# CS610: Programming For Performance

Assignment 5

Devesh Shukla (200322)

November 17, 2024

## Problem 1

Below are summarised the throughputs (in number of operations per second) for insertion, deletion and lookup operations on the concurrent hash table for various combinations of primary and secondary hash functions. Here, $h1$ denotes the primary hash function, $h2$ denotes the secondary hash function and $size$ is the current size of the hash table:

$h1(key) = key \ \% \ size$ (Modular hashing)
$h2(key) = 1 + key \ \% \ (size - 1)$ (Simple modulus)

Insertion throughput $= 1.56 * 10^8$
Deletion throughput $= 1.13 * 10^8$
Lookup throughput $= 4.91 * 10^7$

$h1(key) = key \ \% \ size$ (Modular hashing)
$h2(key) = (key/size) \ \% \ size \mid 1$ (Odd Hash function)

Insertion throughput $= 1.57 * 10^8$
Deletion throughput $= 1.15 * 10^8$
Lookup throughput $= 4.82 * 10^7$

$h1(key) = (key \ \hat{} (key \gg 4)) \ \% \ size$ (Bitwise Hashing with shift = 4)
$h2(key) = 1 + key \ \% \ (size - 1)$ (Simple modulus)

Insertion throughput $= 8.33 * 10^6$
Deletion throughput $= 9.25 * 10^6$
Lookup throughput $= 9.26 * 10^6$

$h1(key) = (key \ \hat{} (key \gg 4)) \ \% \ size$ (Bitwise Hashing with shift = 4)
$h2(key) = (key/size) \ \% \ size \mid 1$ (Odd Hash function)

Insertion throughput $= 8.61 * 10^6$
Deletion throughput $= 9.65 * 10^6$

Lookup throughput $= 9.62 * 10^6$

From the above results, we can clearly see that using modular hashing as the primary hash function gives the best throughput for almost all types of operations. Using simple modulus for secondary hashing has a slightly better throughput for lookup operations, while for insertion and deletion operations, odd hash has a marginally higher throughput, though the difference is not significant.

# Problem 2

The following table shows the average time taken $t$ (in seconds) to perform $n$ concurrent operations on the concurrent stack with $thr$ threads:

| $n$ | $thr = 1$ | $thr = 2$ | $thr = 4$ | $thr = 8$ | $thr = 16$ |
|---|---|---|---|---|---|
| 1e5 | 0.008 | 0.036 | 0.027 | 0.035 | 0.033 |
| 1e6 | 0.062 | 0.30 | 0.24 | 0.33 | 0.32 |
| 1e7 | 0.61 | 2.94 | 2.44 | 3.27 | 3.20 |

For each value of n, we observe the following relation between times taken using different number of threads: $t_{thr=1} < t_{thr=2} > t_{thr=4} < t_{thr=8} > t_{thr=16}$.
Clearly, we don't see the time decreasing monotonically as a function of the number of threads. Therefore, we can conclude that our implementation is not strongly scalable with the number of threads. This was to be expected as, all the operations on the concurrent stack operate using the top of the stack only. Since the top of the stack is made atomic in our lock-free implementation, threads have to wait for each other if they access it simultaneously. As the stack operations (push and pop) can't proceed without reading and writing the top of the stack, increasing the number of threads, while reducing the number of operations per thread, increases the wait time for each thread before accessing and modifying the top of the stack. These two effects oppose each other and hence, no monotonic variation of $t$ with $thr$ is observed.