Implementations of Concurrent Radix Trie and Measurement of their Performance

Sam Wang and Jake Wen

May 2024

1 Summary

Our 15-418 final project focuses on the implementation and analysis of different Raidx Trie data structures, an essential data structure used in IP routing tables and in-memory file systems. We implemented and evaluated the performance of four versions of Radix Tries: (1) sequential Radix trie, (2) Parallel Radix Trie with Global Lock, (3) Parallel Radix Trie with Fine-grained Locks, and (4) Lock-free Parallel Radix Trie.

2 Background

2.1 Introduction to Radix Trie

A Radix Trie, also known as a Radix Tree or Patricia Tree, is an efficient data structure used for storing and searching strings, such as IP routing tables, document scans, in-memory file system, and auto-complete systems. This data structure is a compressed form of a trie, where each node in a typical trie represents a single character of a string. In contrast, a Radix Trie collapses chains of nodes that do not branch into single nodes, each potentially representing multiple characters of the string. This compression reduces memory usage and potentially speeds up search operations by minimizing traversal steps. Radix Tries are powerful for concurrent operations due to their hierarchical structure, which allows independent operations on different parts of the trie. However, achieving an efficient, concurrent Radix Trie implementation is challenging due to issues with synchronization, data locality, and the complexity of managing dynamic node allocations. Coarse-grained locking, fine-grained locking, and lock-free approaches have their own sets of trade-offs between performance and complexity.

2.2 Key Data Structures and Operations

The provided Radix Trie implementation includes the following components:

Data Structures:

- `RadixNode`: Represents a node in the trie, holding a `key` (partial string), `value`, an array of `children` nodes, a terminal flag `isTerminal` indicating end of a valid string, and a `nodeMutex` for thread-safety required with fine-grained lock. For simplicity, we constrained the key string to only contain lower English letters, which means that each node has at most 26 outward edges.

- `RadixTreeParallel`: Represents the entire trie with a `root` node and methods for manipulation such as `put`, `getValueForExactKey`, `collectPairs`, and `removeKey`.

Operations: The inputs and outputs for the concurrent Radix Trie are keys of string and values of any type. The public interface of the `RadixTreeParallel` class provides methods that facilitate the concurrent modifications and querying of the trie structure:

- `put(const std::string& key, const O& value)`: Inserts a key-value pair into the trie. If the key already exists, its value is updated. Otherwise, a new path is created in the trie. Since the Radix Trie may have multiple characters coalesced together in a node to save memory, it is important to note that a new path may split one node into 2 so that the keys concatenated together on a certain path matched with the desired key. For duplicate put requests, the trie will update the value.

- `getValueForExactKey(const std::string& key)`: Retrieves the value associated with a specific key. This method returns a value of type `O` if the key exists in the trie; otherwise, it returns a default instance of `O`.

- `collectPairs(const std::string& prefix)`: Returns a vector of all key-value pairs in the trie that start with the given prefix. This operation is useful for implementing features such as autocomplete or prefix-based searches.

- `removeKey(const std::string& key)`: Removes a key and its associated value from the trie. This method adjusts the trie structure accordingly to maintain its integrity and compactness. Similar to `put`, it is important to keep the memory property of the Radix Trie. In other words, if the removal leads to situations that nodes can be coalesced together to save space, it will combine those nodes greedily. Removal of a non-existing key should act as a no-op.

2.3 Computational Expensive Part and Parallelization

The insertion, deletion, and exact-key search in a Radix Trie are usually cheap computations because their complexities depend on the length of the keys. The only prefix search that returns all key value pairs associated with a certain key prefix may be expensive with a short key that is closed to the root level. However, typical use case of the Radix Trie like auto-completion avoids prefix

searches with a short prefix that may give a very large output. In terms of parallelization, the trie structure supports high levels of concurrency because different keys can be inserted or searched in parallel without interfering with each other, provided they belong to different parts of the trie. This structure also supports parallelism as operations on non-overlapping sub-trees can proceed independently. Yet, SIMD execution or exploitation on locality do not seem to be applicable in this case because the tree structure requires pointers on disjoint memory addresses.

