Comparing the Performance of Concurrent Lock-based and Lock-free Hash Tables

1st Minh Dang Truong
Department of Computer Science
Denison University
Granville, Ohio
truong_m1@denison.edu

2nd Uyen Le
Department of Computer Science
Denison University
Granville, Ohio
le_u1@denison.edu

***Abstract*—Hash tables are a data structure that provides the foundation for techniques like caching and database indexing. The main advantage of the data structure is that it can perform insert, delete, and search operations in constant time in the average case. However, hash tables must be thread-safe to be used in multi-threaded programs. Thus, concurrent hash tables are key to having multiple threads access shared data in a safe and efficient way. In this paper, we present and compare three different implementations of concurrent hash tables using hashing with chaining, including coarse-grained parallel hash tables, fine-grained parallel hash tables, and lock-free parallel hash tables. Evaluations on a varying number of threads suggest that lock-free hash tables are more efficient and scalable in most cases.**

***Index Terms*—hash tables, locking, concurrency, lock-free, synchronization, parallel computing**

I. Introduction

A hash table is a dynamic, associative data structure that stores a collection of key-value pairs such that each key appears at most once in the table. The data structure supports efficient "insert", "delete", and "search" operations by maintaining a hash function that separates each key into a separate "bucket" of the array. A few popular hash functions include the decision method, the multiplication method, and the universal hashing [1]. During look-up, the hash function computes the hash value from the key and locates the bucket using the resulting hash value. Under reasonable assumptions of the hash function, hash tables are guaranteed to perform in constant time, and outperform alternatives in most situations [1].

Given its associative nature and constant amortized time access methods, the hash table data structure is a building block for various software. In computer systems, hash tables are used to implement caches [2], the auxiliary hardware and software components that enable fast data access. In database management systems, hash tables also emerge as disk-based data structures and database indices [3]. Other applications of hash tables lie in key-value stores (such as Memcached, Redis), networking [4], genomics [5], and memoization [2].

In many of these applications, hash table accesses constitute a large portion of the actual running time [6]. Thus, it is essential to maintain highly scalable, concurrent hash tables that can handle a large number of concurrent requests. However, the sequential implementation of the hash table does not guarantee accuracy due to potential race conditions. Race conditions happen when two or more threads attempt to access and modify the same resource at the same time and cause unwanted behavior in the system. Hence, we need to provide hash tables with proper synchronization methods. However, as with other concurrent data structures, concurrent hash tables also suffer from problems like deadlock, starvation, etc that can negatively affect the performance of the hash tables. Hence, until these problems are addressed, we will not see the speedup as the number of threads increases but the overhead.

In the past, there have been several studies on the implementations of concurrent hash tables to mitigate the effects of concurrency-related problems while preserving the correctness of operations on the hash tables. One approach that we consider in this paper is the coarse-grained hash table. It has a single global reader-writer lock to perform all the work and ensure that concurrent read and write requests to the tables are handled separately. Specifically, multiple threads can read the shared data in parallel, but an exclusive lock is required to write to the data. Thus, only one thread is allowed to modify the data at a time. However, aggressive locking can have negative impacts on the performance of the system [6], so we take a look at another variation of the lock-based concurrent hash tables that uses the lock at a finer granularity. Instead of a global reader-writer lock, a fine-grained hash table has a reader-writer lock for each bucket. These local locks are in charge of reading-writing operations to that bucket, while the global lock is only responsible for growing the hash table. We expect to have more parallelism with finer-grained locking. Finally, we look into a more efficient, lock-free solution to the concurrency issues, as presented in [7], where the author demonstrates an atomic hash table that leverages a compare-and-swap-(CAS)-based list-based set algorithm. At a high level, the CAS algorithm modifies the contents of a memory location if and only if they are equal to a given value. The algorithm should be done as a single atomic operation to achieve synchronization. Comparing different variations of parallel hash tables on varying number of threads implies that lock-free hash tables outperform their counterparts most of the time.

