Concurrent Hash Tables: Fast *and* General?(!)

Tobias Maier Peter Sanders
Karlsruhe Institute of Technology
t.maier@kit.edu sanders@kit.edu

Roman Dementiev
Intel Deutschland GmbH
roman.dementiev@intel.com

Abstract

Concurrent hash tables are one of the most important concurrent data structures with numerous applications. Since hash table accesses can dominate the execution time of the overall application, we need implementations that achieve good speedup. Unfortunately, currently available concurrent hashing libraries turn out to be far away from this requirement in particular when contention on some elements occurs.

Our starting point for better performing data structures is a fast and simple lock-free concurrent hash table based on linear probing that is limited to word-sized key-value types and does not support dynamic size adaptation. We explain how to lift these limitations in a provably scalable way and demonstrate that dynamic growing has a performance overhead comparable to the same generalization in sequential hash tables.

We perform extensive experiments comparing the performance of our implementations with six of the most widely used concurrent hash tables. Ours are considerably faster than the best algorithms with similar restrictions and an order of magnitude faster than the best more general tables. In some extreme cases, the difference even approaches four orders of magnitude.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; E.1 [*Data Structures*]: Tables; E.2 [*Data Storage Representation*]: Hash-table representations

Keywords Concurrency, Lock-Freedom, experimental

1. Introduction

A hash table is a dynamic data structure which stores a set of elements that are accessible by their key. It supports insertion, deletion, find and update in expected constant time. In a concurrent hash table, multiple threads have access to the same table. This allows threads to share information in a flexible and efficient way. Therefore, concurrent hash tables are one of the most important concurrent data structures.

Hash tables are ubiquitous in sequential algorithms. Many of these algorithms have in common that hash table accesses constitute a significant fraction of the running time. Therefore, offering highly scalable concurrent hash tables is essential for the parallelization of these methods. In our tests, we show that modern library implementations do not deliver significant speedups over sequential ones, especially if there is contention on some elements, or the final size of the table is unknown. For more discussion of related work refer to the full paper (1).

It seems to be folklore that a lock-free linear probing hash table where keys and values are machine-words, which is preallocated to a bounded size, and which supports no true deletion operation can be implemented very efficiently using atomic compare-and-swap (CAS) instructions. Our goal is to improve the usability of this hash table by lifting its restrictions (1). The main instrument that we used for our improvements is a migration algorithm which we optimized to reduce the contention and communication between threads.

2. Implementation

Our imlementation makes use of the atomic 16*Byte* CAS operation present on modern CPUs to store key-value pairs in one operation. Interestingly, we found some performance benefits when using Intel TSX transactions over atomic operations (measurements in (1)).

Improved Update Interface: In addition to our optimized growing implementation, we also propose a new interface for the `update` operation which improves the usability in concurrent scenarios. Typically hash tables either return a mutable reference or they have an `update` function which can be used to replace the element stored at a key. The first option does not enforce changes to be atomic leading to errors made by inexperienced users. Using the second option, it is very hard to implement changes that depend on the current value in a thread-safe way.

Our proposed interface takes an element e and a function f as parameters. The function has to take two elements as parameters. The new stored element will be the result of $f(e, current)$. The atomicity of the update function can easily be enforced with atomic CAS operations.

Growing the Table: In any scenario there are two possibilities to grow a data structure: elements can be migrated into a bigger structure, or additional space can be allocated for new elements without freeing the current space. Due to the running time cost of checking multiple tables during a find operation, we migrate the full table. In a sequential application, this migration can easily be amortized with the number of elements necessary to cause the migration. To attempt a similar amortization in a parallel scenario, it is important that $O(|threads|)$ threads cooperate on the migration.

Since the growing must be hidden from the application, finding threads to grow the table can be a challenge. To find such threads, we implemented two different options: the first, more practical approach is to force each thread accessing the table to help growing the table; the second approach uses a thread pool of growing threads that are only activated once the table begins to grow (all shown measurements are using the first approach).

During the growing process, it is important that elements which have already been copied cannot be changed in the old table. This could lead to lost updates or other inconsistencies. Again, our hash table offers two different approaches to solve this problem. The first and easier option is to mark each element when it is copied (named `m_grow`). The second option ensures that no hash table operations

PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851188

can run concurrently to the migration. To do this we use flags which are set to signal that the hash table is in use (named `fl_grow`). The contention on these flags is minimal since they are exclusive to each thread (they only need to be checked at the beginning of the growing process).

The migration itself is optimized to reduce the number of atomic operations. When we use the same hash function in the target table, a cluster of elements in the source table (index a to b) will be migrated into a concurrent block of the target table (index $2 \cdot a$ to $2 \cdot b$). If one thread migrates all elements within one cluster, insertions do not need to be atomic because no other element will be written into this block.

Interestingly, even triggering the growing procedure becomes harder in concurrent scenarios. Keeping an exact count of stored elements can lead to contention on the global counter. Therefore, we only keep an approximate count of all elements. Each thread counts the number of its insertions and updates the global count every $\mathrm{O}(|\mathit{threads}|)$ insertions.