2.4 Locality and Challenges

Tries can suffer from poor data locality due to nodes often being dynamically allocated, leading to scattered memory access patterns. This can be particularly challenging for caches and memory performance in a concurrent environment. Since the data accessing runtime is hard to optimize and implementing the data structure itself limits our ways to optimize concurrent throughput through busy time of doing work, our main goal focus on reducing the synchronization overhead.

Efficient synchronization is challenging. With a correct and fast sequential implementation of trie, we can naively turn it into a concurrent trie by wrapping the whole trie with a mutex lock. This brute-force approach enforces one client to have the exclusive right to use the entire trie at a time. Thus, it did not take advantage of the inherent parallelism on the trie structure, leading to a poor throughput and speedup. More refined approaches should not block the traversal of a certain path on the trie as much as possible. This can be done by either implementing the fine-grained lock to minimize unnecessary synchronization or the lock-free data structure that eliminates any synchronization on queries.

3 Approach

3.1 Journey

Initially, we implemented sequential versions of the trie data structure, focusing on three variations: the basic trie, Radix trie, and PATRICIA trie. Our exploration of the basic and Radix tries was successful. However, we struggled with implementing the PATRICIA trie due to its requirement to compress strings into ASCII bit representations, a concept that proved challenging to comprehend and implement correctly. Ultimately, we decided to use the Radix trie as our baseline for future performance benchmarking because of its balance between efficiency and implementation complexity.

With the baseline in place, we proceeded to design a parallel implementation. We used a shared address space to enable concurrent operations by multiple threads. Our first attempt involved a global lock that guarded the entire data structure. This ensured thread safety but proved inefficient because threads working on different parts of the trie were unnecessarily restricted. Locking the

entire structure led to significant performance bottlenecks as all threads had to wait their turn.

To address this, we refined our locking strategy to employ fine-grained locks at the node level. Adopting a "monkey-bar" approach, each thread would lock both the current node and its parent, releasing the parent lock only after securing a lock on the child node. This approach ensured that each thread held exclusive access only to the portion of the data structure it was working on, reducing the contention for locks and thereby improving parallelism.

However, we recognized that even finer optimizations were possible. We began exploring lock-free algorithms to remove the overhead associated with locking altogether. This task was challenging because ensuring the atomicity of concurrent operations is inherently difficult. We relied heavily on the compare_exchange operation provided in C++ to ensure that modifications to shared memory regions were atomic and safe. This required significant time and effort to get right, but the result was a notable improvement in performance.

In summary, our approach began with understanding different trie structures, selecting the Radix trie for its favorable characteristics, and gradually refining our implementation from a global lock to fine-grained locking. Finally, by leveraging lock-free algorithms with atomic operations, we achieved an efficient, highly concurrent version of the Radix trie suitable for high-throughput applications.

3.2 Fine-grained Lock Implementation Details

3.2.1 Insertion

When implementing the insertion functionality, several scenarios must be taken into account. One of the most common situations occurs when the new key shares a prefix already present within a node. In this instance, the solution involves creating a new node and linking it to the existing prefix node as a child at the appropriate index.

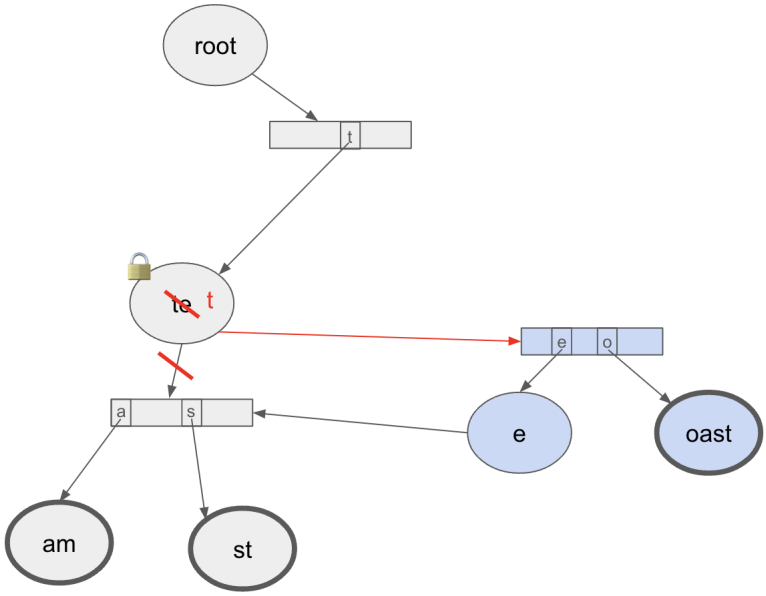
Another possibility arises if the key is already fully stored within a node. Here, the task is straightforward: update the value and modify the isTerminal boolean variable accordingly.

