Concurrent skip list

Isa Lie*, Raghu Radhakrishnan†

15-418/15-618: Parallel Computer Architecture and Programming, Fall 2024

*hlie@andrew.cmu.edu, †raghur@andrew.cmu.edu

I. Summary

The aim of this project is to analyze the parallelism capabilities of various skip list implementations. We first implemented a sequential access skip list as a baseline. We subsequently implemented three concurrent versions: coarse-grained, fine-grained, and lock-free. To compare our implementations, we showcased their scalability by measuring both wall-clock speed and speedup as thread count increases. We utilized both the GHC machines and the PSC machines to aid in our investigation.

II. Background

A. The Skip List Data Structure

Skip lists are data structures used in Database Management Systems (DBMS) for database indexes. It is commonly desirable to enable concurrent access to a database index in order to support query parallelism and ultimately speed up database queries. This motivates our investigation into various concurrent skip list implementations and examine their scalability on parallel workloads.

Skip lists are probabilistic data structures that maintain a sorted set of elements, allowing efficient insertions, deletions, and searches with average complexity $O(\log(n))$. They achieve this by using multiple linked lists, forming a hierarchical structure that facilitates fast traversal and searching. Conceptually, skip lists can be thought of as a multilevel extension of a sorted linked list, with higher levels ”skipping” over more nodes, thereby allowing for rapid access. Elements

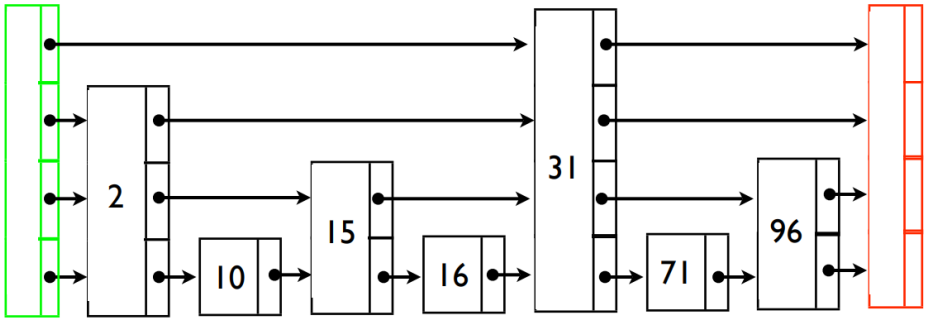

Fig. 1: A skip list with a head and tail tower. [1]

with the same value at different levels form a ”tower”, and a tower’s height is determined by multiple coin flips. Each successful coin flip ”promotes” the height until a ”tail” is flipped.

On a very basic level, each node in a skip list contains a key, a pointer to the next node of the same level, and a pointer to the next node in the same tower. It is worth mentioning that sometimes a node will use an array of pointers to represent the tower, each index symbolizing a node at a different level. For the sake of simplicity, all node keys used in this project will be integers, and we do not insert nodes with duplicate values.

The skip list data structure has a pointer to the head, which is the first node of the skip list with a value of negative infinity. This node is never removed. In some cases, there is also a tail tower with a value of positive infinity, and it is also never removed. Figure 1 diagram showing this data structure.

The key operations of a skip list are as follows:

- `contains`: Returns whether a given key exists in the skip list. The program starts at the topmost level and traverses the nodes until the key is greater than the target key. The program then drops down to the next level and repeats until the target key is found or the bottom level is reached.
- `insert`: Inserts a new value into the skip list. The node’s levels are chosen based on a random coin-flip strategy.
- `remove`: Removes a value from the skip list. The program traverses the levels to locate the node and removes its references from all relevant levels.

