Parallel Programming Course Project Report

Lukas RABIER

Nicolas BLIN

Pierrick MADE

Théo LEPAGE

June 2020

Contents

1 Introduction

This report will compile what has been done for the PRPA project, by the group composed of Lukas R., Nicolas B., Pierrick M. and Théo L.. The project started on June 10th, 2020 and was due on July 12th, 2020.

The aim of this project is to build a simple search engine over a database of documents (books). Each document has many words. Only three operations are needed : search a word in the database to return which books contain it, insert a document and his words in the database and remove a document from the database. They have to be as fast as possible and support concurrent running.

This report is divided in three parts, the first implementation using a hash map, the implementation using concurrent tries and the asynchronous process.

2 First strategy: the hash map

Our dictionary is based on two hash maps: one associating a word to documents and one associating a document to words. This organization of the data seems to be the most convenient for a many-to-many relationship. The implementation differs from the naive dictionary as we use our own concurrent hash map and we store values in vectors instead of sets. Furthermore, small optimizations were made by adapting the behavior of our code according to the subject's problematic.

2.1 Architecture and data structures

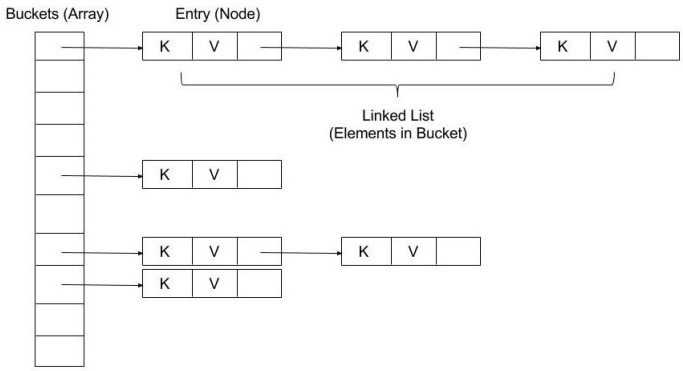


Figure 1: Representation of the hash map data structure

Our hash map consists in a vector (`std::vector`) of a fixed number of buckets. The size of the vector is 8192, which was chosen empirically. Each bucket is a shared pointer to the sentinel of its linked list (cf.

Figure 1). Our collision resolution method relies on those linked lists because if two elements have the same hash, they will both be stored in the same bucket. The hash function used is `std::hash`, from the Standard Template Library, which permits a good distribution of values in the data structure.

Using a sentinel allows us to ignore the special case when a node is the first element as we do not have to create locks for the vector itself. Finally, a node is a simple class containing a key, a value, a shared pointer on the next node and a shared mutex.

2.2 Our fine grained locking strategy

The aim is to reduce the number and the scope of locks on resources to improve the speed of the program. Thus, we decided to rely on **chain locking** to lock at most two nodes instead of a whole bucket. The principle of hand-over-hand traversal is to go through the linked list of the bucket while locking the next element before unlocking the current one. Furthermore, we combined chain locking with the use of a shared mutex for a **read-preferring implementation**. Our class `forward_lock_guard` allows us to handle the different traversals of the linked lists by relying on RAII (Resource Acquisition Is Initialization) to prevent missing an unlock leading to a deadlock.

Although we created functions that handles multiple operations at the same time to prevent doing multiple lockings (e.g. insert a value in a node or create it if it does not exist), the main idea behind our algorithm is described below.

Book search During the process of fetching information from the hash map we apply a shared lock according to the hand-over-hand traversal. As this operation is always read-only, our program is able to do multiple searches concurrently as long as previous nodes in the bucket are not being modified, inserted or removed. We release the lock as soon as we find the node and return a copy of the value to prevent data races.

Book insertion For the insertion of a new pair of key/value the difference is that we use a unique lock to traverse the linked list. Depending on the case, we insert the new node just after the sentinel if we already know it does not exist or at the end of the linked list if we were looking for a pre-existing node. If multiple values will be inserted in the value of a node, our class `forward_lock_guard` allows us to maintain the lock during the whole process.