However, more complex cases demand splitting an existing node. For example, consider a trie containing the keys "team" and "test," to which we aim to add the new key "toast." During the search for the correct node to operate on, we find that a node with the prefix "te" is present, but this prefix only partially aligns with "toast" at the initial "t" character. As a result, the "te" node needs to be split.

For our fine-grained lock implementation, we first acquire a lock for the "te" node, then update its key to the substring "t." A new node is created to represent the remaining "e" suffix, which becomes a child of the newly modified "t" node. The new "e" node retains the children originally linked to the old "te" node. Additionally, a separate node is created to represent "oast," the remaining characters of "toast."

After these changes, the original "te" node is now "t," with "e" and "oast" as its children. The new "e" node contains the children that previously belonged to the "te" node. Upon completing these updates, the lock on the "te" node is released.

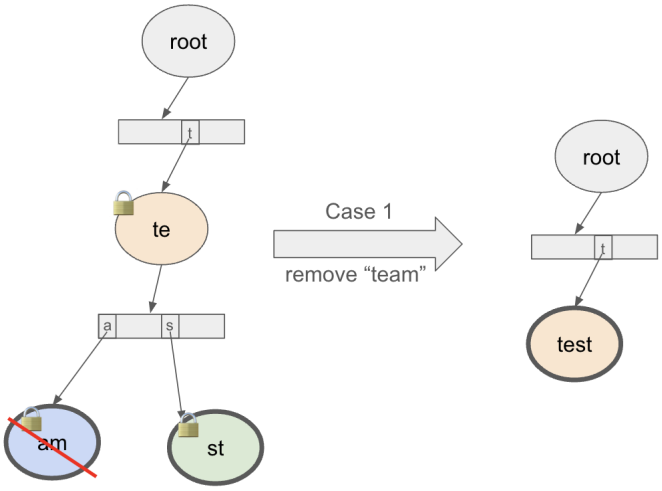
Throughout this operation, a locking scheme akin to the monkey-bar locking mechanism discussed in class is used. As each node is traversed, the lock on the parent node is released before acquiring the lock on the child node. This scheme is effective, as modifications are restricted to a single node at a time, ensuring no changes are propagated up or down the hierarchy that could compromise the insertion process.



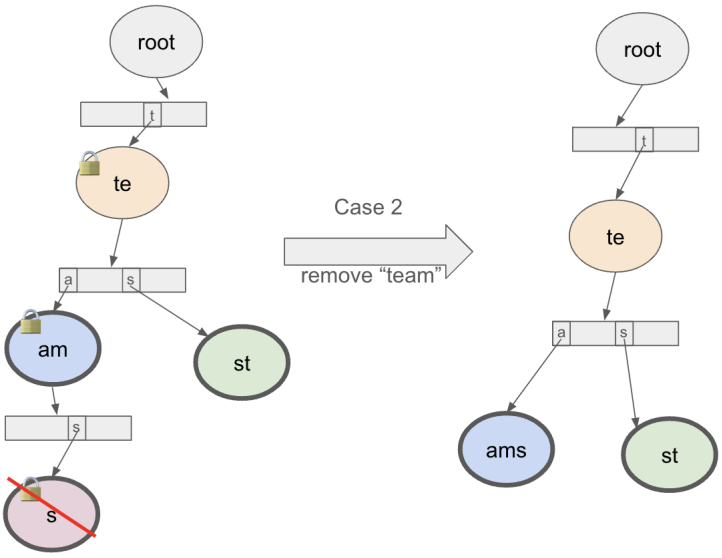
3.2.2 Deletion

The deletion operation presents greater challenges compared to insertion, as it requires handling a broader range of scenarios.

