Sri Hari Malla - CS19BTECH11039
Sanjay Krishna Ambati - CS19BTECH11013
Prof : Dr. Sathya Peri
Parallel and Concurrent Programming
Indian Institute of Technology Hyderabad

Project - Report

Sri Hari Malla - CS19BTECH11039
Sanjay Krishna Ambati - CS19BTECH11013

June 13, 2022

Efficient Hash Tables

1 Abstract:

To cope up with the fast developing world, the computers need to speed up and provide better computation power than the old versions. This being said, most of the systems in the current day are multi-thread supported. But is it being used to the fullest? Absolutely not!. If we can make use of these multi-thread supporting architecture to at least a remarkable extent, then it would make a significant revolution in terms of computation power and speed. Leaving this aside, the Hash tables are one of the most used and popular data structure which is very well known for its speed in look ups, inserts and deletes. In this course project, we are trying to make hash tables parallel and concurrent so that most multi-threaded architecture makes most of it. We are going to implement the following in `C++`.

The main motivation for choosing this project is to fasten the fastest data structure, even a small addition to efficiently use the hash tables is a significant addition I think.

2 Algorithms:

We found an insightful paper where it is being stressed on the usage of `sub_map`. We've studied other papers, but none of them was as interesting as this and also they are not providing a significant increase in performance.

The skip list based implementation of chaining, coarse-grained and fine-grained synchronization implementation are well versed in the literature. So we are going with `sub_map` implementations of both the build hash tables. If time permits, we will also touch the implementations of the parallel hash tables in the literature and check performance amongst all the build hash tables.

3 Detailed Design of Sub_Map:

- The initial hash table implementation contains insert, delete and search operations which can be used as `insert(key,value)`, `delete_item(key)` and `search(key)`
- The whole point is to parallelize these operations without actually ruining the consistency and now we use this sub_map method to achieve this in this project.
- The idea of sub map is to maintain several hash maps at once and use them.
- The main class creates several threads, and each thread is assigned to a sub map.
- Each sub map is an actual hash map implemented before.
- We first take the key and pass it through an extremely simple hash function. The outcome of the simple hash function ranges from 0 to number of threads available so that, we have each thread assigned to a hash map.
- Creating a thread for each entry is not consistent and linearizable. The later thread might run earlier than the former thread, due to any context switch or thread's load balancing, thus we need to avoid this approach.
- Instead execute the commands one by one for particular hash. Thus we can achieve parallelism to some extent. But the main task is to execute the commands one by one.
- We tried to implement using locks, but for the test cases we built, it is worse than expected. So we need to develop an algorithm which doesn't use locks but can still achieve the task.
- We achieve it by using signalling mechanism which can be achieved using semaphores.
  - Start a thread and assign a hash map to each of the threads.
  - Maintain a queue to load and remove instructions.
  - Create and init a set of semaphores, each initially waits at the start of the function.
  - Load instructions from test cases, en queue in the queue, and post the semaphore as soon as the instruction is loaded.
- This ensures the computation power is not gone waste as `cpus` need not check for queue every time.
- Also as the former is a simpler hash function, it takes relatively less time than the original hashing function. It affects the performance of the data structure in the long run or heavy loaded case.

Sri Hari Malla - CS19BTECH11039
Sanjay Krishna Ambati - CS19BTECH11013
Prof : Dr. Sathya Peri
Parallel and Concurrent Programming
Indian Institute of Technology Hyderabad

