Implementation of Concurrent Hash Tables in Java

Shawn Guydeene Dalton Kajander Blake Celestian

***Abstract*—Concurrent programming is using multiple processes at the same time, often reducing total time waiting for operations to complete. A concurrent hash table is a hash table that utilizes concurrent programming in order to reduce the time it takes to do multiple operations, such as the add(), search(), and remove() methods hash tables utilize. It does this by doing different operations at the same time instead of one after another, waiting for the previous to finish before the next can even start. This report is about attempting to create our own concurrent hash table in the Java language instead of using Java's own HashTable class. We will be achieving this by using function keywords like "synchronized" and manually locking and unlocking, experimenting will both options. We will show evidence of reduced time by running the same commands on an unthreaded custom HashTable and on a custom concurrent HashTable.**

***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 expandable 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 search 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 addressing, 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 that index, 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].

B. What is Concurrency?

Now given what a hash table is, this paper will look at what exactly it means for a computer program to be concurrent. Concurrency means that multiple computations will be occurring at the same time. This has become increasing more important, with the clock speed of processors no longer increasing, it results in manufacturers adding more cores to

chips, meaning splitting up computation among those chips is the only way to make it faster. Concurrent programming has two common models: shared memory models and message passing models. This paper will only look at shared memory models as the implementation for a concurrent hash table will utilize a shared memory model. A shared memory model works by having concurrent processors read and write objects into the memory that they share [3].

C. Challenges of Concurrency

The largest challenge of creating any concurrent program is the concept of thread safety. For a data structure or algorithm to be consider threadsafe, it must behave correctly when multiple threads are using it, without additional instructions from the function calling it. There are many strategies to utilize for creating a threadsafe data structure. One such strategy is known as *confinement*, which avoids the issue of threads reading and writing to the same memory location because all the data is confined to a single thread, and the other threads do not have the ability to read or write to memory directly. Another strategy is *immutability*, which means that the information which the threads share will be immutable, meaning they are unable to be altered by the threads. Other than those two, there is also the strategy of utilizing *threadsafe data types*. Naturally utilizing data types that are already designed to be threadsafe helps when creating threadsafe algorithms and data structures because these data types are designed to not fail when multiple threads are being used [4]. There is one final strategy for desgining a threadsafe program, and that is to synchronized the threads so that they do not attempt to read or write to the same memory location at the same time. The most common way to implement thread synchronization in shared memory models is the use of locks. The way locks works is by taking the part of the program that accesses the data, and creating a "lock" around it. What this lock does is prevent multiple threads from being with in the "locked" region of the code. If a thread will attempt to enter the "locked" region, while another thread is already inside of it, the other thread will be a frozen until the lock is removed and the thread will be able to access the region, reinitiating the lock again to prevent another thread from entering. Now that no other threads can enter the "locked" region, the thread inside of it is able to alter the memory location without having any other threads attempting to access it at the same time.

III. Algorithms

A. Coarse-Grain Locking Hashtable

```
public synchronized boolean put(int item)
    {
    // If capacity if full, do not add
    if (capacity >= ARR_SIZE)
        return false;

    // Get key
    int key = hash(item);

    // Check if first try addition
    if (table[key] == null) {
        table[key] = item;
        cleans[key] = false;
    }
    // Otherwise, linearly probe for
       empty indices
    else {
        int i = 1;
        int new_key = (key + i) %
            ARR_SIZE;
        while (table[new_key] != null) {
            i++;
            new_key = (new_key + i) %
                ARR_SIZE;
        }
        table[new_key] = item;
        cleans[new_key] = false;
    }

    // Increase capacity
    capacity++;
    return true;
}
```

Listing 1. Source code for the course-grain put() method.