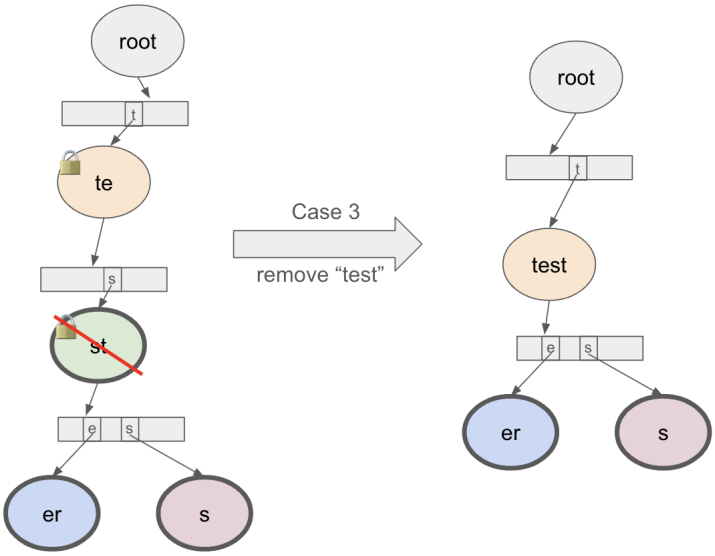
Case 1 If we are deleting a word like "team" and the terminal node has no children, the process is straightforward: simply remove the terminal node with the key "am." However, even after deleting "am," we need to consider the parent node "te." With only one child ("st") remaining and "te" itself not being a terminal node, we must optimize the trie by merging "te" with "st," resulting in a single node with the key "test" to save space.



Case 2 In the second scenario, we are still aiming to delete the word "team." Here, the node "am" has a single child, "s." After removing "team," "am" ceases to be a terminal node. Since it only has one child, we must merge "am" with "s" to optimize the structure.



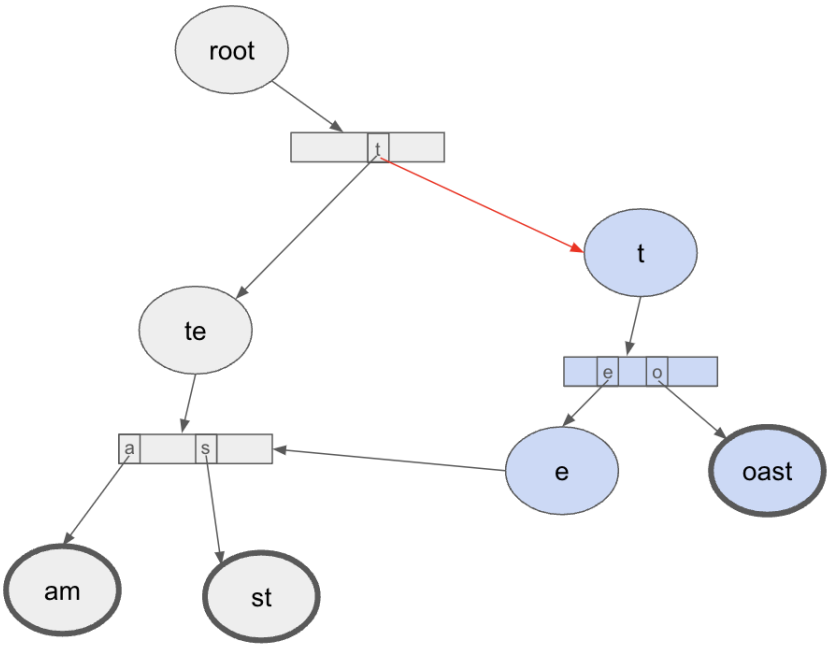
Case 3 Lastly, consider the case where we are deleting the word "test." Once "test" is removed, "st" becomes a non-terminal node and is the sole child of "te." As a result, "te" and "st" must be merged to streamline the structure.