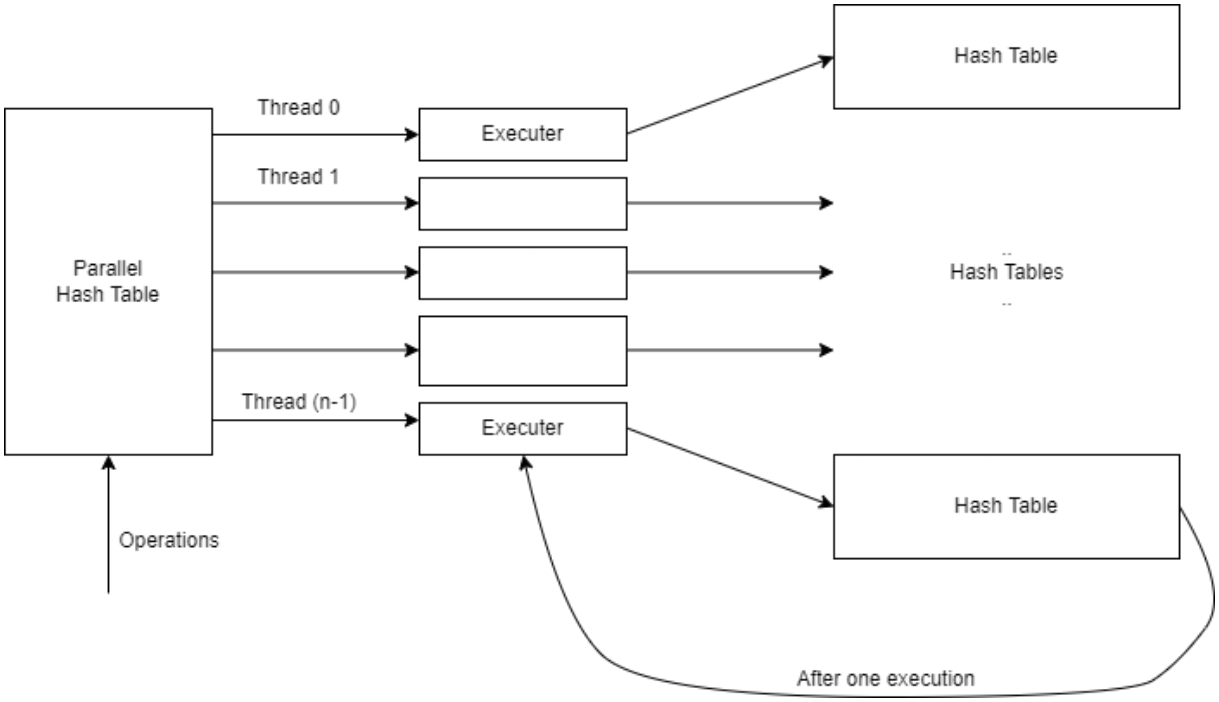


Figure 1: Flow structure

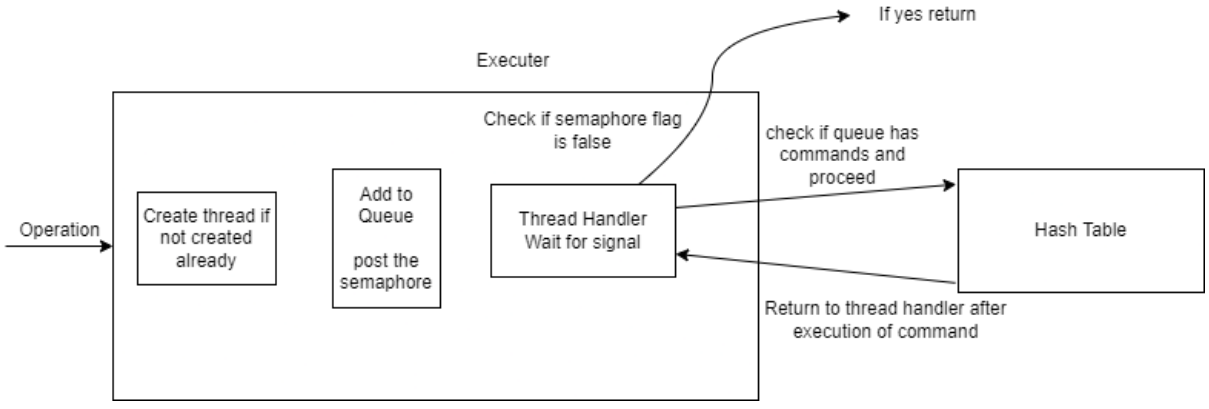


Figure 2: Single thread structure

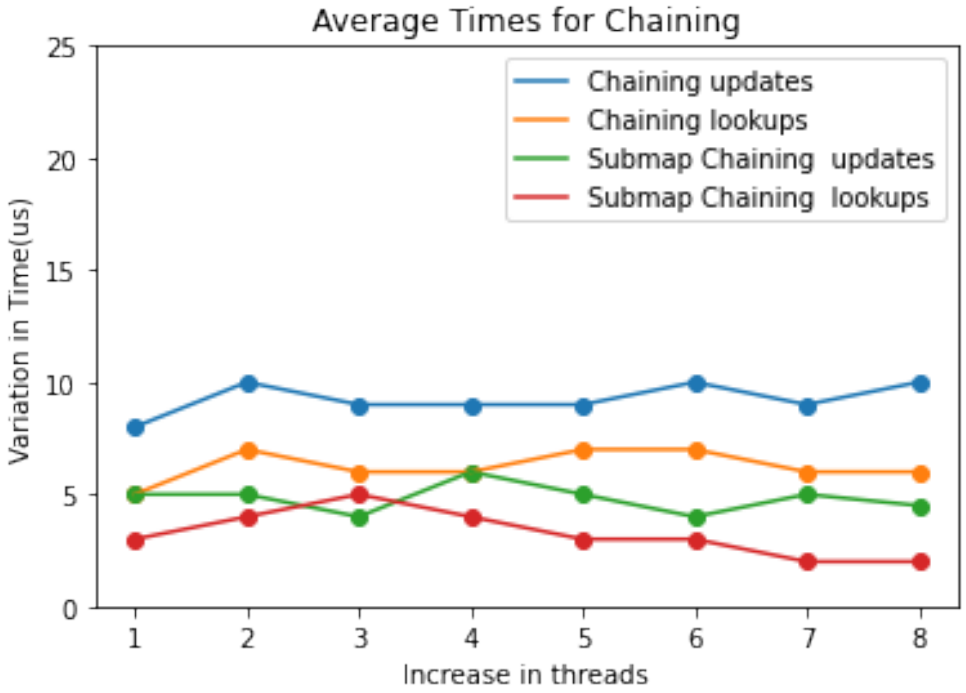
Sri Hari Malla - CS19BTECH11039
Sanjay Krishna Ambati - CS19BTECH11013
Prof : Dr. Sathya Peri