Book removal During a deletion, we traverse the linked list with a unique lock and we maintain the lock on the node preceding the one to remove. This way, as all the other operations, the deletion is thread safe.

2.3 Other locking methods considered

We have considered using other methods to allow concurrent access to our hash map. However, most of them were not giving interesting results as opposed to the strategy described in the previous section.

Optimistic locking The principle of optimistic locking is to check if the node was updated by another thread before applying the modification. This method is well suited for long lists as it uses less locking which are very time expensive. However, in our case the number of documents and words would need to be really high to create long linked lists. Our buckets contain few nodes and thus a lock blocking operations on next nodes is infrequent.

Atomic locking The idea behind atomic locking is to remove locks by using `std::atomic` pointers. It is much more simpler and faster on insertions and searches operations. However, the deletion is more complicated to implement because we have to deal with the ABA problem. We tried this implementation but we struggled on some issues with atomic shared pointers in the removing process.

2.4 Results and interpretations

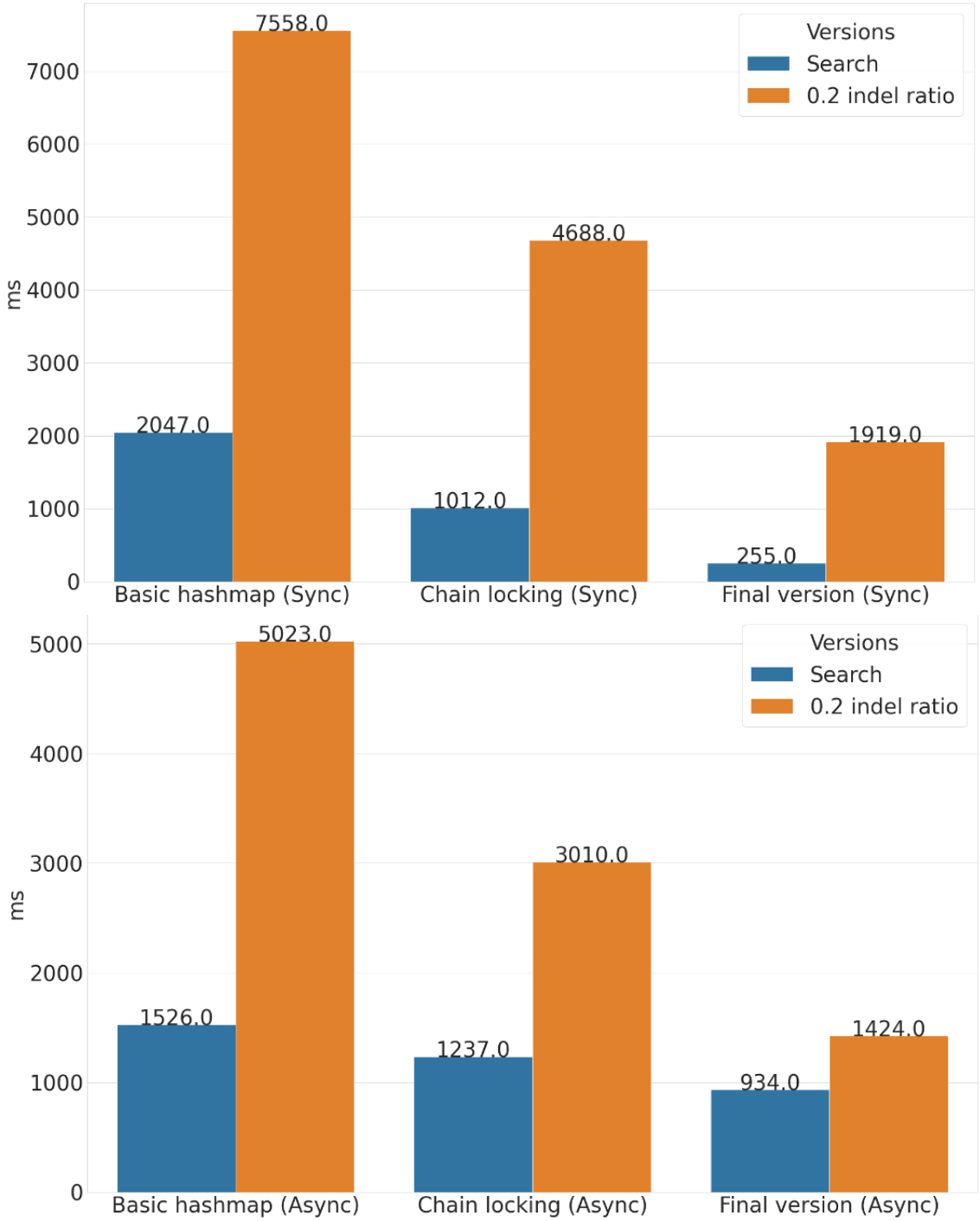
The difference between the benchmarks of three stages of our implementation is shown on the figure below. They were ran once on 1 000 000 search queries and then on 800 000 search queries and 200 000 insertion/deletion queries. We can notice that the **Basic hashmap** with fat locks is much slower compared to the **Chain locking** implementation and the **Final version** which removes several useless locks.