3.3 Lock-free Insertion Implementation Details

To achieve atomic insertion in our lock-free implementation, we utilize the strong_compare_exchange function in C++. This function compares the value at a specified memory address to an expected value and, if they match, replaces it with a new value. The operation is performed atomically by the hardware, ensuring safe concurrent use.

During insertion, whether adding a new node, modifying an existing node, or splitting a node, we first create a new node with the desired key, value, isTerminal status, and children. We then use compare_exchange to replace the existing node atomically. If successful, the parent node's child pointer is updated to reference the new node, similar to the red arrow depicted in the image below.



4 Results

4.1 Goals Reached

This project aims to implement a concurrent Radix Trie that supports concurrent operations so that a bulk number of these operations can be accelerated with more threads splitting a job. To this end, we have implemented several different variations of concurrent Radix Trie. Among these solutions, the lock-free version seemed to reduce the most synchronization overhead when there is high contention presented. It reaches near linear speedup for some workloads that we have designed to simulate actual use case of a Radix Trie.

4.2 Experiment Setups

The experiments are conducted on GHC and PSC machines with the following hardware configurations.

GHC:

- CPU: 8 cores x86_64 Intel i7-9700 (8) @ 4.700GHz
- Memory: 16 GB
- Operating System: Ubuntu 22.04.1 LTS x86_64

PSC Regular Memory Machine:

- CPU: 2 AMD EPYC 7742 64-Core
- Memory: 256 GB
- Operating System: Linux

In our experimental setup to evaluate the performance of the concurrent Radix Trie, we have designed 3 different scenarios including put-intensive, get-intensive, and even distribution of puts and gets. Specifically, the total number of operations was set at 10 million. The nature of these operations was determined by a specified ratio between the expected number of `put` and the number of `getValueForExactKey`.

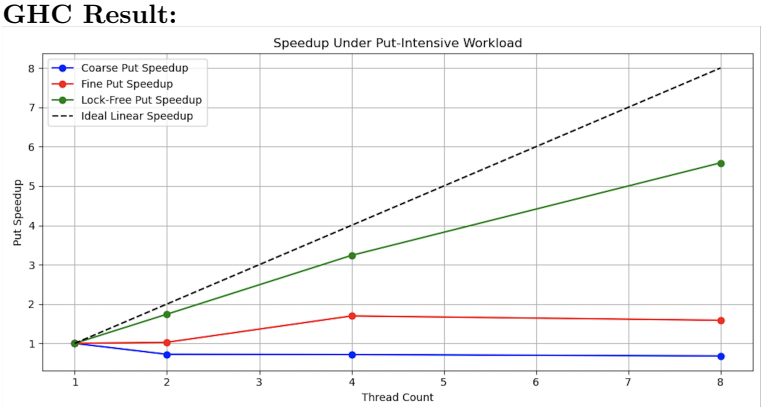
The keys for these operations were randomly selected from a Kaggle English word dataset, each being a string of lowercase English letters with average length 10. The selection between a put or getValueForExactKey operation for any given instance was randomized, adhering to the predefined operation ratio. This approach was intended to simulate a realistic and varied workload where the trie structure would be subject to concurrent accesses, testing both its performance and robustness in a multi-threaded environment. The test utilized OpenMP to parallelized the operations with a self-defined number of threads, aiming to reflect a moderate concurrency level that could be typical in many practical applications. The baseline for speedup is defined to be the optimized parallel implementation for a single CPU, so the experiment needs to measure the run-time against different number of threads and calculate the speedups by comparing the the baseline.