Parallel and Concurrent Programming
Indian Institute of Technology Hyderabad

Lazy List implementation:

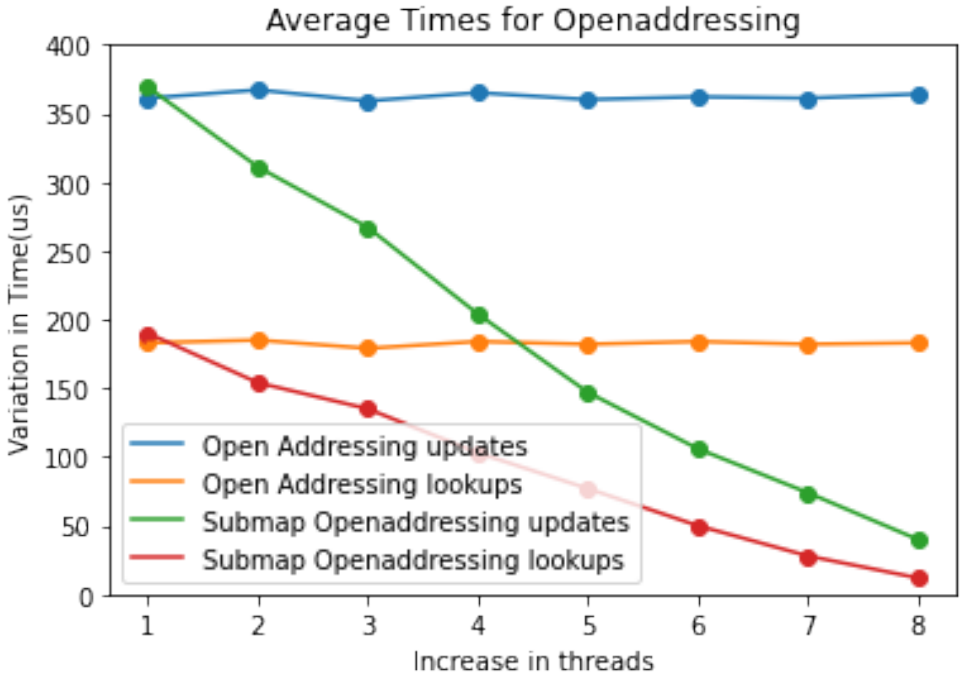
We also started implementing the wait-free list present in the textbook into the hash tables. But not sure if we can give this at the point of deliverable time.