It is noteworthy that some optimizations not concerning directly the hash map were developed. As an example, the data is pre-processed to ignore duplicates words before inserting them as we now use vectors ; we keep pointers on the hash map nodes to reduce linked lists traversals and thus locks and we make sure to extend lock scopes instead of locking several times the same resource.

As a result our implementation is way faster than the naive dictionary even in sequential where our locks could lead to additional time during operations. The difference of processing time is more noticeable as the number of queries increases. For instance, on my laptop and with 1000000 queries and 10000 words, in asynchronous our implementation handles 827 301k items per second compared to 216 299k items for the naive dictionary. Therefore, ours is nearly four times faster.

According to Gprof the function to create a node or insert a value if it exists, is the most time expensive as the node is locked while data is being inserted. A solution to this problem would be to replace the vectors in the nodes value by an other concurrent data structure.

Figure 2: Benchmarks of hashmap implementations



3 Second strategy: concurrent trie

Definition

In computer science, a trie (also called prefix tree), is a tree data structure used to store a dynamic set of strings in a specific fashion (Figure 3). In this project's case, it is useful in order to mark the presence of books containing a word directly in a node of the trie.

3.1 General architecture

- Words are marked by a succession of different nodes
- Marked nodes contain a vector structure containing all the books they are in
- Hashmap linking a book to the sub-nodes in which it is referred, to ease deletion
- Shared lock during search, unique lock during book insertion and removal