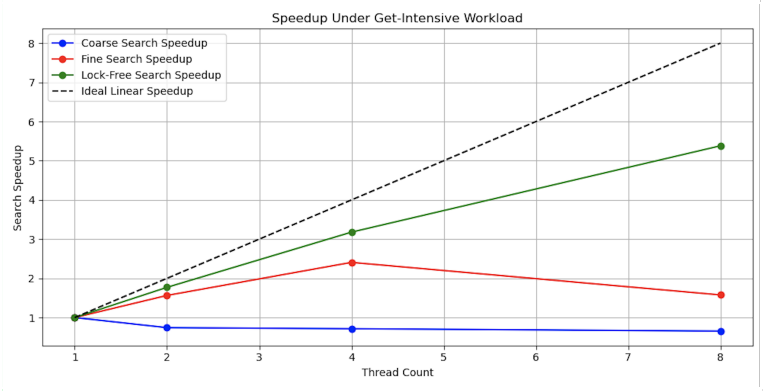
There are two distinct problem size parameters that we are interested in including the number of operations and the length of the keys inserted. It is important to observe that the amount of total work depends on the former while the time took for each operation is determined based on the latter. Since the length of the keys on average is strongly related to the height of the trie, we must take it into account when analyzing the contention. We initially chose the number of operations to be 10 million so that the job takes a nontrivial amount of time to complete.

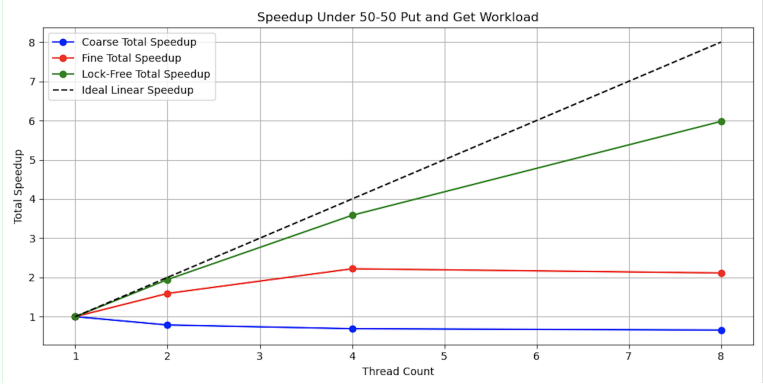
4.3 Experiment Results

Overall, we observed the lock-free Radix Trie presented a good speedup when increasing the number of threads with fine-grained lock being at the second place. All workloads result in similar speedup graphs, and the only exception is that the drop-off of speedup in find-grained lock Radix Trie is more significant in Get-intensive workload. Note that the coarse-grained lock trie always underperfroms with respect to the single thread run-time, which is expected since more threads incur significant more synchronization overhead due to wait time on lock that wraps the sequential operations.

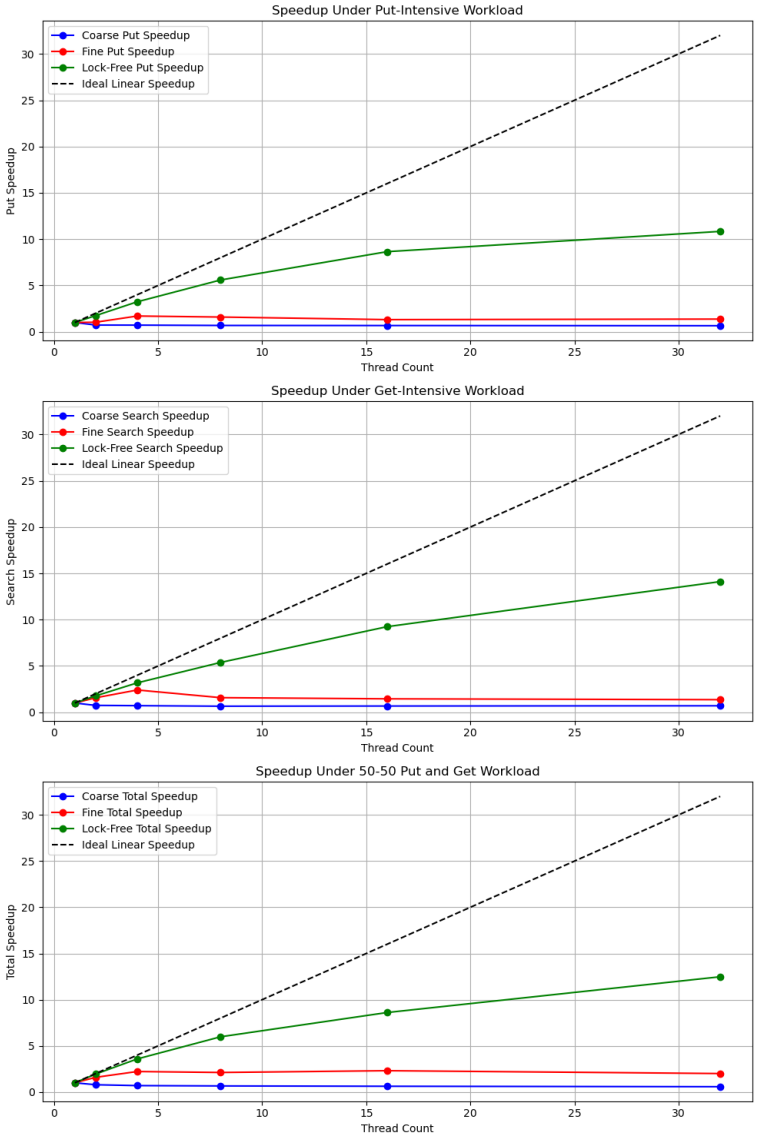
GHC Result:


Speedup Under Put-Intensive Workload
Coarse Put Speedup
Fine Put Speedup
Lock-Free Put Speedup
Ideal Linear Speedup
Put Speedup
Thread Count


Speedup Under Get-Intensive Workload
Coarse Search Speedup
Fine Search Speedup
Lock-Free Search Speedup
Ideal Linear Speedup
Search Speedup
Thread Count


Speedup Under 50-50 Put and Get Workload
Coarse Total Speedup
Fine Total Speedup
Lock-Free Total Speedup
Ideal Linear Speedup
Total Speedup
Thread Count

PSC Result:



4.4 Results Explanations

The trie with fine-grained lock presents the most unexpected performance. This can be explained by the contention in synchronization. Since the fine-grained lock trie implements hands over hands locking, every operation will begin by acquiring the lock for the root node. Similarly for other nodes, the closer they are to the root node, there are more likely contention among these nodes. To confirm

such hypothesis, we have run experiments with find-grained locking trie that does not ensure correctness but can save the synchronization overhead. In other words, we carefully removed locking from the Radix Trie and patched the code to avoid run-time errors in threads to get a sense of the amount of synchronization overhead. It turned out that removing the lock leads the speedup of the Radix Trie to near linear similar to the lock-free variants.

Run Time Without Lock (s)	Run Time With Lock (s)
8.69	11.84
4.49	7.44
2.64	5.33
1.54	5.60

Table 1: Speedup of Fine-grained Locking Trie on GHC

Contention also impacts the performance of the lock-free radix trie implementation. Although experimental results show that this implementation performs the best, achieving up to a sixfold speedup with eight threads, it still falls short of linear scalability. This is primarily due to contention. Specifically, when multiple threads attempt to update the same node using 'strong_compare_exchange', some threads fail and must retry, which increases the number of instructions compared to a sequential version. This conclusion is supported by experimental data, not merely speculative. To confirm this, we tracked the number of retries for the insert operation in our lock-free script.

Number of Threads	Retry count
1	0
2	1764
4	4029
8	7193

Table 2: Retry Count of Lock Free Insertion on 420,000 words

5 References

- Taku Oka. GloVe: Global Vectors for Word Representation (840B, 300d). Available: https://www.kaggle.com/datasets/takuok/glove840b300dtxt.
- dwyl. A List of English Words: https://github.com/dwyl/english-words/blob/master/words.txt.
- "npgall/concurrent-trees": https://github.com/npgall/concurrent-trees.
- ACM Digital Library. "Title of the Article" (Access the PDF directly via https://dl.acm.org/doi/pdf/10.1145/2555243.2555256).

Work Distribution: 50%-50%