NUMASK Integration to SetBench *

Ruifan Wang and Anlan Yu

12/2020

1 Project Overview

In this project, we are trying to integrate NUMASK [1] from SynchroBench [2] to SetBench [3].

2 Background

2.1 NUMASK

NUMASK is a high performance scalable skip list for NUMA. It prevents pointer chasing between different NUMA zones during skip list operations by defining intermediate layer as a per NUMA zone local copy of data layer.

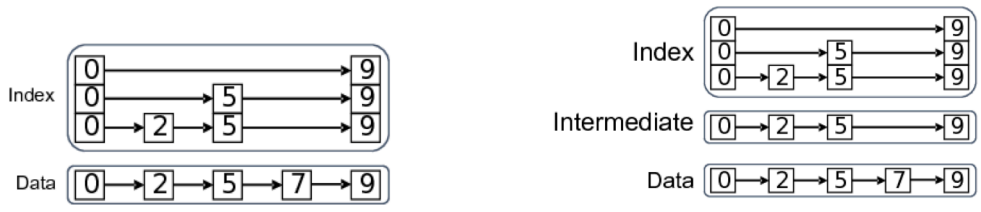
As shown in Fig. 1 (a), traditional skip list has an index layer per NUMA zone and a data layer in total where nodes are scattered on different NUMA zones. For each skip list operation, searches are going through the index layer on the query NUMA zone and changes are made on the data layer as well as index layers. In this case, the modifications made on data layer, where pointer is chased among different NUMA zones, are finished on the worker thread and the critical path delay is quite high as a result. NUMASK is proposed to solve this problem. By introducing intermediate layer as a local copy of data layer, pointer chasing among different

*The implementation is available at https://github.com/anlanyu66/NUMASK_setbench.

[1] http://www.cse.lehigh.edu/ palmieri/files/pubs/CR-disc2018.pdf

[2] https://github.com/gramoli/synchrobench

[3] https://gitlab.com/trbot86/setbench/-/wikis/home



(a) Skip list. **(b)** NUMASK.

Figure 1: Separation of layers for traditional skip list and NUMASK

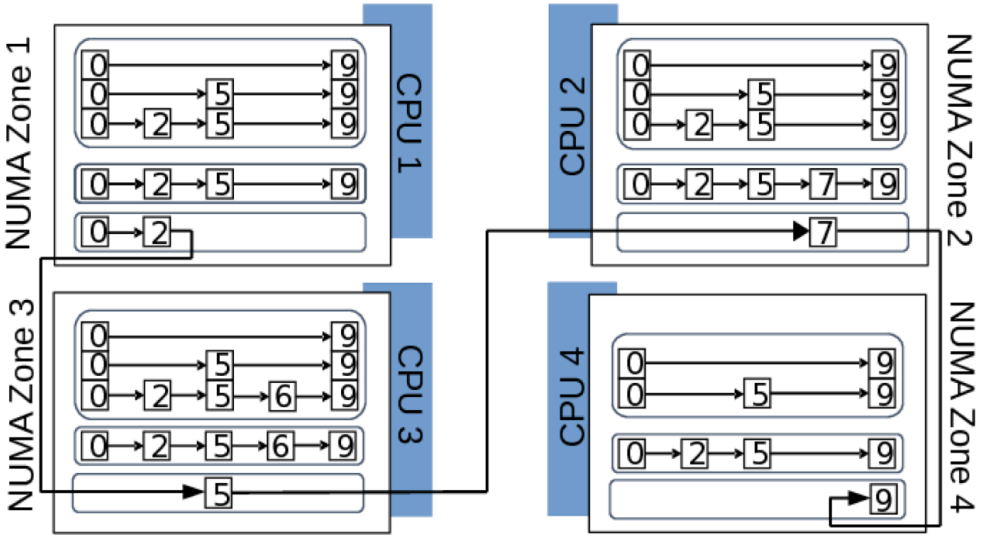


Figure 2: NUMASK on 4 NUMA zones

NUMA zones is eliminated. Also, modifications on the data layer are assigned to helper threads so that the critical path delay is greatly reduced compared with traditional skip list version.