II. Background

A. Hash Table Data Structure

The hash table data structure is an unordered, associative collection of key-value pairs that performs insertion, deletion, and retrieval with amortized constant time, assuming a proper choice of hash functions. Typically, a hash table is implemented using an array indexed by the hash value of a key. In particular, given a key k and a hash function $h(x)$, then $h(k)$ will return a slot number, also known as a bucket that stores the data value corresponding to k. Ideally, we want h to produce one unique bucket for each key. However, existing hash functions are imperfect and, hence, a source of conflicts when multiple keys are assigned to the same slot. This issue is often referred to as hash collisions.

One common solution to hash collisions is hashing with chaining. Specifically, when a hash function assigns the same value to multiple elements, the collided items will be chained through a single linked list that can be traversed to access the item with a unique search key. In collision resolution by chaining, a new element is inserted at the head of the linked list at the bucket of the hash key. A deletion searches for the item in its corresponding bucket and removes the item from the list if it exists.

In addition, the load factor of a hash table is defined as the number of elements in the hash table divided by the number of slots. The performance of the hash table is negatively correlated with its load factor, so a hash table is usually resized as the load factor approaches 1. A well-chosen hash function and a constant average load factor guarantee that a hash table performs in expected constant time [7]. Common hash functions involve hashing by division and hashing by multiplication. Other works have proposed to use universal hashing [8] and minimal perfect hashing [9].

B. Concurrency

In the scope of this paper, concurrency means the ability to handle concurrent requests by different computer threads to the shared data. The parallel execution of multiple computations can result in an increase in the overall performance of the system. To be able to quantify the performance increase, we define the parallelism of a computation as the time to execute the computation with just one processor (T_1) divided by the time to execute the computation with an infinite number of processors (T_∞) [1]. The speedup of a computation on P processors is defined as the ratio of the time to execute the computation on one processor (T_1) to the time to execute the computation on P processors (T_P) [1].

In parallel computing, when multiple threads attempt to access a shared resource at the same time, a mutual exclusion (mutex) is often used to avoid race conditions. A mutex addresses the problem of resource sharing by ensuring that only one thread can have access to the shared resource in the critical section at a time. While a mutex allows threads to cooperate efficiently, it is also a source of issues such as deadlock and resource starvation. Deadlock refers to the situation where each process is blocked while waiting for a shared resource to be released by another process, including itself. On the other hand, resource starvation occurs when a process is constantly denied the resource it needs to complete its execution due to the competition for that resource.

One solution is to implement a lock-free (non-blocking) version of the shared object. A lock-free implementation guarantees that each thread gets to operate on the shared object in a finite number of steps without mutexes regardless of the threads' actions [7]. Prior lock-free algorithms are either array-based or list-based, but array-based algorithms are often considered impractical [7]. A few lock-free list-based algorithms have been discussed in the past, but the first to be considered correct is one that uses the compare-and-swap (CAS) technique [10]. CAS is an atomic instruction that compares the current value of a variable to the expected value of the variable. If the current value matches the expected value, then CAS swaps the current value of the variable for the new value passed in. An atomic instruction means there is no intermediate result, so multiple instructions can happen concurrently without corrupting each other's results. Thus, a CAS-based list-based lock-free algorithm often improves synchronization and performance robustness. However, CAS-based algorithms are typically prone to the ABA problem [7]. The problem arises from the C of the CAS (compare) when a thread is tricked into thinking that a value has never been changed even though another thread has been modifying the value back and forth. One common workaround is to introduce a tag to the quantity being considered, which we will discuss later in this paper.

III. Concurrent hashing with chaining

A. Reader-writer lock

For the lock-based hash table implementation, we used `std::shared_mutex` class as the read-writer lock to enhance the parallelism of the hash table. While a mutex is often used to support mutually exclusive access on shared data, a shared mutex can facilitate both exclusive access and shared access [11]. If a thread is holding an exclusive lock, all other threads that try to acquire the exclusive and shared lock will be blocked. However, if at least one thread is owning the shared lock, other threads can acquire that shared lock, but no threads can get the exclusive lock.