The code outlined in "Listing 1" is the `put()` method for the coarse-grain hash table created for this project. As it is a course grain hash table, it utilizes the `synchronized` keyword to make a lock on this method, meaning that if one thread were to be in the middle of a call to this method then any thread that attempted to call it would have to wait until the original thread finished it's method call. Since this implementation uses a fixed size hash table, the method will first check to see if the hash table has reached it's maximum capacity. If the hash table is full, then the method returns false, indicating that the attempt to put the item into the hash table was unsuccessful. In the case that the hash table still has unused memory, then the method will create a key for the item by hashing it, and checks to see if the memory location associated with that key has been used yet. If no other key is associated with that memory location, then we add the item to the array at that location, and change the clean array so the hash table will know that data was stored at this memory location for the linear probing in the `remove()` method. If that memory location already has data associated with it, then the method will begin linear probing. The way linear probing works is by simply incrementing the key value by one, and checking that memory location. If the memory location is empty, then the method will store the item at that key value, otherwise, the method will continue to increment and check until it finds a memory location that is unused. After the value has been put in to the hash table, the hash table's capacity variable is increased to signify how many items are in it, and the method call will finally return true to indicate that the

value has been added successfully.

```
public synchronized boolean search(int
    item) {
     int key = hash(item);
     int i = 0;
     int new_key = (key + i) % ARR_SIZE;
     // Linearly probe for the item
     // Dirty indices are considered to be
          a collision, the item may of been
          probed
     // farther down
     while (!cleans[new_key]) {
          if (table[new_key] == null) {
          } else if (item == table[new_key
              ]) {
               return true;
          }
          i++;
          new_key = (new_key + i) %
              ARR_SIZE;
          if (i >= ARR_SIZE)
               break;
     }

     return false;
}
```

Listing 2. Source code for the course-grain search() method.

The code outlined in "Listing 2" shows the `search()` method for the coarse-grain hash table created for this project. This method's purpose is to accept a value, and then check the locations in the hashtable where it could be stored for it. It does this by first acquiring the first key for that item with the `hash()` method. After the method has the initial key associated with that value, it will check to see if that key is stored in the array at that location. It does this by creating a loop that checks to see the `clean` array contains any data. What this accomplishes is a simple test, if there is no data in the `clean` array then the item has never been put into the hash table, otherwise, there is a possibility that the value has been put in and is located here or further into the hash table due to linear probing. In the case that the locations checked in the `clean` array are ever empty, the while loop is broken and the method returns false, indicating that the item has not been found. The while loop will continually check different locations inside the `clean` array to see if they are equal to the item that is being searched for. If the item is found, the method returns true, otherwise the method will loop until every single memory location in the hash table has been checked and returns false.

```
public synchronized boolean remove(int
    item) {
     int key = hash(item);
     int i = 0;
     int new_key = (key + i) % ARR_SIZE;

     // If item not in hash table, no need
          to look for it again
     if (!search(item)) {
          return false;
     }

     // Linearly probe for the item
     // Dirty indices are considered to be
          a collision, the item may of been
          probed
     // farther down
     while (!cleans[new_key]) {
          if (item == table[new_key]) {
               table[new_key] = null;
               break;
          }
          i++;
          new_key = (new_key + i) %
              ARR_SIZE;
     }

     // Decrease capacity
     capacity--;
     return true;
}
```

Listing 3. Source code for the course-grain remove() method.

"Listing 3" contains the code for coarse-grain locking hash table's `remove()` method. The method is used to remove the value given to the method from the hash table. It begin's by calculating the initial key for that value by utilizing the hash table's `hash()` method. After it calculates the key, the method makes a call to `search()` to see if the value is in the hash table. If the value has not been added, then there is no way to remove it and the `remove()` function returns false, due to the impossible nature of removing it. Now that the `remove()` method is aware that the value it is trying to remove exists within the hash table, it needs to find and remove the value. It does this by utilizing the same exact loop from the `search()` function, which revolves around testing locations in the `clean` to see if there has ever been data at this memory location in the hash table. The loop will continually check every single location that the data could have been placed into until it finds it, which the method is guaranteed to do as the call to `search()` indicated that the value was in the array. Once the memory location with the value has been found, it is set to null in order to remove it from memory. After that, `remove()` will decrease the `capacity` variable, so that the hash table knows how many values are stored inside of it, and then returns true to indicate that the value was removed from the table successfully.