The computationally expensive component is the traversal of the list. Each search, insertion, or

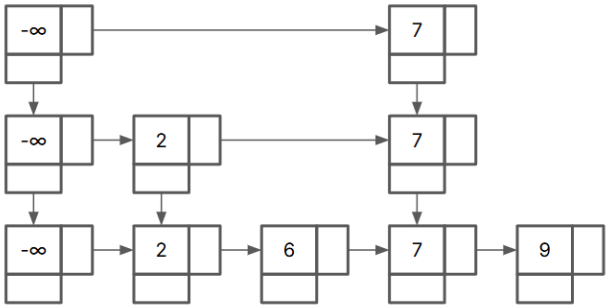

Fig. 2: A diagram of an example sequential skip list

deletion operation requires traversing multiple levels to locate the target node or insertion point. Fortunately, each operation can be parallelized because different threads can work in separate regions of the skip list. However, operations on overlapping regions introduce dependencies and require synchronization mechanisms.

III. Approach

All implementations were done in C++ 17 using modern features such as smart pointers and RAII in order to ensure proper memory management and avoid memory leaks. All code associated with this project can be found here.

A. Coarse-Grained Lock skip list Implementation

The coarse-grained approach to parallelizing a skip list is the most naive way in converting a sequential skip list to a thread-safe one.

To implement the coarse-grained lock skip list, we first implement a sequential skip list. Each sequential skip list node has a down and a next pointer. We construct the skip list by initializing the tower of head nodes. When inserting each node, we find where to insert the node at the lowest level, decide the height of the tower, and change the next and down pointer of all relevant nodes. Figure 2 shows a sequential skip list implemented in this matter.

The coarse-grained lock skip list simply contains a sequential skip list and a global mutex lock. Before each thread accesses the sequential skip list, it locks the mutex, preventing any other threads from editing it thus ensuring correctness.

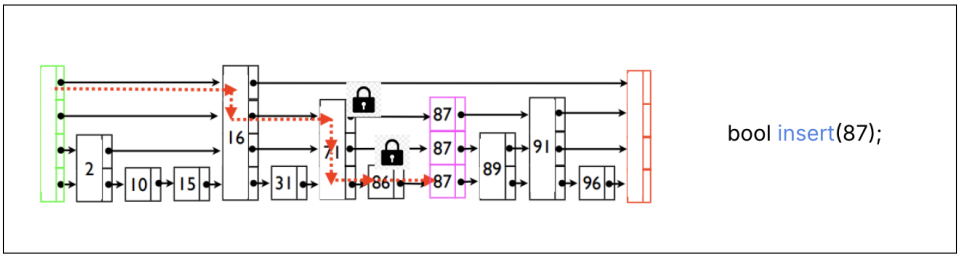

Fig. 3: Example of an insertion into a fine-grain skip list

B. Fine-Grained Lock skip list Implementation

Our fine-grain approach utilizes an optimistic approach on insertion and deletion [2]. In this approach, each tower has its own lock, where a tower is represented as an array (with size equal to the number of layers of the tower) of next pointers to subsequent towers. In addition, each tower contains two flags: `fully_linked` used for insertion and `marked` for deletion.

Figure 3 displays an example of a simple insertion. Observe that when inserting tower 87, we first grab locks of all previous towers that will need to connect to the new node.

As mentioned prior, this algorithm is an optimistic approach. On insertion, we first search the data structure for the value without acquiring any locks. During this search, we will also obtain all previous and successor nodes for which the new tower should be inserted between (unless the value already exists in which case we abort). We then perform the insertion by acquiring the locks on the appropriate previous towers only if the previous tower is not deleted (using the field `marked`) and it is in a valid state where all its current successors are connected (`fully_linked`). If these conditions are not met, we simply retry. Finally, we set the next pointers for the new node and set the new node's `fully_linked` flag.

On deletion, we again search for the node of interest without acquiring any locks. If found, we mark the node as logically deleted. This allows other queries to use the current node to traverse the skip list without physically deleting the node and still appearing as deleted. We then perform the deletion process by acquiring the locks on previous towers and setting the previous towers' next pointers to the appropriate towers.

