Cache-Conscious Concurrent HAMT in Rust

Authors:
Chi-Chun, Chen
Princeton Ferro
Shaojie Wang

May 1, 2018

Contents

Abstract

We investigate how managing the memory layout of elements in a particular data structure can improve access time. In particular, we investigate how the performance of a hash array-mapped trie (HAMT) is affected by cache-conscious behavior. Also, we implement both blocking and lock-free HAMT to achieve concurrency. Our cache-conscious implementation without concurrency performs much better than the Rust official `HashMap` when the size is large. The lock-free version of cache-conscious hamt implementation has good locality and very low miss rate, and is also scalable.

1 Introduction

Concurrent data structure is gaining great attention. Prokopec et al. [2] proposed the very first lock-free hash trie, Ctrie, which achieves O(1) time complexity for snapshot operations. Prokopec [3] implemented cache-trie whose operations can run in expected constant time. Cache-trie uses "homemade cache", where a list of arrays are used to act as cache. In our project, we use Rust to implement a cache-conscious concurrent HAMT.

1.1 HAMT

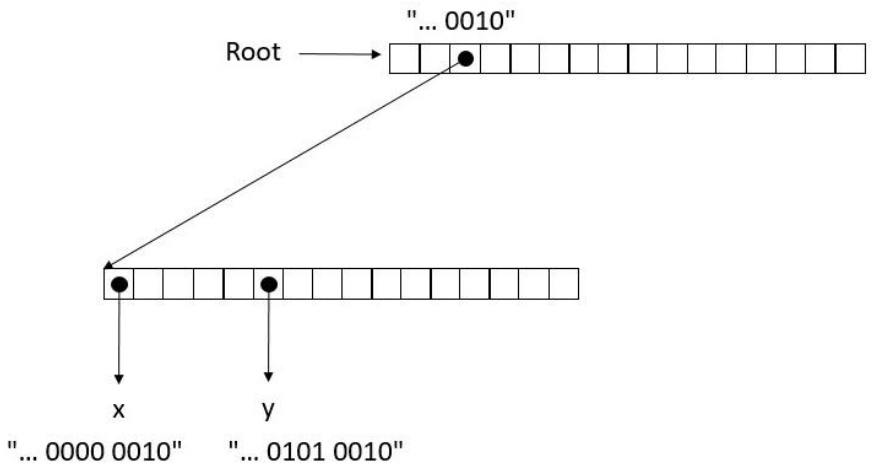
The basic idea of hash array mapped trie (HAMT) [1] is to build a trie (prefix tree) with the hash value of keys. Here is an intuitive illustration. Suppose we have a hash value of 16 bits, say "0000 0000 0000 0010", and value x to insert. We take 4 bits as a group, therefore, we need an array with length $2^4 = 16$ as a trie node to store the key group. Let us denote it by `a`. The first group we take is "0010", which is 2 in decimal. Therefore, we access `a[2]`, and nothing is stored there, so we store the value of this key in `a[2]`. Let us have one more hash value "0000 0000 0101 0010" with value y. When accessing `a[2]`, we find a value x there, which leads to conflict. So, we need to create a pointer to another trie node (denoted by `a_child`) in this position, and store x in `a_child[0]` since the second group of its key's hash value is "0000", and store y to `a_child[5]` for similar reason. The process is illustrated in Figure 1.

The HAMT is said to be fast (O(1) operations) and space-efficient when compared to chained or double hash trees [1]. Actually, HAMT uses shared nodes for different hash codes, and values are only stored on leaves.

1.2 Why Rust?

Rust is a safe high-level imperative programming language that compiles to native code. The main selling point of the language is its "borrowing system" and strong type checker, which in general prevents the programmer from having more than one mutable reference to an object at the same time. If the programmer follows the ownership rules that Rust defines without using any unsafe annotation, he or she would not have dangling pointers, memory leask, or even race conditions in the program. Unlike most programming languages, almost all the features that Rust provides have very little cost at runtime. For example, Rust has no polymorphism the way other object-oriented languages have. In contrast, Rust requires the programmer to implement *traits* (like an interface in Java) for a *struct* to achieve the effect of polymorphism. Though it could be a little bit inconvenient at the first glimpse, after getting used to Rust, the programmer will notice that Rust always finds the balance between runtime performance and the feature that programmers need.