3. Experiments

All tests were performed on a two-socket machine with Intel Xeon E5-2670 v3 processors (codenamed Haswell-EP). Each processor has 12 cores runnng at $2.3\mathit{GHz}$ base frequency ($128\mathit{GB}$ RAM). The presented numbers are averaged between five executions.

Test Instances: In this short paper, we want to highlight three common access patterns from the extended test suite we used in (1).

Uncontended Insert: We fill the hash table with 100 000 000 random elements (keyspace sampled uniformly). We use a varying number of threads (strong scaling) to show the superior scalability of our implementation. To show the growing capabilities of the tested tables, they are initialized with an insufficient initial size (our table was initialized to its minimal size of 4096 cells, the competitors' tables to half their target size).

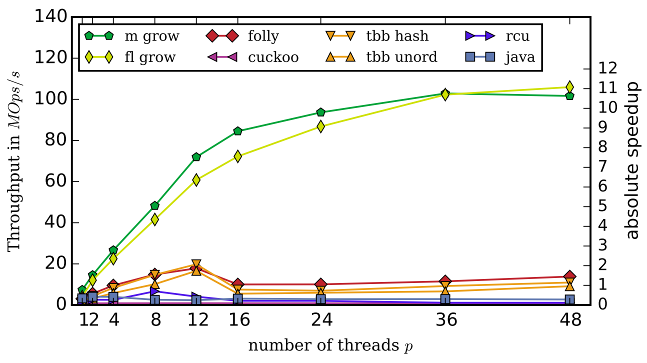
Contended Aggregation: This test uses insert or increment to count the number of elements with any specific key. 100 000 000 keys are pregenerated with a Zipf distribution with varying contention ($p(k) = \frac{1}{k^s \cdot H_{N,s}}$ contention regulated with s). Contrary to the uncontended insert, we initialize each table with the maximal needed capacity (except for the marked table).

Contended Find: This test uses a similarly generated key sequence, to test the find performance when some elements are searched repeatedly. All searched elements are contained. Additionally, there are 100 000 000 elements such that the table size is independent from the contention.

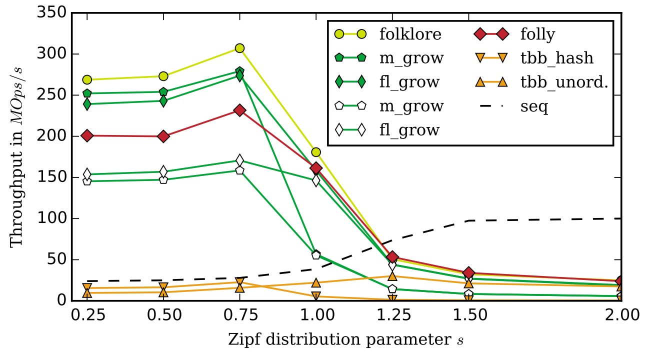
Analysis: In Figure 1a we see that our growing implementation is significantly more efficient than any other implementation capable of growing. More interestingly, no other tested hash table achieves a significant speedup over the sequential table (see right side scale), even though the sequential table grows from its initial size of 4096 comparable to our implementation. Additionally, none of the competitors' implementations scale well beyond one socket. This problem even remains for some of the implementations when growing is not necessary (especially TBB).

In Figure 1b we can see that the performance gets drastically reduced when an element is frequently changed (under contention). Interestingly, contention can also have a positive effect, but once the contention increases (at $s = 1.0$; $p(1) \approx 5\%$) all concurrent tables begin to deteriorate. From $s = 1.25$ forward sequentially aggregating elements is faster than the parallel aggregation.

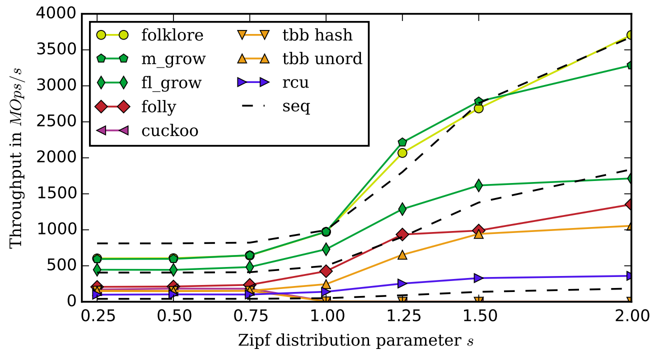
We can also see that without growing, folly can achieve comparable running times to our solution. This is not surprising, since their basic approach is very similar to ours (linear probing without true deletion).



(a) Uncontended insert with growing (speedup relative to growing sequential table)



(b) Contended aggregation (white marked tables are growing; 48 threads)



(c) Contended find (successful; $(1/10/20) \cdot \mathit{sequential}$; $p = 48$)

Figure 1: Performace on three different operation patterns.

Figure 1c shows, that contention can have the opposite effect on find operations. Find operations can improve with contention, because the visited cells are likely already stored in the cache.

4. Conclusion

By adapting a simple linear probing hash table we constructed a concurrent, growing hash table which is very efficient. In some tests, the resulting table is the only hash table that achieves significant speedups over a sequential hash table.

References

[1] Tobias Maier, Peter Sanders and Roman Dementiev, *Concurrent Hash Tables: Fast and General?(!)*, arXiv:1601.04017, 2016.