B. Coarse-grained hash tables

The coarse-grained hash table uses one global reader-writer lock to maintain the internal consistency of the hash table. A thread must acquire the shared lock to read a key-value pair (through `GET` method) from the hash table. Threads that come later can still read key-value pairs from the hash table by acquiring the shared lock. However, no thread can `INSERT` or `DELETE` any keys from the hash table. A thread that wants to write to the hash table has to request the exclusive lock. Once a thread holds the exclusive lock, no thread can perform read/write on the hash table. As the hash table becomes dense, i.e. the number of elements per bucket exceeds a certain

threshold, the hash table will acquire the exclusive lock to grow the number of buckets. This could cause lots of lock contention inside the hash table since all threads have to stall until the hash table completes resizing. This approach is inefficient to be practical in large-scale systems that require a high degree of concurrency and high performance. Therefore, we want to use the reader-writer lock in a more fine-grained manner.

C. Fine-grained hash tables

Using a single global reader-writer lock to protect data consistency inside the hash table significantly decreases its performance. This performance degradation motivates us to use the lock at a finer granularity. Given each key, we can read/write to its corresponding bucket without interfering with other buckets in the hash table. Therefore, we can let each bucket have its own reader-writer lock and only use the global reader-writer lock for growing the hash table. Although this approach increases the lock overhead (i.e., more memory used by the hash table), it considerably reduces the lock contention.

A `GET` method takes a global shared lock and its corresponding bucket shared lock to read the value of a key. Thus, other threads that want to read on the same or different bucket will not be blocked.

`INSERT` and `DELETE` methods require an exclusive lock on their corresponding bucket since they modify the state of the internal linked list. However, these methods only need to take a global read lock since they do not affect read/write operations on other buckets. The `INSERT` and `DELETE` methods may change the size (number of key-value pairs) of the hash table if insertion and deletion are successful. Therefore, they need a global write lock to modify the size of the hash table. To avoid taking an exclusive lock for a small operation, we use the `std::atomic` object [12] to hold the size member variable of the hash table, thus maintaining the consistency of that shared variable without locking.

However, if we want to grow the hash table (increase its number of buckets), we have to acquire a global exclusive lock since we have to rehash every key into a new bucket after modifying the size of the hash table. This process affects read/write operations on every bucket, thus the need for a global exclusive lock.

Although keeping reader-writer lock at a bucket level could enhance the performance of `GET`, `INSERT`, and `DELETE` operations, growing the hash table is still a major source of bottleneck inside our hash table. The next section presents the lock-free concurrent hash table, which removes the use of lock from the hash table.

D. Lock-free hash tables

Michael presented a lock-free algorithm for the hash table that uses hashing with chaining to resolve collisions [7]. We implemented our lock-free hash table in C++ based on his idea to investigate its performance. This paper only gives a high-level explanation of how `SEARCH`, `INSERT`, `DELETE` work, not as detailed as they were described by Michael [7].

Each node in our implementation contains `Key`, `Value`, `Mark`, `Next`, and `Tag` fields. The `Key` and `Value` store the actual data of the hash table. The `Mark` field denotes if a node has been deleted. The `Next` field points to the next node in the linked list or is a null pointer. The `Tag` field is a mechanism to prevent the ABA problem [13]. According to Michael, the `Mark`, `Next`, and `Tag` fields must be placed in a contiguous aligned memory block on which we can perform atomic operations using CAS. We achieved this in our implementation by using `__int128` type, which is wide enough to hold 128 bits [14]. We placed the `Next` pointer in 64 low order bits and the `Mark` bit in the least significant bit of these bits since memory addresses are word aligned in most architecture. The remaining 64 bits can be used to hold the `Tag` field. By doing this, we can atomically read from or write to the `Mark`, `Next`, and `Tag` fields using `__sync_bool_compare_and_swap` [15].

The `GET`, `INSERT`, and `DELETE` methods keep private pointers `prev`, `cur`, and `next` when traversing the linked list, as shown Fig. 1. The `GET` function starts by reading the head of the linked list until the `Next` pointer is null. During execution, whenever there is a change in `*prev`, the search restarts from the head of the linked list since there are threads that modify the linked list. In addition, if we find a node marked as deleted but has not been freed yet, as indicated by the `Mark` field, we use CAS to swap the `Next` pointer of the previous node to point to the next node and then free the deleted node [7].