Figure 1: An illustration of HAMT.



Though Rust is not only a safe and fast language, it does have some issues. The first thing is how it guarantee the "safety". Rust achieves "safety" by forcing programmers to follow the reference rules [5]. However, the reference rules is sometimes too rigorous for programmers to implement some kind of program. For example, if a programmer is implementing a multi-threaded application that each thread would like to modify the same vector, the programmer would found that Rust does not allow he or she to do so without locking the whole vector, since more than one mutable reference to the same vector certainly violates the reference rules. If a reader has the experience of implementing SIMD [6] program, he or she might think that the reference rules is nothing but a burden and would like to write some unsafe annotations to avoid the excess reminder from the compiler. Yet, if a programmer write lots of unsafe annotation in his or her program, the experience of writing Rust code would be almost the same as he or she write C/C++ code plus some redundant annotations.

Therefore, an canonical implementation of concurrent data structure could be very important in Rust, in that we could imagine that a good implementation of concurrent data structure in Rust should not only make use of the safety guarantees that Rust provides, but also have the same performance as other compiled language without garbage collection such as C/C++ and such an implementation is not trivial at all. However, the standard library of Rust does not have any concurrent data structures. If one need a concurrent data structure in Rust, he or she have to find it in Crates [7]. And since Rust is a new language comparing to other compiled language such as C/C++, the choice of concurrent data structure is not as much.

Furthermore, the concurrent data structure in Rust we found in Github does not take advantage of temporal locality or spacial locality [8]. Hence, in this project, we focus on exploring the possibilities of how cache and program locality could improve the performance of Rust and then try to make it concurrent in different ways.

2 Cache Conscious

Same as other high level programming language, to take advantage of cache, a programmer should design the memory layout and predict the program behavior according to his or her own heuristic. Before discussing the way we design the data layout for our trie implementation, we have to introduce about the memory model in Rust [9] at first. The memory model in Rust is mostly like C/C++, it has static variable that has infinite lifetime, memory automatically allocate and deallocate on stack, and memory on Heap that *sometimes* requires programmers to manually manage by themselves. In this project, our goal is to outperform the canonical HashMap [10] by using the memory locality while also provides scalability.

2.1 Memory Layout

To design the memory layout for hash trie, we should firstly think about what assumptions should we made. Firstly, which action should we care more about? read (get) or write (insert)? Secondly, if the hash trie is read (get) for most of the time, which pattern of read is the most common, namely, do the programmer traverse the hash trie repeatedly or get a few elements periodically. The assumption we made for our implementation is fairly simple, we think that user might read (get) the entry in the hash trie more often than he or she insert data. In addition, we think that programmer might access individual key-value pair more often than he or she traverse the whole hash trie. However, if the hash trie is big enough that even only one traverse happens could we benefit from designing the memory layout for traversing the whole hash trie.

2.2 Static data packing

For the above assumptions, we decide to use cache clustering [12] technique. Firstly, by packing the array mapped trie, we could certainly know that the first several level of memory access in the hash trie should always be cache hits. To put it more specifically, assume the depth of our hash trie is 4, the length of the array trie is 8, and the hash trie is full of nodes, we could expect the memory access of key-finding have near to one hundred percent of hit rate if the cache memory has more than $2^8 + 2^8 * 2^8 + 2^8 * 2^8 * 2^8$ bytes. It might be absurd to allocate more than ten megabytes of heap memory contiguously for merely one data structure, yet we think that it's a good starting point for preparing to design a more complicated memory layout in that we know the potential of integrate cache-conscious to hash trie.