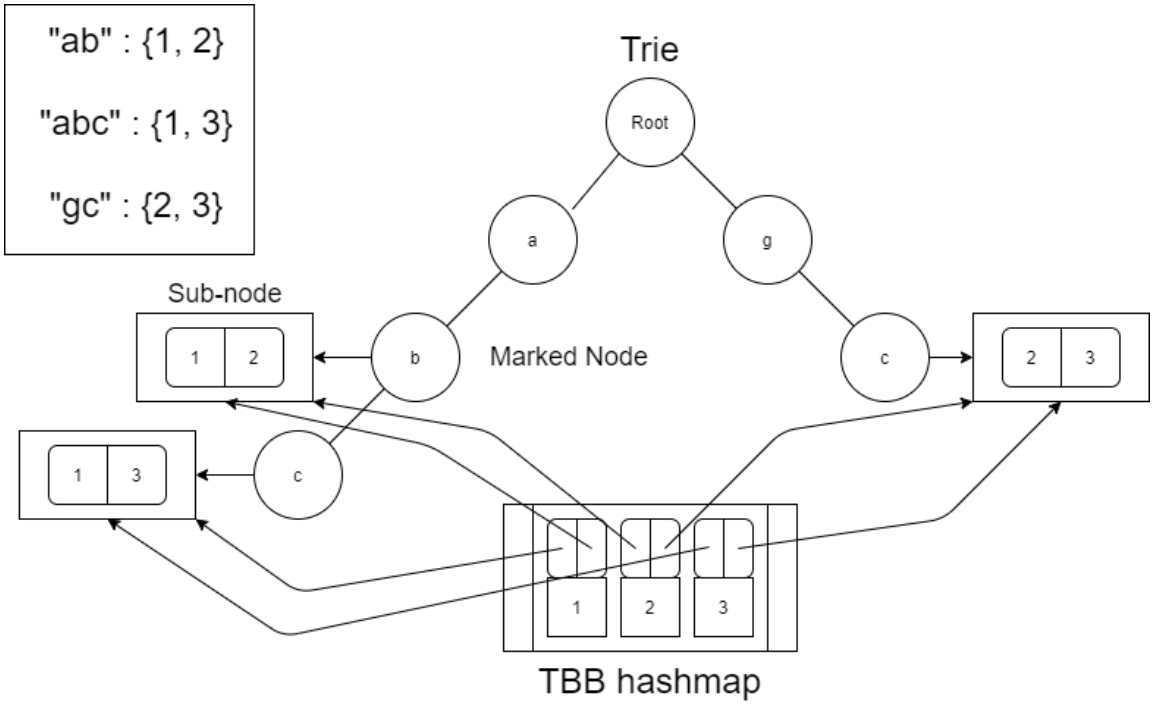


Figure 3: Our Trie Structure

3.2 Implementation

Book search During the process of fetching information from the trie, we loop through the letters of the input word. To jump from one node of the tree to another in constant time, we use a fixed size array representing the 26 letters of the alphabet. If a node is empty in the array, we return.

When reaching the end of the word, we fetch the sub-node structure only contained in nodes marking a word. Once inside it, we share_lock the mutex inside to prevent any other thread from writing or deleting inside our vector structure containing the books. Another thread wanting to read at the same time will be allowed to since it is a shared lock. After locking, we simply copy the content of the word's book vector in a result record and return it.

Book insertion During a book insertion in the structure, we go trough our Trie the same way we did in search. During a book insertion, some words may need to be inserted too. To cope with that problem, every node contains a mutex. If a new word is added (i.e. an entry in our [26] array is empty at some point, or the final letter node is not marked yet) we lock the node and add a new node at the right index.

Once we reach the end of the word, we lock the sub-node part of the node. Not locking the node itself allows other thread to still traverse the tree. The new book is added to the sub-node vector. There a unique_lock is used to prevent concurrent write/delete on the same sub-node.

During this process of insertion, we also modify a TBB hashmap [(6)]. Indeed, to speed-up the deletion, we use an hashmap containing, for each book, all the sub-nodes containing the book. During insert, we add a new entry to the hashmap and fill its vector with references to the nodes of the words in this book.

We also use this hashmap to optimize the insertion process. Indeed, since books cannot be added twice in the structure, we start by calling find in our hashmap. To optimize we also remove duplicates from the input words.

Book removal During a deletion, we simply retrieve the document's entry. Again, thanks to our hashmap, if the document is not there, we break directly. We then traverse the vector of sub-nodes, removing the book's id from their vector before deleting the entry itself of the hashmap. During this step, each sub-node is locked while it's being processed, and the hashmap's entry is locked during the whole process.