The `INSERT` method first finds a correct place to insert the node. After the search completes, it gives a correct `prev` and `cur` pointer. Then, we change the `Next` field of the new node to point to `cur`. After that, we use a CAS operation to verify if `prev` is still pointing to `cur`. If successful, we know that the linked list has not been modified by any other threads. Therefore, we change the `Next` field of `prev` to point to the new node, as illustrated in Fig. 2. Otherwise, the `INSERT` restarts [7].

The `DELETE` method uses the search routine to get the `cur` pointing to the node we want to delete and the corresponding `prev` and `next` pointers. Then, the `DELETE` operation uses two CAS to delete a node. The first CAS is used to mark `cur` as deleted. If successful, we use another CAS to change the `Next` field of `prev` to point to the `next` node, as shown in Fig. 3. If one of these CAS operations fails, we have to restart the `DELETE` procedure [7].

Michael did not explain how to grow the hash table when it becomes dense. Therefore, in our implementation, the number of buckets for the lock-free hash table is static and determined based on the expected number of keys inserted into the hash table.

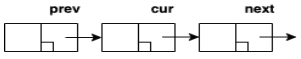

Fig. 1. Traversing the linked list with three pointers.

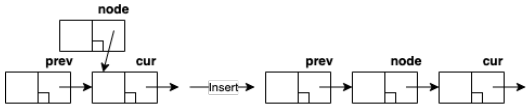

Fig. 2. Inserting a new node into the list.

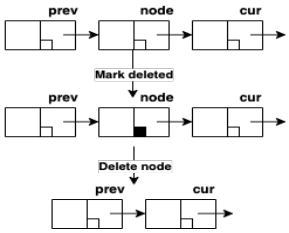

Fig. 3. Deleting a node from the list.

Evaluation

We used a 16-processor (8 cores with 2 threads per core) 11th Gen Intel(R) Core i9-11900 (2.5 GHz) to evaluate the performance of concurrent lock-based and lock-free hash tables.

In the following experiments, for the lock-based hash tables, we set their load factor to 5, i.e., on average, there will be 5 elements per bucket. We want to use this load factor to minimize the number of times a hash table resizes, which is a major bottleneck in the lock-based hash tables. For the lock-free implementation, since we cannot resize the hash table as it gets dense, we preallocate $100,000$ buckets for the hash table. This is reasonable since we expect our hash table to store millions of key-value pairs. Thus, each bucket will have around 10 elements.

We benchmarked the runtime of each implementation when handling one million access with various numbers of threads used. Fig. 4 illustrates the run time of coarse-grained, fine-grained, and lock-free hash tables performing 800 thousand `GET`, 100 thousand `INSERT`, and 100 thousand `DELETE`. The performance of the coarse-grained hash table decreased as we increased the number of threads used due to the overhead of locking. For the fine-grained hash table, we achieved a better runtime when using 4 or 6 threads, but as we increased the number of threads, we got diminishing returns. The lock-free hash table outperforms the two lock-based implementations. It takes under 50ms to complete one million access, and its latency becomes lower if we increase the number of threads.

In the next experiment, we evaluate the speedup when using the three concurrent hash tables versus `std::unordered_map` to complete one million access. Fig. 5 demonstrates that the speedup of coarse-grained parallel hash tables is less than 1. Therefore, it is better to use the sequential `std::unordered_map` to perform the same workload. For the fine-grained hash table, the speedup we got is around 2.5, which is worth considering using if the complexity of lock-free data structures is a problem. Lastly, we got more than ten times speedup when using the lock-free hash table for processing 80% `GET`, 10% `INSERT`, 10% `DELETE`. Using the lock-free hash table with 8 or 12 threads could achieve the potential speedup of up to 30. Therefore, this lock-free parallel hash table is the data structure of choice for high-performance systems.

