

Unexpected Scaling in Path Copying Trees

Ilya Kokorin

ITMO University

Russia

kokorin.ilya.1998@gmail.com

Trevor Brown

University of Waterloo, Canada

Canada

trevor.brown@uwaterloo.ca

Alexander Fedorov

IST Austria

Austria

afedorov2602@gmail.com

Vitaly Aksenov

ITMO University, Russia

Russia

aksenov.vitaly@gmail.com

Abstract

Although a wide variety of handcrafted concurrent data structures have been proposed, there is considerable interest in universal approaches (henceforth called Universal Constructions or UCs) for building concurrent data structures. These approaches (semi-)automatically convert a sequential data structure into a concurrent one. The simplest approach uses locks that protect a sequential data structure and allow only one process to access it at a time. The resulting data structures use locks, and hence are 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 by Herlihy, a NUMA-aware UC by Yi et al., and an efficient UC for large objects by Fatourou et al.

We borrow ideas from persistent data structures and multi-version concurrency control (MVCC), most notably path copying, and use them to implement concurrent versions of sequential persistent data structures. Despite our expectation that our data structures would not scale under write-heavy workloads, they scale in practice. We confirm this scaling analytically in our model with private per-process caches.

1 Introduction

Although a wide variety of handcrafted concurrent data structures have been proposed, there is considerable interest in universal approaches (henceforth called *Universal Constructions* or UCs) for building concurrent data structures. These approaches (semi-)automatically convert a sequential data structure into a concurrent one. The simplest approach uses locks [3, 5] that protect a sequential data structure and allow only one process to access it at a time. The resulting data structures use locks, and hence are 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 [9] 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* (also called *functional*) data structures, which preserve the old version whenever the data structure is modified [6]. 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) [8], 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 to not 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 [7] 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 $\Omega(\log N)$.

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 clearly produces 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 RP_P with key 2; 4) Q constructs a new version RP_Q without key 5; 5) P successfully executes `CAS(&Set.Root_Pointer, RP, RP_P)`; 6) Q executes CAS from RP to RP_Q but fails; thus, Q must retry its operation.

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

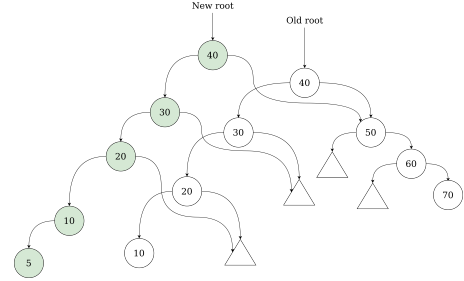
At first, we consider the sequential execution. We insert key 5 into the tree. It should be inserted as a left child of 10. Thus, we traverse the tree from the root to the leaf 10. On the way, we fetch nodes $\{40, 30, 20, 10\}$ into the processor's cache. Note this operation performs four uncached loads.

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*

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



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. (The full proof appears in Appendix A.) 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 through $\log N$ 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 $M = O(N^{1-\epsilon})$ is the cache size 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 $R = \Omega(\log N)$.

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 Appendix B.)

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

References

- [1] Panagioti 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] Leslie Lamport. 1987. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)* 5, 1 (1987), 1–11.
- [6] Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press.
- [7] Raimund Seidel and Cecilia R Aragon. 1996. Randomized search trees. *Algorithmica* 16, 4 (1996), 464–497.
- [8] Y Sun, G Btleloch, W Lim, and A Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *VLDB* 13, 2 (2019).
- [9] Z Yi, Y Yao, and K Chen. 2021. A Universal Construction to implement Concurrent Data Structure for NUMA-muticore. In *50th ICPP*. 1–11.