3.3 A word on thread safety

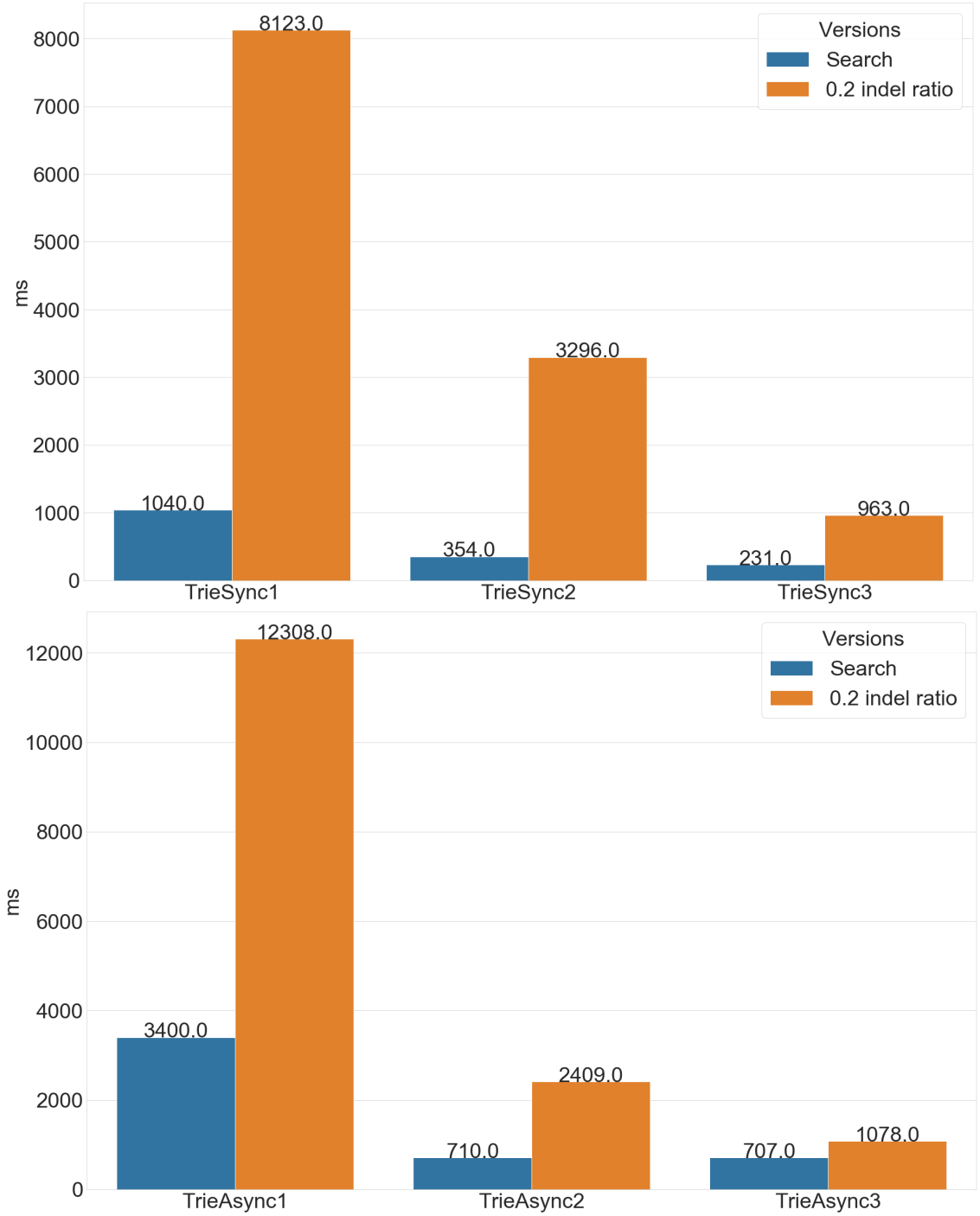
Each sub-node contains a mutex. On reads (search) shared_lock is called while unique_lock is called on insert and delete. The structure (vector) containing the books is thus protected from concurrent modifications.