Fig. 2 shows an example of NUMASK on 4 NUMA zones. Each NUMA zone holds its own index layer and intermediate layer. Note that intermediate layers are not necessarily the same for different NUMA zones because the modifications made by helper threads are not synchronous.

2.2 SetBench

SetBench is a powerful tool for data structure experiments in c++. SetBench supports for gathering statistics concerning the data structure and benchmark, both during and after an execution. SetBench uses record manager file for memory allocation and reclamation. There's different methods of memory allocation and reclamation user can choose from.

2.3 SynchroBench

NUMASK is currently integrated in SynchroBench. Synchrobench is a micro-benchmark suite used to evaluate synchronization techniques on data structures. Synchrobench is written in C/C++ and Java and currently includes arrays, binary trees, hash tables, linked lists, queues and skip lists that are synchronized with copy-on-write, locks, read-modify-write, read-copy-update and transactional memory. A non-synchronized version of these data structures is proposed in each language as a baseline to measure the performance gain on multi-(/many-)core machines.

3 Motivation

NUMASK is implemented in SynchroBench [4] previously. However, no memory reclamation techniques were implemented and deletion operation never actually frees the memory of the deleted node. This memory leakage can be handled with memory reclamation techniques. As mentioned in Section 2.2, SetBench has flexible memory allocation and memory reclamation choices for users to choose from. So the memory leak problem will be well addressed using SetBench.

4 Major Challenges

4.1 Lack of Documentation

The major challenge is that the documentation of Setbench is not complete. The only instructions SetBench provided was the compilation and setup steps. In their to-do list, they have had "How to add a data structure to the synthetic benchmark" for years and still has not been added to their wiki. Therefore, it was both necessary and difficult for us to understand the structure and the functionalities of the entire benchmark by reading other data structure's code. Since every data structure has a unique implementation, the task of transforming NUMASK based on other data structures was not easy. The code for data structure was readable since they are not extremely long. However, if we want to understand how SetBench work fundamentally such as the "record manager" or the different memory reclamation techniques, it was almost impossible to read these files because they contain thousands of lines of code.

4.2 SynchroBench vs. SetBench

The difference between SynchroBench and SetBench raises the second challenge. From a very high level speaking, SynchroBench grants more freedom over the data structure than SetBench does, especially the testing aspect. A lot of functionalities that was in test.cpp [5] are already being taken care of by SetBench itself. In SetBench, each data structure is required to be created as one single object, which requires the main data structure to be in one class. From there, every data structure will have a adapter.h class that basically integrates the data structure with SetBench. In contrast, SynchroBench's NUMASK is part of the SkipList folder, so we could easily borrow test.cpp class from a different type of skip list data structure.

5 Implementation Details

5.1 Adaptor.h

Similar to the test.c class in SynchroBench, adaptor.h is what SetBench uses to integrate each data structure to SetBench as well as any memory reclamation techniques we want to implement to the data structure. Due to the similarity between all the

[4] https://github.com/gramoli/synchrobench
[5] https://github.com/gramoli/synchrobench/blob/master/c-cpp/src/skiplists/numask/test.cpp

adaptor.h class, based on the adaptor.h class for a skiplist data strucutre that was already implemented in SetBench, we created a very similar adaptor.h class for NUMASK. Due to the time constraint, there are two major aspects of adapter.h class that we need to take care of in the future. Firstly, since record manager takes in a parameter of Node<K, V>, we need to change the node that NUMASK is currently using so that it is templated. The second part is the goal of the entire project: adding a memory reclamation technique to record manager. We first decided to integrate NUMASK to SetBench before the start adding any memory reclamation technique, which is why this part has not been done yet.

5.2 NUMASK.h

NUMASK.h provides the implementation details for the data structure, where APIs for data structure operations are provided. Initialization of search layer, per-NUMA-helper threads and data-layer thread all happen in this file. Since the worker threads are evoked in the application file in SetBench, the helper threads dedicated for NUMASK is evoked in NUMASK.h. There are still more missing pieces to the puzzle. We still need to add a barrier in the constructor. Also, the transition from SynchroBench allocator to SetBench allocator still need to be finished. The initthread function needs to be implemented as now we made sure each thread is running on the correct NUMA zone within the operation functions. This was supposed to be done in the initthread function.