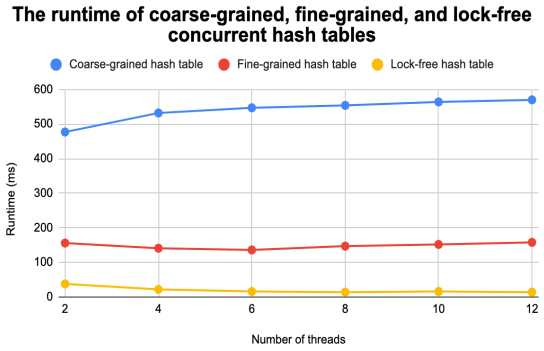

Fig. 4. $1,000,000$ access, 80% `GET`, 10% `INSERT`, 10% `DELETE`

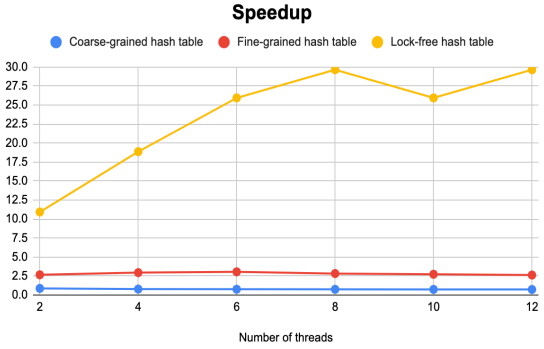

Fig. 5. $1,000,000$ access, 80% get, 10% insert, 10% delete

Conclusion

In this paper, we described the working of coarse-grained, fine-grained, and lock-free concurrent hash tables and evaluated their performance. Our experimental results show that due to extensive locking, coarse-grained hash tables perform worse as we increase the number of threads. The sequential `std::unordered_map` even achieves a better speedup than a coarse-grained hash table on the same workload. On the other hand, fine-grained hash tables have a better running time since the lock is used at a finer granularity. Nevertheless, lock-free hash tables achieve both the best running time and the highest speedup when tested on a varying number of threads. In future work, we plan to explore lock-free extensible hash tables, which grow dynamically without allocating new buckets as they get dense.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[2] G. Zhang and D. Sanchez, "Leveraging caches to accelerate hash tables and memoization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 440–452. [Online]. Available: https://doi.org/10.1145/3352460.3358272
[3] M. Desai, R. Mehta, and D. Rana, "A survey on techniques for indexing and hashing in big data," 07 2019.
[4] T. Zink and M. Waldvogel, "Efficient hash tables for network applications," *SpringerPlus*, vol. 4, no. 1, p. 222, 2015. [Online]. Available: https://doi.org/10.1186/s40064-015-0958-y
[5] T. D. Wu, "Bitpacking techniques for indexing genomes: I. hash tables," *Algorithms for Molecular Biology*, vol. 11, no. 1, p. 5, 2016. [Online]. Available: https://doi.org/10.1186/s13015-016-0069-5
[6] T. Maier, P. Sanders, and R. Dementiev, "Concurrent hash tables: Fast and general?(!)," *CoRR*, vol. abs/1601.04017, 2016. [Online]. Available: http://arxiv.org/abs/1601.04017
[7] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 73–82. [Online]. Available: https://doi.org/10.1145/564870.564881
[8] J. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022000079900448
[9] Z. J. Czech, G. Havas, and B. S. Majewski, "An optimal algorithm for generating minimal perfect hash functions," *Information Processing Letters*, vol. 43, no. 5, pp. 257–264, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/002001909290220P
[10] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 300–314.
[11] cppreference.com, "C++, Concurrency support library, std::shared_mutex," https://en.cppreference.com/w/cpp/thread/shared_mutex.
[12] ——, "C++, Concurrency support library, std::atomic," https://en.cppreference.com/w/cpp/atomic/atomic.
[13] A. Padegs, "IBM System/370 Extended Architecture, Principle of Operations," *IBM Journal of Research and Development*, 1983.
[14] "GCC Manual, Extensions to the C Language Family, 128-bit integers," https://gcc.gnu.org/onlinedocs/gcc/_005f_005fint128.html.
[15] "GCC Manual, Extensions to the C Language Family, Built-in functions for atomic memory access," https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/Atomic-Builtins.html.