Each node in the tree contains a mutex. This is used to make the addition of a new word (and thus of new nodes) thread safe. For example if "ab" was present and we want to add "ac" we need to add a new entry to our [26] array. The node "a" is locked (unique_lock on the node's mutex) and "c" is added. We use the "double-if" technique to avoid having two threads wanting to add at the same index at the same time. We only need to lock on writes and not use shared_lock on reads since the accesses are made at different indices of the array and thus different memory locations. The only concurrent writes will happen in empty cells, no read can happen in those cells (at the same time). If two writes happen in different cells, they will be sequential and thus safe since there is only one mutex per node. If two writes happen in the same cell, the "double if" technique protects us from any double assignment problem. We could argue that a data-race happens on the array as a whole but as we found [4;3;5] accessing different index of an array concurrently doesn't cause any thread-safety issues as long as concurrent writes are locked at the cell level (and only reads are done in the other cell). To be extra sure, we ran really long (several minutes) async consistency tests. The goal was to check that both structures (sync and async) ended up with the same content even after millions of search/insert/delete made by multiple threads for one, and sequentially for the other.

The hashmap we use to track the leaves associated to each book is implicitly thread safe since we are using TBB's. While we have the accessor (key, value) no other thread can come, so we can safely unique_lock each sub-node, remove the book and at the end, remove the entry from the hashmap.

3.4 Benchmark Results

Figure 4: Benchmarks of three Trie implementations



The benchmark was ran on two different tests: 1 000 000 search queries, and 800 000 search queries and 200 000 insertion/deletion queries. They were ran on three evolutions of our Trie implementation.

- **TrieSync1** is the first implementation: each node has its children in a vector (no direct access in a [26] array), books are stored in a set, no deletion hashmap. Deletion means traversing the entire trie
- **TrieSync2** is the second implementation: we optimise the general structure with direct access to children thanks to a 26 sized array and the addition of a deletion hashmap. Searches are 3 to 4 times faster and deletion is 5 times faster.
- **TrieSync3** is the final implementation: replacing the set structure in marked nodes by a vector and adding algorithmic optimisation to insertion and deletion. It is twice faster as the previous one on average.

4 A thread pool to make the program asynchronous

To make the calls asynchronous, we use TBB's task_group. When a function is called (search, insert, delete) a lambda wrapping the actual function is sent to TBB's task_group. We pass a promise to our lambda and return the associated future. TBB's documentation states that adding a lot of tasks to one thread_group is not scalable. We tried to dispatch the task to different task_group but it did not improve performance.

5 Conclusion

To conclude, during this project, we have achieved to build two different concurrent and thread-safe data structures to store books and their contents, allowing search, insertion and deletion.

We achieved to beat the naive implementation, both synchronous and asynchronous, with both the Trie and Hashmap implementation.

We tried implementing a last implementation: replacing the concurrent TBB hashmap in the Trie implementation by the concurrent hashmap of our team. This merge of two implementations was promising since our hashtable is supposed to be lighter than TBB's, and could thus be quite faster in our project. However, we were sad to see than this implementation's performance were actually not that higher.