A Mathematical model

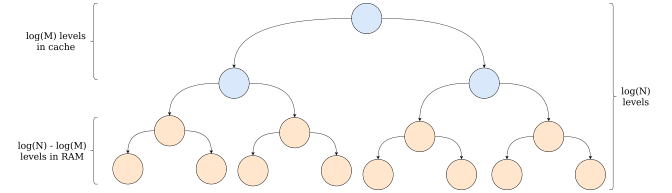
A.1 Sequential execution

Let us estimate how much time is spent on executing T operations sequentially on a binary search tree. Suppose our

binary search tree is *external*, i.e., data is contained only in leaves, while internal nodes maintain only routing information. Suppose tree contains N keys and the tree is balanced, therefore the tree height is $O(\log N)$. We suppose *uniform workload*: all keys from the tree are accessed uniformly at random.

Suppose the cache size is $M = O(N^{1-\epsilon})$, therefore, approximately upper $\log M$ levels of the tree are cached, while $\log N - \log M$ lower levels of the tree are not (Fig. 2).

Figure 2. Upper levels of the tree are cached, while lower levels reside in RAM



Each operation first loads $\log M$ nodes from the cache, spending 1 time unit per each cache fetch. After that, it loads $\log N - \log M$ nodes from the RAM, spending R time units per RAM fetch.

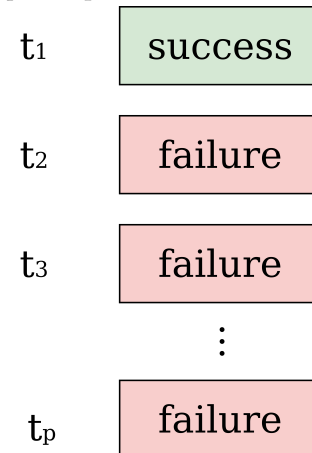
Thus, the sequential execution will take $T \cdot (\log M + R \cdot (\log N - \log M))$ time units to finish, where T is the number of operations.

A.2 Concurrent execution

Suppose we have P concurrent processes $\{t_i\}_{i=1}^P$ executing operations concurrently, while each process has its own cache of size larger than $\log N$.

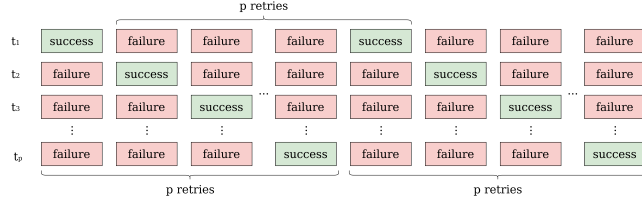
In our model we assume that each successful try of a modifying operation causes $p - 1$ unsuccessful tries of modifying operations on other processes (Fig. 3).

Figure 3. Each successful try of an operation causes unsuccessful tries of $p - 1$ operations



We also assume that operation completion events are distributed among processes in a round-robin pattern: first process t_1 executes its successful try of the operation, then process t_2 executes its successful try of the operation, and so on. Finally, t_p manages to complete its operation, the next process to get its successful try is yet again t_1 (Fig. 4).

Figure 4. Nearly each successful modifying operation consists of P retries: $P - 1$ unsuccessful and one successful

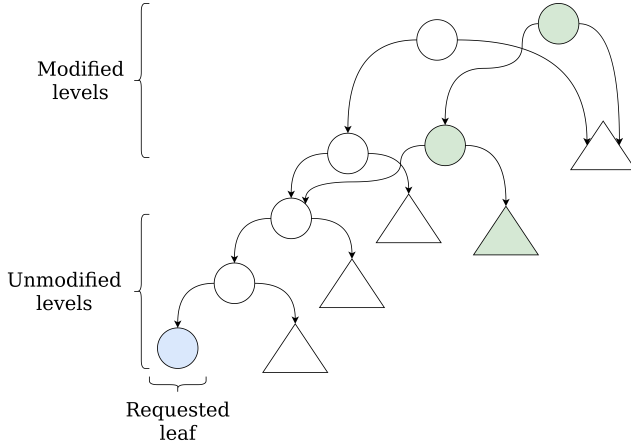


As follows from the diagram, almost each successful try of an operation is preceded by $P - 1$ unsuccessful retries (except for $P - 1$ first successful operation, which are preceded by the lower number of unsuccessful retries).

