

Comp 522 Final Project

Author¹

¹Aaron Thomas Cherian

atc8@rice.edu

***Abstract.** Motivation, what was decided*

1. Introduction

Concurrent data-structures are a necessity in a parallel application. Almost all of the data-structures that we are familiar with; arrays, binary trees, linked-lists, hash tables, queues, skip-lists have been tweaked concurrent usage. Based on the performance needs of one's application, the choice of the data-structure can be made.

The motivation of my work is to identify a concurrent data-structure to replace the lock-based hash table implementation inside HPCToolkit. Ideally, the replacement should be lock-free and obviously, outperforming the current solution.

Factors for choosing a concurrent data-structure are performance, synchronization techniques, ease of implementation, programming language. Gramoli¹ does a comprehensive analysis of 31 such concurrent data-structures and compares their performance. I have taken help of this work to select the data-structure for our use case inside HPC-Toolkit. The performance results show that hash tables support the highest operations/second. This is intuitive, since hash tables are the only data structure that give near constant time complexity.

The hash table choices presented in the paper are `j.u.c.ConcurrentHashMap`(Java), Micheal's hash table(C/C++), Cliff Click's hash map(Java), Contention-friendly hash table(Java), Resizable hash table(Java) and Elastic hash table(C/C++). Since HPCToolkit uses C/C++, I compared Micheal's hash table and Elastic hash table. Java algorithms can be considered, but after going through the efforts of porting the code to C++, there is no guarantee of getting a similar performance. Micheal's hash table outperforms Elastic hash table by a big margin. Another advantage of Micheal's algorithm is that it uses the Read-Modify-Write or Compare and Swap(CAS) synchronization technique. CAS techniques are the fastest for multi-core architectures. Thus we will explore Micheal's implementation and see how itself and its enhancements fare.

Roadmap. In Section 2, we will look at Micheal's hash table implementation and its enhancements. Section 3 and 4 will explore the implementation details and performance comparison. Section 4 describes the issues we faced during integration with HPCToolkit.

2. Background

We will look at the implementation complexity of the hash table algorithms and the memory reclamation techniques for Micheal's hash table and its variants.

2.1. Algorithms

We have selected a few algorithms for review because they gave 3 attractive properties: lock-free, synchronization operation and time complexity of shared data structure.

Advantages of lock-free algorithms: no dead-locks, no delays due to slow threads, equal or better performance than lock-based implementations.

Advantages of CAS: Best of the existing synchronization techniques

Advantages of hash table: Constant time operations

First we look at the original implementation of Micheal's hash table, then at 2 variants: the implementation used by Gramoli¹ and an extensible hash table implementation.

2.1.1. Micheal's Hash table

Micheal's hash table² implements the buckets in a linked list fashion. It provides lock-free synchronization and has other desirable properties such as space-efficiency, dynamic and linearizable. For memory management, there are multiple feasible techniques (IBM freelist, Safe Memory Reclamation (SMR), etc). Since the algorithm mentioned in the paper contains the logic for SMR, we stick with that approach.

Questions

1. What is the hash function used?

2.1.2. Micheal's Hash-table with Harris Linked-List

This is the implementation that Gramoli uses in his Synchrobench paper.¹ It reuses the lock-free linked list by Harris.³ The reason for the difference in implementation in Gramoli's work is because Micheal published the source code long after the original work was published. When Gramoli started with his work, he did not have the source code that we refer to in section 2.1

A key difference, as mentioned by Gramoli, is that Harris did not specify a memory reclamation algorithm while Micheal discussed a few. Harris's algorithm allows a thread traversing the list to access the contents of a node or a sequence of nodes after they have been removed from the list. If removed nodes are allowed to be reused immediately, the traversing thread may reach an incorrect result or corrupt the list. This precludes Harris implementation from using simple and efficient thread management techniques like SMR and freelist. Instead, ThreadScan⁴ has been used to automatically add memory reclamation to Harris linked list.

Questions

1. What is the difference between a normal LL and Harris LL?

2.1.3. Split-Ordered Lists: Lock-Free Extensible Hash Tables

Split-ordered lists⁵ tackles the problem of increasing the bucket count in hash tables without using locks or sub-optimal methods. Dynamically changing the bucket size i.e Extensible hash table is done using the concept of recursive split-ordered lists. The basic idea of the algorithm is this: rather than transferring objects between hash-buckets, data is stored in a single linked list and bucket pointers are updated during a resize operation. The advantages of this algorithms are: lock-free, easy to implement, good performance in both multi-core and non multi-core environments and the flexibility of efficiently extensible hash table.