B. Fine-Grain Locking Hashtable

```
public boolean put(int item) {
```

```
    // If capacity if full, do not add
    if (capacity >= ARR_SIZE)
        return false;

    // Get key
    int key = hash(item);

    // Quadratically probe for empty
       indices
    int i = 0;
    int new_key = (key + (i * i)) %
       ARR_SIZE;
    while (true) {
        if (table[new_key] == null) {
            locks[new_key].lock();
            // Make sure still null
            if (table[new_key] == null) {
                table[new_key] = item;
                cleans[new_key] = false;
                locks[new_key].unlock();
                break;
            }
            locks[new_key].unlock();
        }

        i++;
        new_key = (new_key + (i * i)) %
            ARR_SIZE;
    }

    // Increase capacity
    capacity++;
    return true;
}
```

Listing 4. Source code for the fine-grain put() method.

The source code shown in "Listing 4" contains the `put()` method for a fine-grain locking hash table. This method resembles the `put()` method for the coarse-grain locking hash table, but differs in a few key ways. The first notable difference is the lack of the `syncrhonized` keyword in the method declaration. This means a call to this method would not block other threads from calling the method until the call finishes. Rather, this method implements the use of locks to block threads at the critical sections of the code, the critical section being the writing of the item into the hash table. What this code does is check to see if the memory location that the data will be stored in is null, and then acquires the lock for that location. Once the lock is acquired, the method checks once again that memory location is null to ensure no other calls to the method wrote data in the time between checking and acquiring the lock. Once the data is added to the hash table, the lock will be relinquished and other thread will be able to continue with their method calls. The only other notable difference between the coarse-grain and fine-grain methods is that the fine-grain one utilizes quadratic probing.

```
public boolean search(int item) {
    int key = hash(item);
    int i = 0;
    int new_key = (key + (i * i)) %
       ARR_SIZE;
    // Quadratically probe for the item
    // Dirty indices are considered to be
       a collision, the item may of been
       probed
    // farther down
    while (!cleans[new_key]) {
        if (item == table[new_key]) {
            return true;
        }
        i++;
        new_key = (new_key + (i * i)) %
            ARR_SIZE;
        if (i >= ARR_SIZE)
            break;
    }

    return false;
}
```

Listing 5. Source code for the fine-grain search() method.

The fine-grain `search()` method is once again notably similar to its coarse-grain counterpart. The only difference between the two is difference in probing methods. The reasoning behind this is that since search only reads data from the array and does not write any, there is no reason for it to acquire a lock at any point in its runtime. The program has no reason to block access to the memory location in order to read it, and if data was being written to the memory location during the call to `search()` then search would be blocked by `put()` or `remove` acquiring the lock to that memory location.

```
public boolean remove(int item) {
    int key = hash(item);
    int i = 0;
    int new_key = (key + (i * i)) %
       ARR_SIZE;

    // If item not in hash table, no need
       to look for it again
    if (!search(item)) {
        return false;
    }

    // Quadratically probe for the item
    // Dirty indices are considered to be
       a collision, the item may of been
       probed
    // farther down
    while (!cleans[new_key]) {
        if (item == table[new_key]) {
            locks[new_key].lock();
```

```
                table[new_key] = null;
                locks[new_key].unlock();
                break;
            }
            i++;
            new_key = (new_key + (i * i)) %
                ARR_SIZE;
        }

        // Decrease capacity
        capacity--;
        return true;
    }
```

Listing 6. Source code for the fine-grain remove() method.