This being an optimistic approach, when there is little contention and each thread operates on different regions on the list, the scalability of this solution is excellent. However, the use of

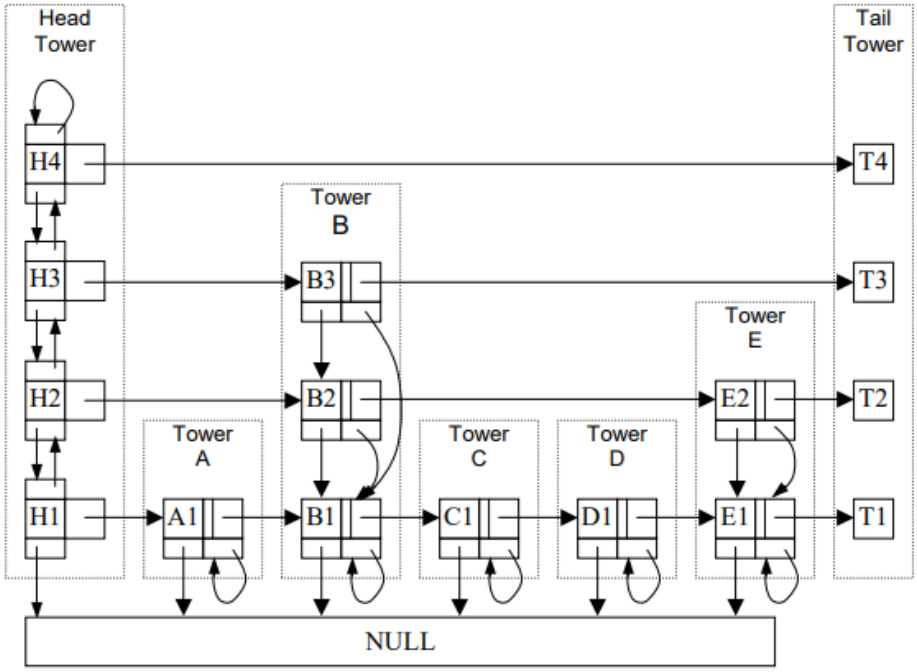

Fig. 4: Lock-free skip list design. From Figure 24 of [3]

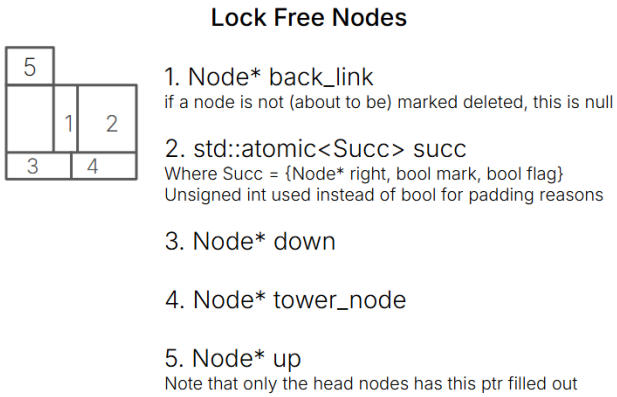

Fig. 5: A node of the lock-free skip list

locking still introduces overhead as we will examine in further detail under analysis.

C. Lock-Free skip list Implementation

The lock-free skip list is implemented based on the pseudocode from [3], and a diagram is shown in Figure 4. Aside from the normal pointers used by the other nodes, there are also a few more notable variables in the structure of the lock-free node, as shown in Figure 5.