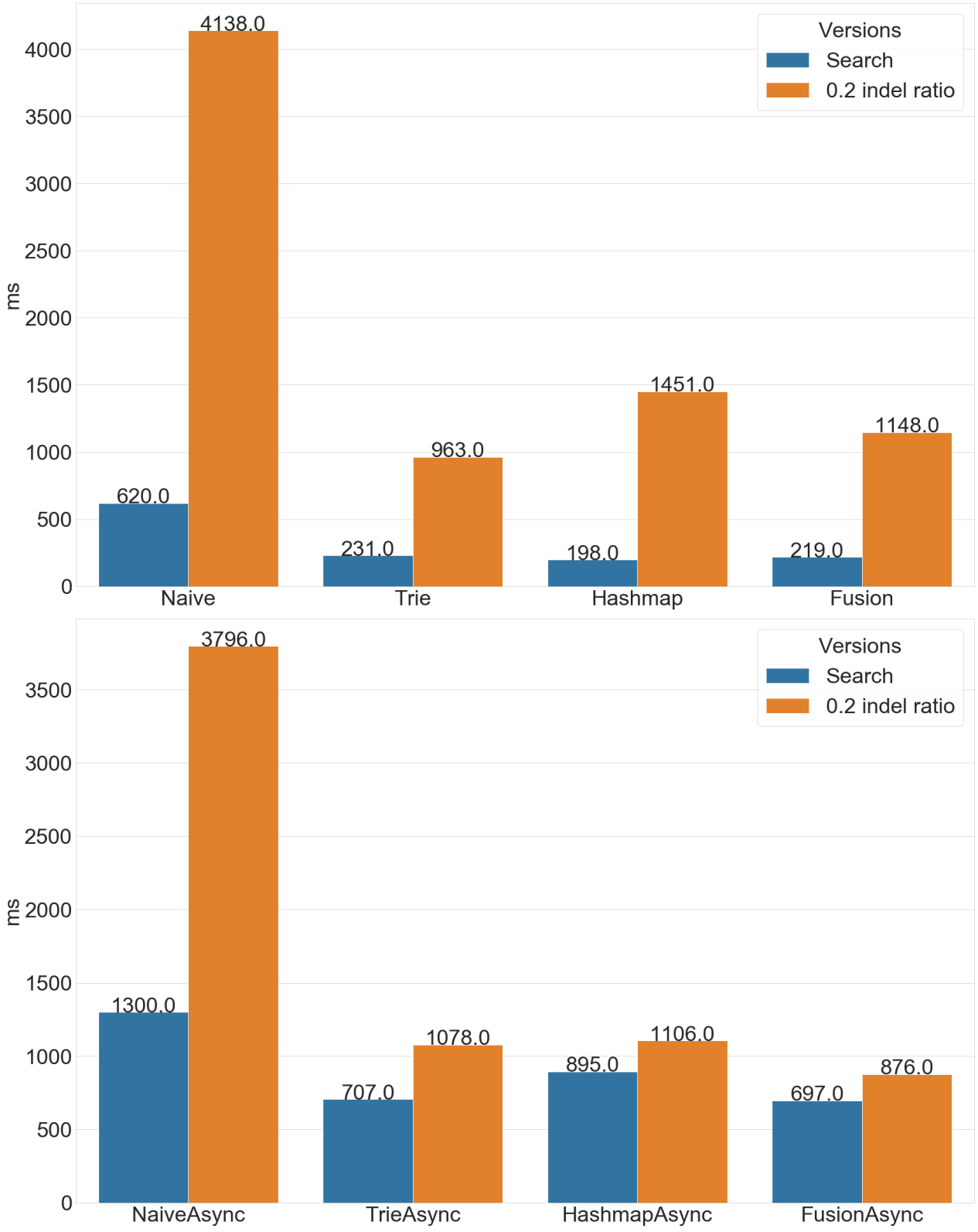


Figure 5: Benchmark of our three implementations

References

[1] Carlinet, Edwin: *Cours de PRPA*. – URL https://www.lrde.epita.fr/~carlinet/cours/parallel/slides.html?j=5#1

[2] cppreference: *std::shared_mutex*. – URL https://en.cppreference.com/w/cpp/thread/shared_mutex

[3] Danes, Mike: *Is a simple array thread safe?*. – URL https://social.msdn.microsoft.com/Forums/vstudio/en-US/7781a7fc-9964-4d3f-883d-f19935d52eb9/is-a-simple-array-thread-safe?forum=netfxbcl

[4] Gary: *Are C# arrays thread safe?*. – URL https://stackoverflow.com/questions/1460634/are-c-sharp-arrays-thread-safe

[5] Robert: *Is writing to an array of floats from 2 different threads at two different indexes safe?*. – URL https://stackoverflow.com/questions/20458673/is-writing-to-an-array-of-floats-from-2-different-threads-at-two-different-index

[6] Software, Intel: *TBB concurrent_hash_map Template Class*. – URL https://software.intel.com/en-us/node/506191

[7] Wikipedia: *Non-blocking linked list*. – URL https://en.wikipedia.org/wiki/Non-blocking_linked_list#Insert