This algorithm extends over the ordered list implementation of Micheal's² algorithm.

Questions

1. Why is this not compared with Micheal's lock-free algorithm?

2.2. Memory Management Methods

Memory management is complicated for Lock-free algorithms. In lock-based algorithms, holding a lock on an object ensures that delete operation by a thread can be performed without any worrying that some other thread may be accessing that variable.

2.2.1. Safe memory reclamation

Safe Memory Reclamation⁶ is a memory management technique developed by Micheal. Internally, it uses hazard-pointers methodology. It takes constant amortized time for each delete operation. Other advantages of hazard-pointers are: wait-free, does not need shared memory, lock-free solution for ABA problem.

The core idea of SMR is to associate a small number of single-writer multi-reader shared pointers, called hazard pointers, with each thread. The hazard pointers keep tabs of nodes inside the shared data structure. A hazard pointer either has a null value or points to a node that may be accessed later by that thread (without further validation that the node is still valid). Each hazard pointer can be written only by its owner thread, but can be read by other threads.

Whenever a thread passes a node for deletion, it keeps the node in a private list. When this list reaches a threshold of R items, the thread scans the hazard pointers of other threads for matches with its private list. If a node(in the private list) is not matched by any of the hazard pointers, then it is safe for this node to be reclaimed. Otherwise, the thread keeps the node until its next scan of the hazard pointers. This prevents the freeing of any node marked for deletion if it is continuously pointed to by one or more hazard pointers of one or more threads from a point prior to its removal.

2.2.2. ThreadScan

ThreadScan is a garbage collection tool for C/C++. If the memory node to be reclaimed cannot be reached by any application thread, then the memory reclaiming thread will use

OS signalling techniques to find if the to-be-deleted block is safe to be deleted i.e. no other thread is accessing it. ThreadScan provides $2\times$ performance improvement over Hazard Pointers, which is used internally inside SMR.

A shared buffer of delete blocks is maintained internally by ThreadScan. When a memory block is marked for deletion, it should be passed to the ThreadScan free function. The free function will add this memory block to the delete buffer. When the delete buffer becomes full, the last thread which called free becomes the memory reclaiming thread. This thread will send signals to all threads accessing the data-structure and check within its stacks and registers to see if the memory blocks are being accessed by them. If yes, then the block is marked. Once all application threads have responded to the signal, the reclaiming thread safely frees all unmarked blocks.

Modern operating systems give higher priority to signals over threads and have measures to avoid thread starvation. Thus ThreadScan is lightweight and avoids the memory reclaiming thread to be blocking. Another advantage is, since its a garbage collector, the only thing the programmer needs to do is call ThreadScan API for memory reclamation. The memory management code gets simplified.

Questions

1. ThreadScan assumption is that pointer masking techniques wont be used. Does this assumption hold inside HPCToolkit?
2. The memory node needs to be unlinked(if the hashbuckets are implemented using linkedlists). Will this pose any difficulties?
3. what happens on a free call when collect operation is in progress?

3. Implementation

ThreadScan seems to be recent and a promising garbage collector tool. We need to try all 6 combinations of the 3 algorithms with the 2 memory management techniques.

To-do

4. Performance Comparison

To-do

ThreadScan claims to be better than SMR, is this true? Does memory reclamation dent the overall performance ?

Hazard pointers scale well in the lock-free hash table because bucket traversals are short and so there are few memory barriers per operation.

5. Data Race Detection

To-do

6. Integrating with HPCToolkit

To-do

7. Conclusion

To-do

References

- [1] V. Gramoli, “More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms,” in *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP 2015, (New York, NY, USA), p. 1–10, 2015.
- [2] 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*, SPAA ’02, (New York, NY, USA), p. 73–82, 2002.
- [3] T. L. Harris, “A pragmatic implementation of non-blocking linked-lists,” in *Proceedings of the 15th International Conference on Distributed Computing*, DISC ’01, (Berlin, Heidelberg), p. 300–314, Springer-Verlag, 2001.
- [4] D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit, “Threadscan: Automatic and scalable memory reclamation,” *ACM Trans. Parallel Comput.*, vol. 4, May 2018.
- [5] O. Shalev and N. Shavit, “Split-ordered lists: Lock-free extensible hash tables,” *J. ACM*, vol. 53, p. 379–405, May 2006.
- [6] M. M. Michael, “Hazard pointers: safe memory reclamation for lock-free objects,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.