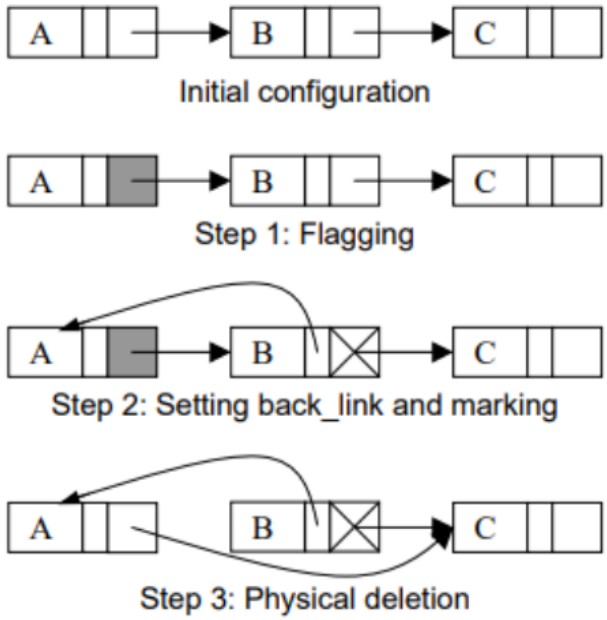

Fig. 6: The three step deletion process on the lock-free skip list. From Figure 9 of [3]

The most important among them is the variable `succ`. It is updated via the atomic Compare-And-Swap operation, which, with the mark and flag field of the `Succ` data structure, ensures that there is no race condition. The next node of a flagged node should be marked and then deleted. As CAS compares all three fields of the `Succ` structure, it ensures that no valid node's next node will be set to a superfluous node. Figure 6 shows the three step deletion process inside the data structure using the mark and flag field. More specifically, a node after a flagged node is superfluous, and should be marked and deleted. When traversing the skip list, we can access the previous valid node of any superfluous node via the $back\_link$ pointer.

While implementing the lock-free skip list based on the pseudocode in [3], several challenges not mentioned in the paper were encountered. These roadblocks required additional modifications and insights to ensure the correctness and efficiency of the implementation.

First, the pseudocode contained an error in the conditional statement of the `InsertNode` function. Specifically, on line 10, the condition incorrectly required the return value of `CAS` to be the new value in order to indicate a successful compare-and-swap operation. This was corrected to ensure proper functionality.

Second, while not explicitly illustrated in the diagrams, the head and tail towers of the skip

list needed to be one level higher than the maximum height of the skip list. This adjustment ensured that there was a level where the head node pointed directly to the tail node. Without this modification, the `FindStart_SL` function could potentially enter an infinite loop.

Third, `std::atomic` requires its type to be trivially copyable. This presented a challenge because `shared_ptr`, which was used for memory management in other parts of the skip list, does not meet this requirement. To address this, raw pointers were used for successor pointers in the lock-free implementation, ensuring compatibility with `std::atomic`.

Lastly, an unexpected issue arose with the `compare_exchange_strong` function in C++. The function consistently failed, even when the expected and current values appeared identical. Upon inspecting the memory layout, it was discovered that the Succ data structure, originally declared as `ptr, bool, bool`, introduced 16-byte alignment due to padding. This led to potential discrepancies in the padding bytes, causing `compare_exchange_strong` to fail. The problem was resolved by changing the data structure to `ptr, unsigned int, unsigned int`, which fully occupied the 16 bytes and eliminated padding-related inconsistencies.

IV. Benchmarking

A. Correctness

The first step in testing our implementations was ensuring their correctness. Firstly, we ensured that each implementation functioned appropriately in a single-threaded context. To do so, we compared our structure to a `std::set` data structure on a 100000 random `insert`, `delete`, and `contains` operations. In addition, we also introduce a `validate` function to our skip list interface which checks that the data structure maintains all skip list invariants.

To test our structures in a concurrent environment, we simply partitioned the queries into 8 threads and ran them on the skip list at the same time. Afterwards, we ensured that all skip list implementations were still in a consistent state.

B. Performance

We benchmark our implementations on a set of 100000 random `insert` and `delete` operations. Our testing suite inputs a number of threads and the operations are evenly partitioned

across the threads. We then wait for all threads to finish and use the wall clock time to determine the performance of the skip list.

V. Results

A. GHC Machines

We first examine the results from the GHC machines as seen in Figure 7. The coarse-grain skip list behaves just as expected. We observe that as the thread count increases, we actually experience slowdown. Since the coarse-grain skip list utilizes a global lock in order to make modifications to the data structure, this data structure essentially offers sequential access. Thus, the overhead of multiple threads and lock contention causes the program to incur additional slowdown.

