Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *VLDB* 13, 2 (2019).

[10] Z Yi, Y Yao, and K Chen. 2021. A Universal Construction to implement Concurrent Data Structure for NUMA-muticore. In *50th ICPP*. 1–11.