4 Observations:

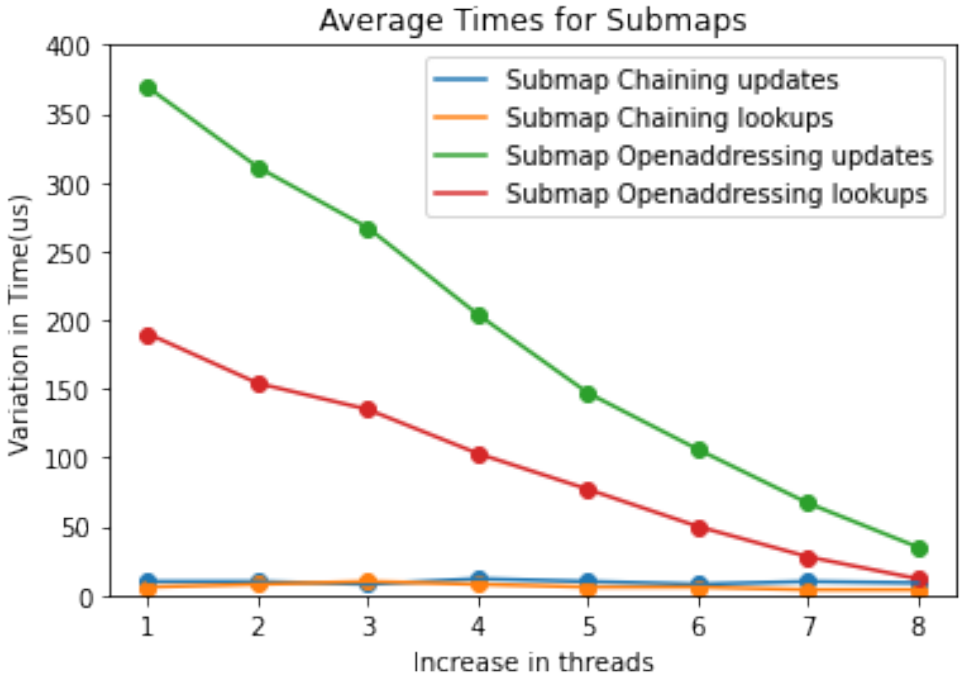
- The values are recorded based on several observations and averaged.
- For proper time flows and measurements, usleeps are used.
- If not, on using threads, we are unable to properly measure the time.
- There are some uncertainties observed while running code on different machines.
- The usleeps are targeted to obtain results. The time point measurements in cpp can cause some uncertainity. For example, the usleep(10) should sleep for 10 useconds but the Cpp program is actually taking 100us for the operation. This might be due to thread load or context switches.



- Clearly, from the above graph, as the number of threads increase, the average time taken for "Submap Chaining updates" and "Submap Chaining lookups" has gradually decreased, which is the expected output.
- It is clear that Submap Chaining is giving better performance than Chaining.



- Here, from the above graph, as the number of threads increase, the average time taken for "Openaddressing updates" and "Open addressing lookups" is more or less same, whereas for "Submap Openaddressing updates" and "Submap Open addressing lookups", the average time taken is drastically decreasing.
- Clearly, the Submap Openaddressing is outperforming Openaddressing.



- The above graph shows that for less number of threads, the "Submap Chaining" is performing very well compared to that of "Submap Openaddressing"
- But as the number of threads increase, both the submaps are producing productive results(less average times) and are converging to the same time points.
- It is also observed that the submap open addressing is behaving oddly when dealt with more number of threads whilst the submap chaining is performing consistently.

Pseudocode

```
# algorithm
constructor():
    set number of threads
    set sem_flags
    get test cases
    execute each operation

Thread(id) :
    wait_for_semaphore_signal

    if queue[id].has_operations() :
        operation = queue[id].get_operation()
```

```
            queue[id].pop
            execute_operation(operation) on submap[id]

            Thread(id) #recursive function call

        else if sem_flag[id] == false:
            return

execute(key,value) :
    id = get_hash_key(key)
    if thread not created :
        create_thread()
        assign thread to submap[id]

    queue[id].insert(mode,key,value)
    signal_semaphore()

destructor() :
    for i in range(threads):
        if thread[i].is_joinable()
            sem_flag[i] = False
            signal_semaphore()
            thread[i].join()
```

LaTeX generated document

THE END