For the fine-grain `remove()` method, we start the same way as the other `remove()` methods used in the other hash tables. Get the key and search the table for a value with that key. If it is not found, there is no need to try to remove the value, otherwise we just continue. If the table does have the value, we start to look for that value, and in this version, we quadratically probe to search for it. We continually check if the node is clean or not, dirty nodes mean there was never a value there and not worth checking onwards. When we encounter a clean node, we end. If we find the node we are searching for, we first start by locking it. After locking, we set the node to null and unlock it. After that, we break from the search and return true because we successfully removed the value. If we are still going through dirty nodes, we quadratically increase the index because we are quadratically probing. After all of that, we decrease the capacity of the table and return true.

IV. Experimental Results

We began by testing how the standard hash table will perform when run sequentially, and then when multi-threaded. After those tests, we created a locking hash table and proceeded to run tests on that version. After both of those tests, we deicded it would be a good idea to test our hash tables to Java's native concurrent hash map, the closest thing to a concurrent hash table Java has to offer natively. Our measurement for testing was timing how long each hash table takes to add a specific amount of numbers, make sure each one was added, remove all those numbers, and then make sure each was removed. We recorded all the measurements into a CSV file and used a Python program to analyze the numbers, using Pandas to do the math and Matplotlib and Seaborn to visualize the data gathered. The file used to evaluate the data, "evaluation.py," can be found in the GitHub repository used for this project. We were able to gather a plentiful amount of information regarding each hash table and the time it takes to do these processes. The data we gathered includes total time taken, the CPU used, the amount of tests, the time it took to do each test for each hash table, the average, low, and high times, and the standard deviation. All times recorded are in milliseconds.

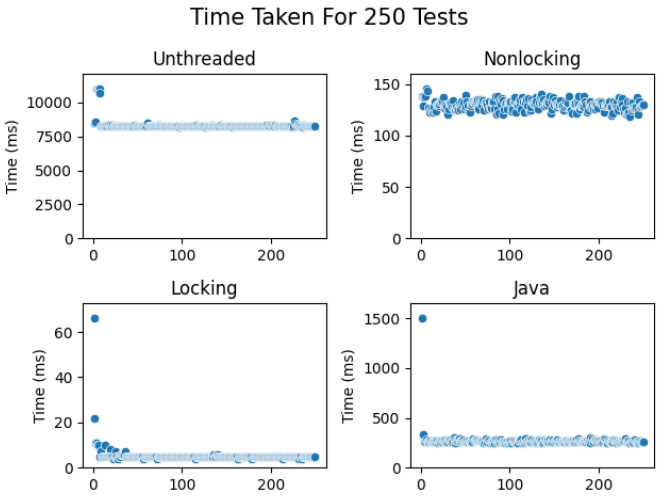

Fig. 1. Graphs of each test for each hash table. The x-axis is the test number, the y-axis is the time in ms the test took.

TABLE I
Experimental Results

Data Calculation	**Hash Table Version**			
	Nonlocking	***Threaded***	***Locked Threaded***	***Java***
Average	8,135 ms	130 ms	5 ms	272 ms
High	10,978 ms	145 ms	66 ms	1,499 ms
Low	8164 ms	118 ms	4 ms	247 ms
Deviation	376.3 ms	4.5 ms	4.1 ms	78.7 ms
Total time	**2,180,629 ms**			

CPU used for testing was a AMD Ryzen 5 3600X 6-Core Processor

Gathered from this data, we were able to come to some conclusions about the types of hash tables we created and the Java native concurrent hash map. The hash table that was created without concurrency in mind was slow, taking approximately 8 seconds per test. The deviation was relatively low at around 375 ms, 4% of a difference from the average. The same hash table tested concurrently was much faster, able to do the same test but 60 times faster. The standard deviation, compared to the previous test, was significantly lower. The test done after that one was the locking hash table, which performed significantly faster than even the concurrent non-locking one, completing the tests 26 times faster, 5 ms per test. Contrary to previous thoughts, the Java hash map performed the second worst, averaging at 272 ms per test.

