Lock-based Closed Addressing Hash Set

Name: Beiming CUI

Lock-based Closed Addressing Hash Set

Object is to implement a hash set to ensure that contains(), add(), and remove() calls take constant average time. There is introduction to these 3 implemented functions:

- add(x) adds x to the set. Returns true if x was absent, and false otherwise.
- remove(x) removes x from the set. Returns true if x was present, and false otherwise.
- contains(x) returns true if x is present, and false otherwise.

1. Implementation of a Striped Hash Set

For closed addressing hash set is implemented by a vector, called the table. Each table entry is a reference to one or more items, traditionally called a bucket, using the vector.

A hash function maps items to integers so that distinct items usually map to distinct values. Then use this distinct value modulo the table size to identify the table entry associated with that item.

For concurrent, we use an array of mutex. Each mutex is corresponding to one or several entries in the table (when resize happened). So when we want to access a bucket in the table, first we will lock this bucket, and then when we finish accessing, we will release the lock. So we will have the concurrency between each entry.

For the resize part, we will try to lock all the mutex in ascending order to avoid the deadlock. When we get all the locks, we will lock the whole set and then start to copy all the elements from the old table to a temporary vector, then we resize the table, and copy all the elements in the temporary vector back to the table. After, moving the whole old table, we will release all the locks.

2. Data Structure

The table[] field is a vector of buckets. Each entry of this vector is also a vector, which is used to represent each bucket.

set_size field is the total number of items in the hash set.

bucket_num field is the length of the table, that is, the number of buckets in table.

mutexes field is an array of mutex which are used to synchronize each bucket independently.

3. Approve Correctness

3.1 Approve for Constant time:

For each functions in hash set, add(), contains() and remove(). There are mainly 2 steps to find an element.

The first step, hashing the target element, takes constant time and then we will find this element belong to which bucket in the table.

The second step, we need to traverse the target bucket. Only when the bucket is short enough so that this traversal takes constant time. Otherwise the time will be linear to n which is the length of bucket.

So what we need to do is to make sure that the length of bucket is constant. Here is the solution that we will resize the table when there are too many elements in the table. In this way, we can ensure that the length of bucket is constant.

For closed-addressing algorithms, one simple strategy is to resize the set when the average bucket size exceeds a fixed threshold.

3.2 Approve for Concurrency:

Each mutex is corresponding to one or several entries in the table (when resize happened). Every time when we want to access a bucket in the table, first we will lock this bucket, and only then we will get the access. So it is safe for us to access the table and also there is concurrent between different entries.

For resize, each time we will lock all the locks and we record the old table size for each thread, when it finds some threads already change the table, it will not change it.

3.3 Several Test Example:

Case1: all the threads want to access a same element into set.

Code: for each add, contains and remove, we first need to get the lock for that entry. So the collision will not happen.

```
void LockHashSet::acquire(int element){

    int lock_num = getHashCode(element) % BUCKET_NUM;
    mutex &m = mutexes[lock_num];
    m.lock();

}
```

```
void LockHashSet::release(int element){

    int lock_num = getHashCode(element) % BUCKET_NUM;
    mutex &m = mutexes[lock_num];
    m.unlock();

}
```

Result:

Create 3 threads, all them want to add a same element into the set, only one of them will successes.

Case2: need to resize the table

Code: when the set_size is too big, we will resize. First, record the old capacity to make sure that only one will resize. Lock all the lock to make sure that there will be not any add(), contains() and remove() happened.

```
bool LockHashSet::policy(){

    return set_size / table.size() > 4 ? true : false;
}
```

```
int oldCapacity = table.size();
for(int i = 0; i < mutexes.size(); i++){
    mutexes[i].lock();
}
if(oldCapacity != table.size()){

    for(int i = 0; i < mutexes.size(); i++){
        mutexes[i].unlock();
    }
    return;
}
```

Result:

First, set the table only has 2 entries. After adding 50 elements into it by 5 different threads, we find that the table is resized to have 16 entries.

Case3: large number of operations

Code:

In resize(), we lock it in ascending order, so we will avoid deadlock, and for add(), contains() and remove() it only need one of these lock, so there will not be deadlock.

```
int oldCapacity = table.size();
for(int i = 0; i < mutexes.size(); i++){
    mutexes[i].lock();
}
if(oldCapacity != table.size()){

    for(int i = 0; i < mutexes.size(); i++){
        mutexes[i].unlock();
    }
    return;
}
```

Result:

Create 10 threads and each threads do 250 operations (100 add, 100 remove, 50 contains). From the performance, we find that there is no deadlock happened.

4. Test Performance:

4.1 Correctness and Runtime

For test each function, create 3 threads, each thread will try to do 20 operations to test the correctness and runtime. Runtime is in microseconds form. BUCKET_NUM is set as 4 which means there will be totally 4 locks for entire table.

	Correctness	Runtime(us)
Add()	100%	315/1252
Contains()	100%	122
Remove()	100%	107

4.2 Large Scale Operations

Create 10 threads to begin to work at same time. Each threads do 10000 times add, 10000 times contains, 5000 times remove which means there will be total 25000 operations. Runtime is 206664 microseconds.

4.3 Compare with other implementations

For each thread, we will use rand() function to generate integer from 1 to 1000. And we will do 20000 times add operations, 20000 remove operation and 10000 contains operations for each threads. Time is in ms form. BUCKET_NUM is set to be 10, which means that there will be total 100 lokcs for the entire table.

Num of threads	ClosedAddress HashSet (Coarsed Lock)	ClosedAddress HashSet(Striped Lock)	LockFree HashSet	OpenAddress HashSet
10	1525.79	340.067	346.129	1206.19
20	2649.01	688.371	573.886	2266.95
100	19297.6	2911.38	3053.05	/