Let us estimate, how long the first retry takes to execute. We must load $\log N$ nodes, none of which might be cached. Thus, we spend $R \cdot \log N$ time units on the first retry.

Let us estimate now how much time we spend on subsequent retries. We begin with estimating, how many nodes on the path to the requested leaf have been modified (Fig. 5).

Figure 5. The number of modified nodes on the path to the requested node



Consider the successful modifying operation op , that led to a latest failure of our CAS and made us retry our operation the last time. Remember, that arguments of operations are chosen uniformly at random, therefore:

- There is $\frac{1}{2}$ probability that op modified some leaf from Root-→Right subtree, thus, the number of modified nodes on our path is 1;
- Similarly, there is $\frac{1}{4}$ probability that the number of modified nodes on our path is 2;
- ...
- Similarly, there is $\frac{1}{2^k}$ probability that the number of modified nodes on our path is k .

Thus, we can calculate the expected number of modified nodes on our path $\sum_{k=1}^{\log N} \frac{k}{2^k} \leq \sum_{k=1}^{\infty} \frac{k}{2^k} = 2$. Thus, the expected number of modified nodes on our path is not greater than 2.

Modified nodes were created by another process, thus they do not exist in our process cache. Therefore, they should be loaded out-of-cache, while all the remaining nodes reside in the local cache and can be loaded directly from it. Therefore, we spend $2 \cdot R$ time on average to load all the necessary nodes. In addition, we spend $\log N - 2$ time on average to load all the necessary nodes from the the local cache. Therefore, we spend $2 \cdot R + \log N - 2$ time to fetch all the nodes required for a last operation retry.

An operation execution consists of the first retry, executed in $R \cdot \log N$ and $P - 1$ subsequent retries executed in $(P - 1) \cdot (2 \cdot R + \log N - 2)$. Thus, a single operation is executed in $R \cdot \log N + (P - 1) \cdot (2 \cdot R + \log N - 2)$.

Therefore, we execute T operations in $\frac{T \cdot R \cdot \log N + T \cdot (P - 1) \cdot (2 \cdot R + \log N - 2)}{P}$ time, since we execute these operations in parallel on P processes.

To measure the speedup we simply divide the sequential execution time by parallel execution time:

$\frac{T \cdot (\log M + R \cdot (\log N - \log M))}{\frac{T \cdot R \cdot \log N + T \cdot (P - 1) \cdot (2 \cdot R + \log N - 2)}{P}} = P \cdot \frac{\log M + R \cdot (\log N - \log M)}{R \cdot \log N + (P - 1) \cdot (2 \cdot R + \log N - 2)}$. This gives us $\Omega(\log N)$ speedup when $P = \Omega(\min(R, \log N))$ and $R = \Omega(\log N)$.

B Experiments on other processors

We did the same experiments on Intel Xeon Platinum 8160 with 24 cores and AMD EPYC 7662 with 64 cores.

Workload	Seq Treap	UC 1p	UC 6p	UC 12p	UC 23p
Batch	638 600	0.93x	1.31x	1.37x	1.08x
Random	487 161	1.24x	3.23x	3.55x	2.8x

Table 1. Results for Intel Xeon Platinum 8160.

Workload	Seq Treap	UC 1p	UC 8p	UC 16p	UC 32p	UC 63p
Batch	459 580	0.96x	1.7x	1.91x	1.55x	1.02x
Random	396 898	1.36x	3.63x	2.41x	2.81x	2.3x

Table 2. Results for AMD EPYC 7662.

Unfortunately, one can see that the results are not so impressive when the number of processes is large enough. We suggest that the bottleneck for our benchmarks occurs in Java memory allocator.