The more interesting results are those of the fine-grain and lock-free skip lists. In terms of raw wall clock time, the lock-free skip list is more performant in all situations. This result is most likely due to the lesser overhead of the CAS operation compared to obtaining locks. However, in terms of scalability, the two implementations behave almost identically. We observe that until 4 threads we are able to achieve almost perfect speedup. However, past 4 threads we incur more contention on regions in the skip list causing more retries (CAS failures) and lock contention. Past 8 threads, we observe that both implementations experience slowdown from 8 threads. This behavior is most likely due to machine limitations: we have only 8 threads available on the GHC machines so additional context switching from the OS causes greater overhead. Thus, to further investigate these data structures, we exploit the high thread counts of the PSC machines.

B. PSC Machines

Figure 8 displays the results from the PSC machines up to 64 threads. Firstly, we again observe the same results from the coarse-grain skip list as on the GHC machine. Moreover, we also see that the lock-free skip list out performs the fine-grain skip list at all thread counts.

The interesting result is that the speedup is much lower on the lock-free skip list when compared to the fine-grain skip list at all thread counts. One potential reason for this could be an architectural one: it is possible that the loops containing the CAS operation does not yield the

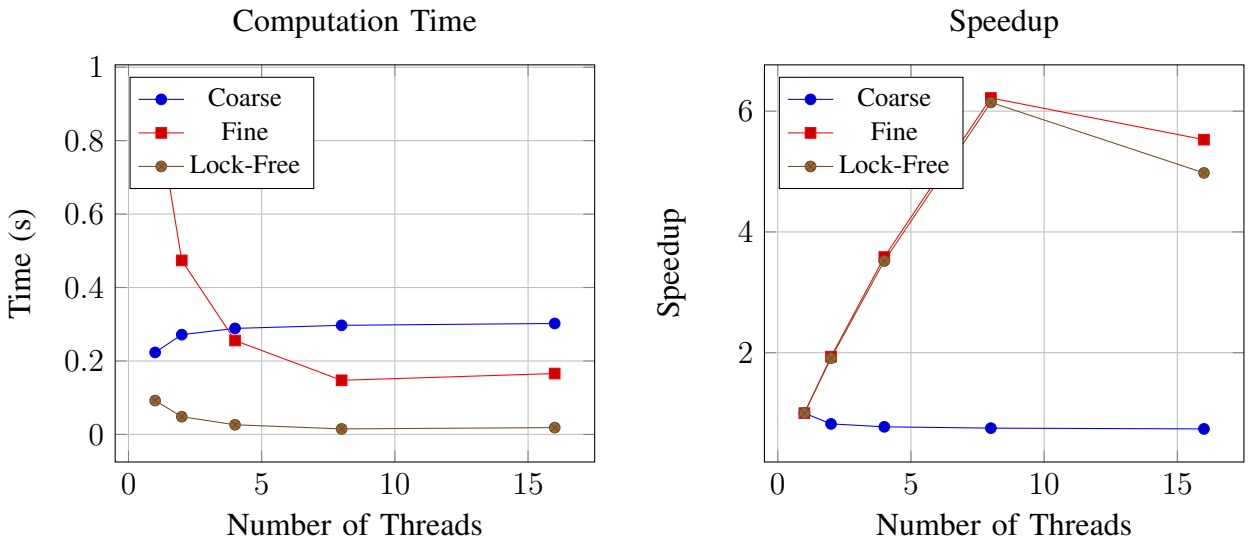

Fig. 7: Performance comparison on GHC Machines: Time taken and speedup.

same performance as it does on the GHC machine which inhibits the lock-free implementation to incur the same level of speedup. However, once we reach 64 threads, speedup decreases on the fine-grain skip list while it is still increasing on the lock-free skip list. This showcases the efficacy of the CAS operation when compared to lock acquisitions: it is possible that the lock-free implementation can get additional speedup from even more threads, while we experience diminishing returns at high thread counts on the fine-grain implementation.

Overall, while speedup gains are not as evident on the lock-free approach on the PSC machines, it is still clearly the most performant approach in all scenarios.

