Implementation of Concurrent Hash Tables in Java

Shawn Guydeene Dalton Kajander Blake Celestian

***Abstract*—Abstract goes here.**

***Index Terms*—Concurrent Programming, Hash-table representations**

I. Introduction

The problem that our group chose to tackle, is how one would parallelize a hash table, and see how much of a performance increase, if any would arise from such an implementation. This paper seeks to explain the how hash tables function as a data structure, previous attempts at parallelizing them, our group's attempt at creating a concurrent hash table, and experimental data collected showing the effects of parallelizing on runtime for the program.

II. Research

A. What are Hash Tables?

Hash tables are a data structure that utilizes an associative array, which is a structure that associates keys to specific values, in order to store data. The way in which a hash table uses these keys, is by entering them into a hash function. The hash function will then compute the array index in which the value associated with the key will be stored. The hash function is naturally the most important portion of the hash table's construction. The most commonly used hashing methods uses some variation of modular hashing. Modular hashing works as follows: assuming there is an array of size M, where M is prime, then for any positive integer key k, compute $k \mod M$.

Modular hashing is exapandable to accepting more than just integer as keys. A common data type used as a key for hash tables would be strings. In order to accomplish this, one would simply have to treat the string as though it were a large integer. This concept of converting data types into integers is commonly used as it is much simpler than creating a custom hashing function. A well made hash function will have three main requirements: deterministic - equivalent keys will always produce the same hash value, computationally efficient, and allows for keys to be uniformly distributed. Deterministic is only one of those requirements that is absolutely necessary for any hash function, as it what allows users to searh hash tables for a specific key-value pairing [1].

There is of course, one issue with the hash function: since the hash table is not an array of infinite size, there is guaranteed to be instances where two distinct keys map to the same index when using the hash function. In order to resolve this, several techniques have been developed: linear probing, quadratic probing, double hashing, separate chaining or buckets. The basic concept for linear probing is that if the current key-value attempting to be entered into the hash table collides at position i, then attempt to place it at $i + 1$. If it collides again, keep increasing the index by 1 until it comes across an index in the hash table array that has not been used yet, ending at $i + n$ where n is the number of collisions. Quadratic probing is similar to linear probing, except instead of ending up at $i+n$, the value will be stored at index $i+n^2$, where n is the number of collisions. Double hashing, also known as rehashing, involves hashing the key a second time, with a different hash function. That result is taken, and then used as the step size for a process similar to the previous two methods, resulting in the value being place at index $i + jn$, where n is the number of collisions, and j is the result of the second hash function. These methods, known as open addresssing, work, but as the hash table begins to fill up, they result in the add() operation taking more time to complete due to collisions. Separate chaining, on the other hand, works by having each index of the array be a linked list. Whenever a value maps to thatindex, it will be added into list, which avoids the issue that open addressing presents. Buckets is a similar method to separate chaining, using an array instead of a linked list. While separate chaining and buckets fix the issues with open addressing, they present more of their own, namely that there exist the possibility that a large majority of keys will map to the same index, which leads to a large amount of time being used searching the array or list at that index for the specified key's value [2].

III. Related work

I've seen a similar problem before

IV. Contributions

I've fixed this problem once

V. Algorithms

And this is how I did it!

Listing 1. Testing surce code in LaTeX.

```
class HashTable<T> {
    bool isConcurrent;
    public static void main(String[] args
        ) {
        isConcurrent = true;
    }

    public HashTable<T>(bool isConcurrent
        ) {
        this.isConcurrent = isConcurrent;
    }
```

```
}
```

VI. Experimental Results

Well, it worked on my machine.

References

[1] R. Sedgewick and K. Wayne. "Hash Tables" princeton.edu https://algs4.cs.princeton.edu/34hash/ (accessed Mar. 13, 2022)

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.