Another reason for a hash trie with a contiguous memory layout to performs ridiculously good when accessing the entry sequentially is that the accessing of leaf node has almost the same percentage of miss rate as sequentially access an array. The statement is quite strong for sequentially accessing a hash trie to have almost the same miss rate as sequentially accessing an array. Yet, assume we have a hash trie indexing with 32 bits key and use 8 bits to find the array trie, therefore could have maximum depth of 4. While accessing the hash trie sequentially, the working set of finding the index in the contiguous memory layout does not change until an array trie has finished traversing all its children nodes, and the statement could be recursively applied to the whole hash trie.

2.3 Dynamic data packing

While cache clustering being a simple and effective way to improve the miss rate of sequentially traversing a hash trie, it might be overkill to group all the array trie together. Therefore, instead of statically packing all the array trie together, we could make use of the idea of dynamic data packing [13]. In the paper, Ding and Kennedy introduces several way to packing the data dynamically. The most intuitive and simple strategy is to packing the hash entry in the order of first appearance, which could be the time when the insert function is being called, and the strategy is called *first-touch packing* in the paper. The second strategy that we think is also simple and powerful is the *Group packing* strategy, which pack the data according to the reuse pattern of the hash trie. The reuse pattern could be an input from the programmer or recognized by the program itself. Firstly, if a programmer know his or her own application has some certain reuse pattern such as always accessing the hash trie entry lexicographically, then he or she could hint the hash trie when creating the data structure. Secondly, we could add some metadata to help us to find the reuse pattern. The metadata could be the depth of hash trie, the frequency of the accessing of array trie, or the dense of the array trie, etc. This part of cache-conscious optimization should be more interesting than static data packing. However, our implementation for this part is lack due to the time span of the project, yet should be a good starting point for other students to understand our previous work and being familiar with Rust.

3 Concurrency

3.1 Lock

Locks in Rust are used in a generic fashion (e.g. `Mutex<T>`, where `T` is the type of shared data), which is different from that in C/C++. One lock can only protect one piece of shared data. One simple example is

```
let data = Arc::new(Mutex::new(0));
for _ in 0..10 {
    let data = data.clone();
    thread::spawn(move || {
        let mut data = data.lock().unwrap();
        *data += 1;
    });
}
```

`data.lock()` acquires the lock, and when `data`'s lifetime ends at the end of the scope, the lock is released. In our project, we use one read write lock (`RwLock<T>`) per trie.

3.2 Lock-Free

This is based off of the work by [14]. We implemented a lock-free version with $O(\log n)$ insert and lookup.

The data structure is implemented generically as `LockfreeTrie<K,V>`, where `K : Hash`. On insert, we hash the key into a 64-bit unsigned integer, which is then broken up into groups of 4 bits. Each 4-bit integer indexes into the current level of the trie.

A `Node<K,V>` may be one of seven things:

1. `SNode { hash: u64, key: K, val: V, txn: AtomicPtr<Node<K,V>>}` This node stores our data.

2. `ANode(Vec<AtomicPtr<Node<K,V>>>)` This is an array node.

3. `NoTxn` This node indicates there is no trasaction. It is only ever valid for the `txn` field of an `SNode`.

4. `FSNode` This node indicates that a `SNode` is frozen.

5. `FVNode` This node indicates that an empty (null) array entry is frozen.

6. `FNode { frozen: AtomicPtr<Node<K,V>> }` This node indicates that an `ANode` is currently frozen.

7. `ENode { parent: AtomicPtr<Node<K,V>>, parentpos: u64, narrow: AtomicPtr<Node<K,V>>, hash: u64, level: u64, wide: AtomicPtr<Node<K,V>>}` This is an expansion node, which represents an in-progress expansion of an `ANode`.