VI. Future Extensions

Ultimately, we concluded that the lock-free skip list does in fact offer the best performance in terms of wall-clock speeds on randomized workloads in all multi-threaded contexts. However, there is still further experimentation required to fully ascertain any insight into the best approach when implementing something such as a database index. For example, it is common that database workloads contain some aspect of locality. Thus, it may be worth investigating these implementations on better work partitioning strategies instead of just random allocation. Moreover, further experimentation should be conducted to obtain quantitative results with regards to the source of the time delta between the fine-grain and lock-free implementations.

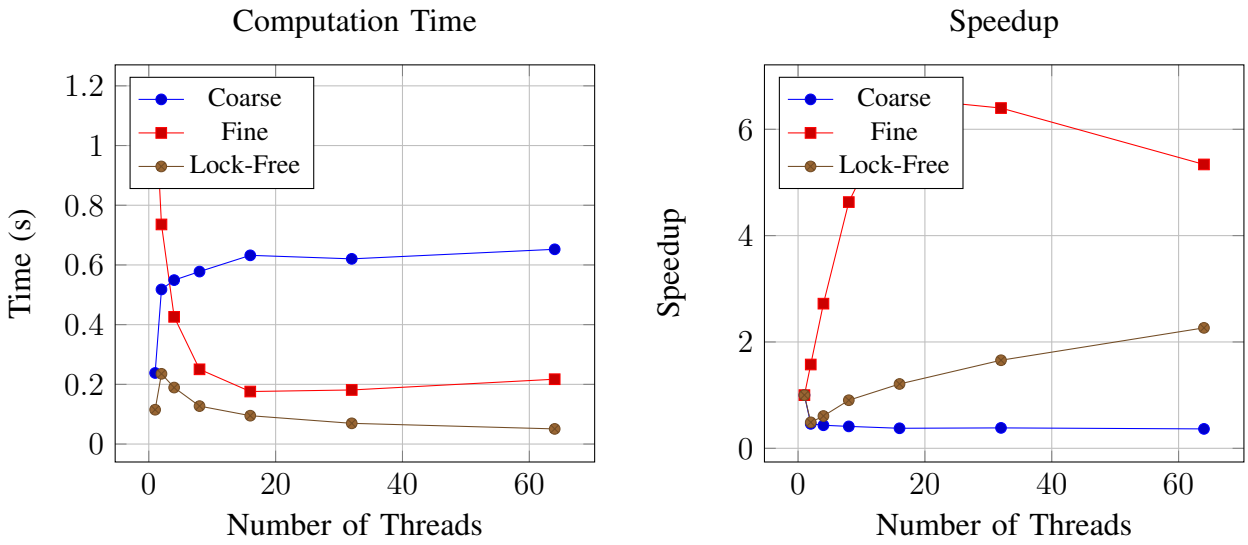

Fig. 8: Performance comparison on PSC Machines: Time taken and speedup.

Tasks	Members
- Research Topic	Isa & Raghu
- Project Proposal	Isa
- Data structure simulation Script and Code	Isa
- Implement Linear Skip List	Isa
- Data structure simulation correctness verification	Raghu
- Implement Coarse-Grained Lock Skip List	Isa
- Project Milestone Report	Raghu
- Implement Lock-Free Skip List	Isa
- Implement Fine-Grained Lock Skip List	Raghu
- Benchmarking Script	Raghu
- Benchmarking and Performance Analysis	Raghu
- Poster	Isa & Raghu
- Final Project Report	Isa & Raghu

Fig. 9: Work completed by each member of the group

VII. List of Work

See Figure 9 for a list of the work completed by each member of the group. We feel it is appropriate to divide the total credit 50-50.

REFERENCES

[1] C. M. University, "Skip lists," Available at https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/skiplists.pdf.

[2] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A simple optimistic skiplist algorithm," *Lecture Notes in Computer Science*, 2007.

[3] M. Fomitchev, "Lock-free linked lists and skip lists," *Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, 2003.