Comp 522 Final Project

Author[1]

[1]Aaron 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, highly performant.

Factors for choosing a concurrent data-structure are performance, synchronization techniques, ease of implementation, programming language. Gramoli[1] 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 HPCToolkit. 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.

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 and the memory reclamation techniques for Micheal's hash table and its variants. First we look at the original implementation, then at 2 variants: the implementation used by Gramoli[1] and an extensible hash table implementation.

2.1. Micheal's Hash table

Micheal's hash table[2] 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?

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

This is the implementation that[1] uses in his paper for performance comparison. 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, ThreadScan[4] 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?

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

Split-ordered lists[5] 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's[2] algorithm.

Questions

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

3. Implementation

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. Integrating with HPCToolkit

To-do

6. 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 *Distributed Computing* (J. Welch, ed.), (Berlin, Heidelberg), pp. 300–314, 2001.

[4] D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit, "Threadscan: Automatic and scalable memory reclamation," 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.