Array nodes start off at length 4 (except for the root, which starts off at length 16). To ensure safety, we do not modify the contents of `SNode`s and `ANode`s directly. Rather, each `SNode` has a transaction (`txn`) that, when modified indicates an in-progress operation. And for `ANode`s that need to be expanded, we first create an expanded node, copy the new contents over, and finally CAS the `wide` element in the position of the original `ANode`.

We chose `std::sync::atomic::AtomicPtr` for this implementation, which has two advantages over `std::sync:Arc`. `AtomicPtr` exposes the much-needed `compare_and_swap`, `load`, and `store` operations, as well as zero overhead from reference counting. However, acquiring a mutable reference to the pointed object requires an `unsafe` context:

```
fn _insert(..., cur: &mut Node<K,V>, ...) -> bool {
...
    if let Node::ANode(ref mut an) = cur {
     // an is type Vec<AtomicPtr<Node<K,V>>>
        let old: &AtomicPtr<Node<K,V>> = &an[i];
        let oldptr: *mut Node<K,V> = old.load(...);
        if !oldptr.is_null() {
         // now we have a safe reference
            match unsafe {&*oldptr} {
             ... => ...
            }
        }
    }
}
```

Safety is ensured because this unsafe cast is only used after a check for null-ness or where the value could not possibly be null.

3.3 Memory Layout

`Node<K,V>`s are packed into a contiguous array, which is managed by the `Allocator<T>`. The allocator reserves a buffer on the heap whose length is statically determined. For our tests we chose 100,000,000 elements.

3.4 Caching

Not referring to the job of the hardware, we also have a separate cache structure that is always the concatenation of all elements in the largest level of the HAMT. Adjusting this level is triggered when the number of cache misses reaches an arbitrary threshold (2048 in our case). It is demonstrated in [14] that this strategy reduces the best-case upper-bound of `lookup()` to $O(1)$.

4 Experiments

4.1 Official HashMap v. contiguous hash trie

In this case, we test the influence of memory layout on the *get* operation. Results are in Fig. 2,3,4. The X-axis is the number of sequential *get()* function in the benchmark of both the *HashMap* in Rust standard library and the non-concurrent cache-conscious trie. The non-concurrent cache-conscious trie outperforms the canonical *HashMap* when the number of sequential *get* is more than 100,000 elements in the benchmark.

4.2 Official HashMap v.s. blocking contiguous hash trie

In this case, we test concurrent `get` and see the scalability of our implementation. Because we are using nightly version of Rust, and on `node` machines we did not install it due to spatial issues, we just have these tests on our own computer, where maximum thread number is 8. As we can imagine, if we can use more threads, the results will be more promising than what will be shown next.

We post experiments on $10^3 \to 10^8$ nodes, and results are shown in Fig. 5,6,7,8,9. We only post tests on read operations because the official HashMap is not concurrent, so we can only compare on the reading performance. We can see that the scalability of our implementation is better than that of HashMap, yet not good enough. However, when node number grows, we can see better scalability and larger speed up for our hash trie. Also, when thread number reaches 8, which is the maximum thread number, in the 10,000,000-node case, the execution time of our trie is going to drop below that of HashMap. If we can test on machines with more cores, we can expect a lower execution time and that our hash trie outperforms HashMap.

4.3 Lock-free cache-conscious HAMT

See Figure 10.

The lock-free cache-conscious version shows good scalability, especially when the number of threads exceeds the maximum number of hardware contexts (at t=16 in this case).

Figure 2:

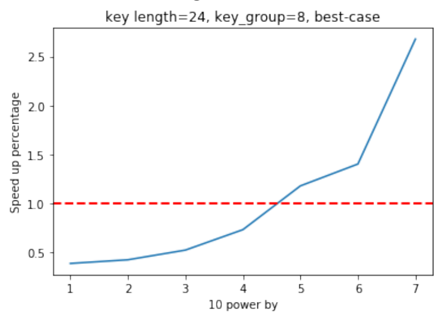


Figure 3:

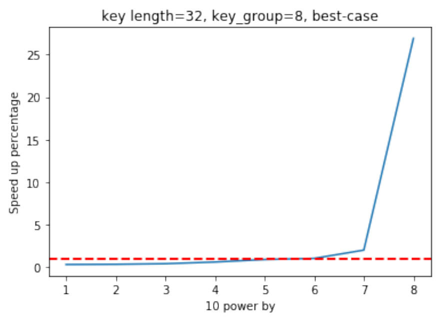


Figure 4:

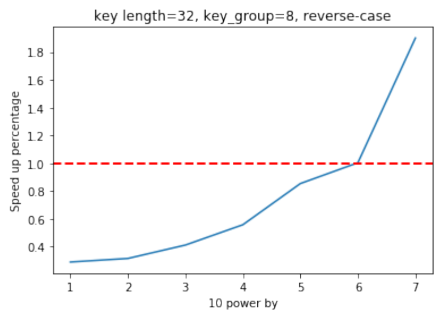
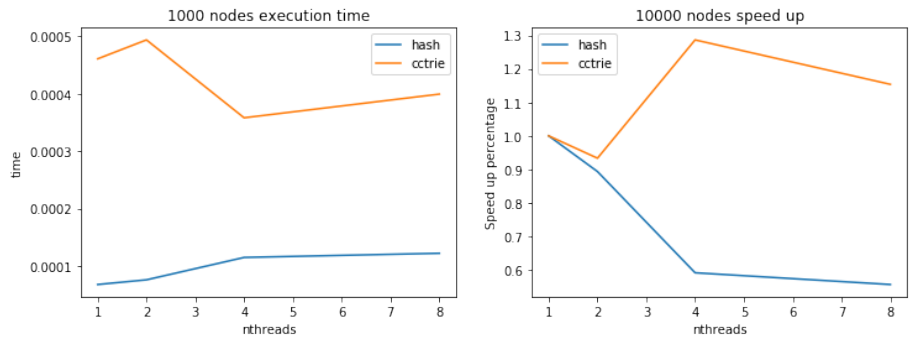


Figure 5: (Contiguous) Read test on 1,000 nodes.



5 Future Works

5.1 Cache Conscious

The data structure in Rust standard library is implemented elegantly and well-tested, however, the implementation is not in general cache-consciousness. While we implement the cache conscious concurrent hash trie in this project, the duplicate thing that we always do is to implement a custom allocator. An allocator library will significantly reduce the workload of designing a cache-conscious data

Figure 6: (Contiguous)Read test on 10,000 nodes.

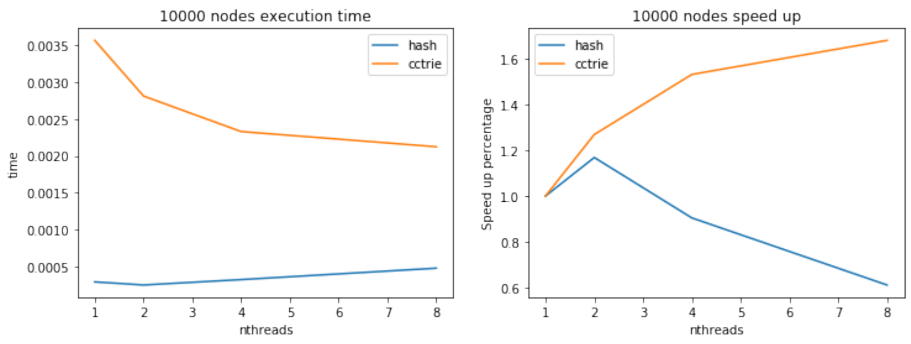


Figure 7: (Contiguous)Read test on 100,000 nodes.