These tests and theories gave us insight into some possible issues with our versions. The first of which, signified by the locking hash table tests, is that our testing methodology may not be perfect and have flaws that would make it seem good, but only in the case we gave it. Instead of thee same test everytime, we could randomize more numbers being added and removed, but adding that ability would slow the overall tests as we need to know which random numbers are being added and removed. That would slow down the tests themselves instead of changing the outcome of the tests. While

not a flaw, another phenominom that was noticed was the amount of time each hash table took to complete the first set of tests. The unthreaded tests took, on average, 25% longer to finish the first set of tests, eventually converging to about 8 seconds. The same can especially be seen in the locking and Java tests, taking approximately 16.5 and 5.5 times longer than the average. The speculated reason is the amount of time it takes for everything to be set up correctly the first time around. After that, the classes are already in memory and nothing new need to be set up.

```
    public void run() {
        for (int i = (this.thread_num *
            8089); i < (this.thread_num *
            8089) + 8089; i++) {
             table.put(i);
        }
        for (int i = (this.thread_num *
            8089); i < (this.thread_num *
            8089) + 8089; i++) {
             if (!table.search(i))
                 insertion_failures++;
        }
        for (int i = (this.thread_num *
            8089); i < (this.thread_num *
            8089) + 8089; i++) {
             table.remove(i);
        }
        for (int i = (this.thread_num *
            8089); i < (this.thread_num *
            8089) + 8089; i++) {
             if (table.search(i))
                 removal_failures++;
        }
    }
```

Listing 7. Source code for the experiment used in each test

In order to test if the hash tables work correctly, the `run()` function was created, one for each experiment, each exactly the same, except for the Java hash map. The Java hash map's `run()` function used `table.put(i, i)` and `table.contains(i)` in place of `table.put(i)` and `table.search(i)`, respectively. In each test, each thread (for the threadded tests) was given a number, from 0 to the number of threads used in the test. The thread will then proceed to `put()` in a series of numbers. After the series of numbers is added the the table, the thread will then, using the same series of numbers, search the table for those numbers. If the table failed to find a number, the insertion failures would increase, signified by the `insertion_failures++;`. After that, the thread will then remove the same series of numbers from the table. Similar to the previous search, the thread will then search for the same series of numbers. However, if the number was found, the removal failed and the removal failure count would increment. This is signifed by the `removal_failures++;` when `table.search()` was successful. The variables `insertional_failures` and `removal_failures` are both public and static, so each thread and the overall experiment have access to it. The `table` variable, for each experiment, was different per hash table testing. For nonthreaded and locked, `table = new HashTable()`. For Java hash maps, we set `table` to `ConcurrentHashMap<Integer, Integer>()` and for locked, we set it to `LockingHashTable()`

REFERENCES

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

[2] S. Mitra. “Hashing.” utexas.edu. https://www.cs.utexas.edu/ mitra/csSpring2017/cs313/lectures/hash.html (accessed Mar. 13, 2022).

[3] S. Amarasinghe, A. Chlipala, S. Devadas, M. Ernst, M. Goldman, J. Guttag, D. Jackson, R. Miller, M. Rinard, and A. Solar-Lezama. “Reading 17: Concurrency” mit.edu. https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/ (accessed Mar. 14, 2022).

[4] S. Amarasinghe, A. Chlipala, S. Devadas, M. Ernst, M. Goldman, J. Guttag, D. Jackson, R. Miller, M. Rinard, and A. Solar-Lezama. “Reading 18: Thread Safety” mit.edu. http://web.mit.edu/6.005/www/fa14/classes/18-thread-safety/ (accessed Apr. 20, 2022).

[5] S. Amarasinghe, A. Chlipala, S. Devadas, M. Ernst, M. Goldman, J. Guttag, D. Jackson, R. Miller, M. Rinard, and A. Solar-Lezama. “Reading 20: Queues & Locks” mit.edu. http://web.mit.edu/6.005/www/fa14/classes/20-queues-locks/(accessed Apr. 20, 2022).