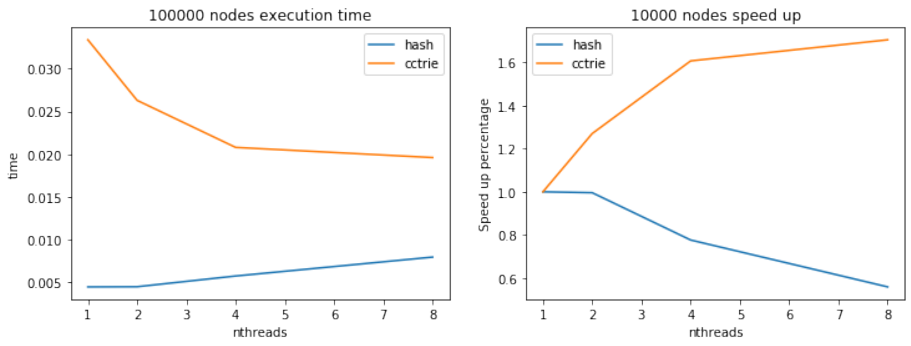
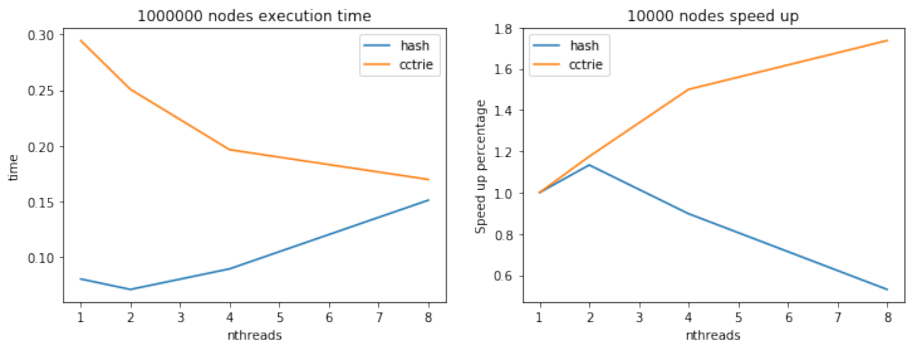


Figure 8: (Contiguous)Read test on 1,000,000 nodes.



structure in Rust. The library should have an allocator structure supporting both static and dynamic data packing, and inputs such as hint from programmer and metadata for finding reuse patterns.

Figure 9: (Contiguous)Read test on 10,000,000 nodes.

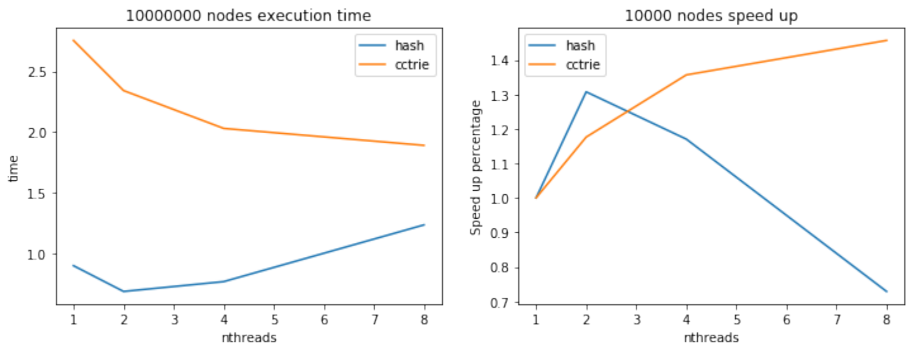
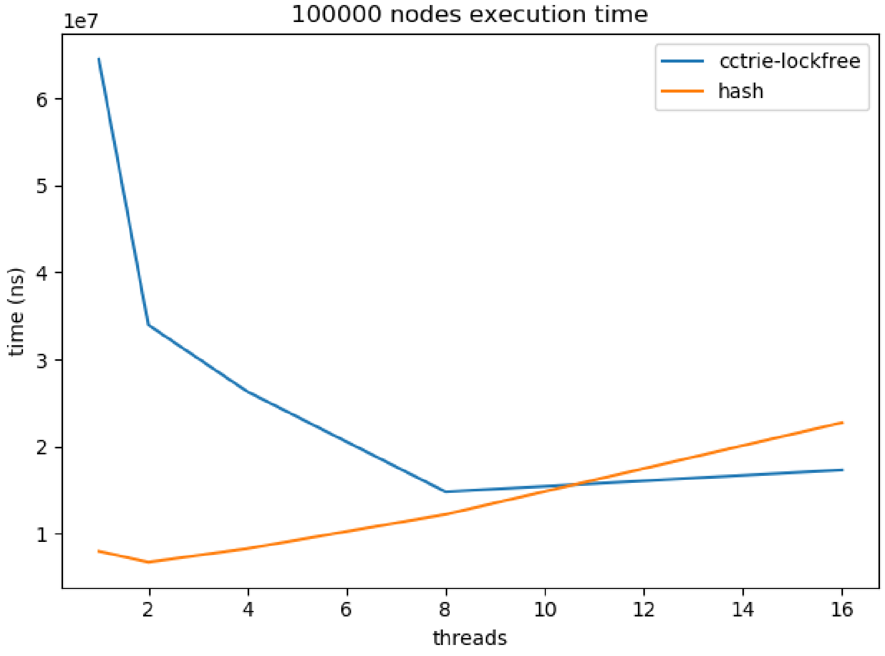


Figure 10: (lock-free) Read test on 100,000 nodes.



5.2 Concurrency

For the blocking HAMT, we have not yet tried adding locks per element. Currently, we only use one lock per trie, which is not so good for insert operations. We can certainly expect better scalability when locks per element is implemented.

5.3 Lock-free

The lock-free implementation would benefit from more optimization and tweaking of parameters, such as M, the maximum number of misses before adjusting the cache level. Furthermore, changing the code to use the non-standard `AtomicRef` and perhaps using features from the `crossbeam` crate would allow us to use the CAS primitive without unsafe code.

6 Conclusion

In this project, we implemented concurrent cache-conscious hash array mapped trie. We first implement the memory layout for cache-consciousness, which gives the hash trie a speed up of 2500% in best case with 10,000,000 nodes. For the blocking version of hash trie, we can see a good scalability when node number goes larger. For lock-free version, we implemented a concurrent HAMT with a cache-trie for $O(1)$ lookup in the best-case. Our implementations scale well with the number of threads.

References

[1] Phil Bagwell. Ideal Hash Trees:
`http://lampwww.epfl.ch/papers/idealhashtrees.pdf`

[2] Prokopec, Aleksandar, et al. ”Concurrent tries with efficient non-blocking snapshots.” Acm Sigplan Notices. Vol. 47. No. 8. ACM, 2012.

[3] Prokopec, Aleksandar. ”Cache-tries: concurrent lock-free hash tries with constant-time operations.” Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2018.

[4] The Rust Programming Language:
`https://github.com/rust-lang/rust`

[5] Rust Reference Rules:
`https://doc.rust-lang.org/book/second-edition/ch04-02-references-and-borrowing.html`

[6] SIMD:
`https://en.wikipedia.org/wiki/SIMD`

[7] Crates.io:
`https://crates.io/crates`

[8] Locality:
`https://en.wikipedia.org/wiki/Locality_of_reference`

[9] Memory Model in Rust:
`https://doc.rust-lang.org/book/first-edition/the-stack-and-the-heap.html#semantic-impact`

[10] Rust HashMap:
`https://doc.rust-lang.org/nightly/std/collections/struct.HashMap.html`

[11] The Documentation of Rust Programming Language
`https://doc.rust-lang.org/beta/nomicon/borrow-splitting.html`

[12] Cache Conscious Data Structure:
`https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/ccds.pdf`

[13] Chen Ding. Ken Kennedy: Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time:
`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.2169&rep=rep1&type=pdf`

[14] Cache Tries: Concurrent Lock-Free Hash Tries with Constant-Time Operations. Aleksandar Prokopec. Oracle Labs.
`https://dl.acm.org/citation.cfm?id=3178498`