CSE475-Assignment 3 Report
Open-address Hash Set Implementations

< I > Author:

Zhaohan Xi, zhx516@lehigh.edu

< II > File structure:

There are three ".h" file and a "hash.cpp" file in my submission, each ".h" file corresponding to a version of hash set, they are sequential version, concurrent version and transactional version respectively.
Also, there is a make file with them.

< III > Run and test:

Just use "cd" to locate the directory and type "make", then " cd obj32/" and "./hashset", you can see some running outcome like below (picture are screenshot of default results when running on sunlab):

```
Number of thread is: 8
Initial capacity is: 10000
Operation number is: 5000

SEQUENTIAL TEST:
sequential version takes : 0.000763488 seconds
After operations, set valid size is: 3140

CONCURRENCY TEST:
concurrent version takes : 0.000565763 seconds
After operations, set valid size is: 3140

TRANSCATION TEST:
transactional version takes : 0.00227154 seconds
After operations, set valid size is: 3140

Actual size should be: 3140
```

By default, number of thread is 8, initial capacity is 10000, operation number is 5000, and running time of three versions of hash set will shown below. Also, there is an outcome of actual hash set in C++, which used as reference to shown whether my outcomes are accurate enough.

Change the parameters:

"./hashset -c <int>" to change the initial capacity,
"./hashset -n <int>" to change the number of operations,
"./hashset -t <int>" to change the number of threads,
"./hashset -h" to show the helper message.

< IV > Brief introduction of my hash set implementation

IV-1: Hash set structure

I use two array, table0 and table1, to achieve two hash sets, each initialized by a fixed capacity, and each position is a vector: vector< T >, T is a template type for a general capability for any type of data, such as integer, character, string, etc.

In sequential version, each position in the table at most contains on vector element, in concurrent and transactional version, each position is list of vector elements, new element are pushed back at the end and old element are popped out at the beginning. You can see my codes for more details.

IV-2: About main function

Here are the procedure of how my main function works:

Section1: set some parameters, such as thread number, initial capacity and operation numbers, etc.

Section2: create three arrays, the "addedArray" stores elements we want to add to each set, the "removedArray" stores elements we want to remove from set, the "initialArray" stores stores elements we want to use for "populate" to initial our sets by 1024 elements.

Section3: add and remove a reference set "realSet", which has the correct result we used for matching whether my outcomes are correct and accurate enough.

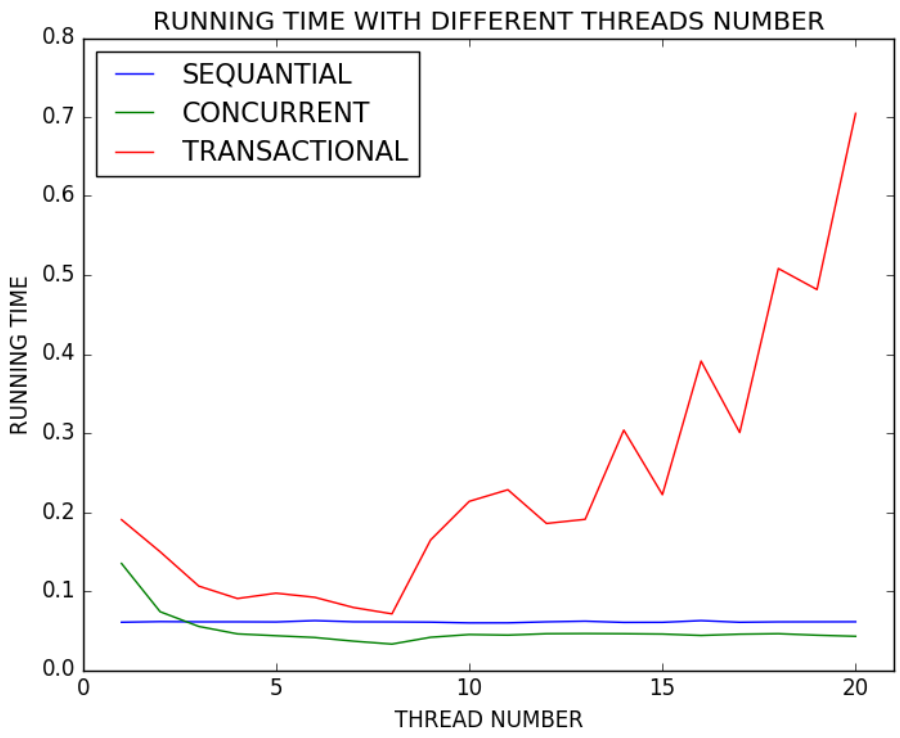
Section4: Sequential part create only one thread and put the "sequential_test" function to test performance of sequential set.

Section5: Concurrency part create many threads to concurrently add elements, then create same amount of threads to concurrently remove elements, to test the performance of concurrent set.

Section6: Transaction part create many threads to concurrently add elements, then create same amount of threads to concurrently remove elements, to test the performance of transactional set.

< V > Plotting the results (all data used to plot are test results on SunLab system)

V-1: When only changing the number of threads (capacity = 1000000, running time measured by seconds)



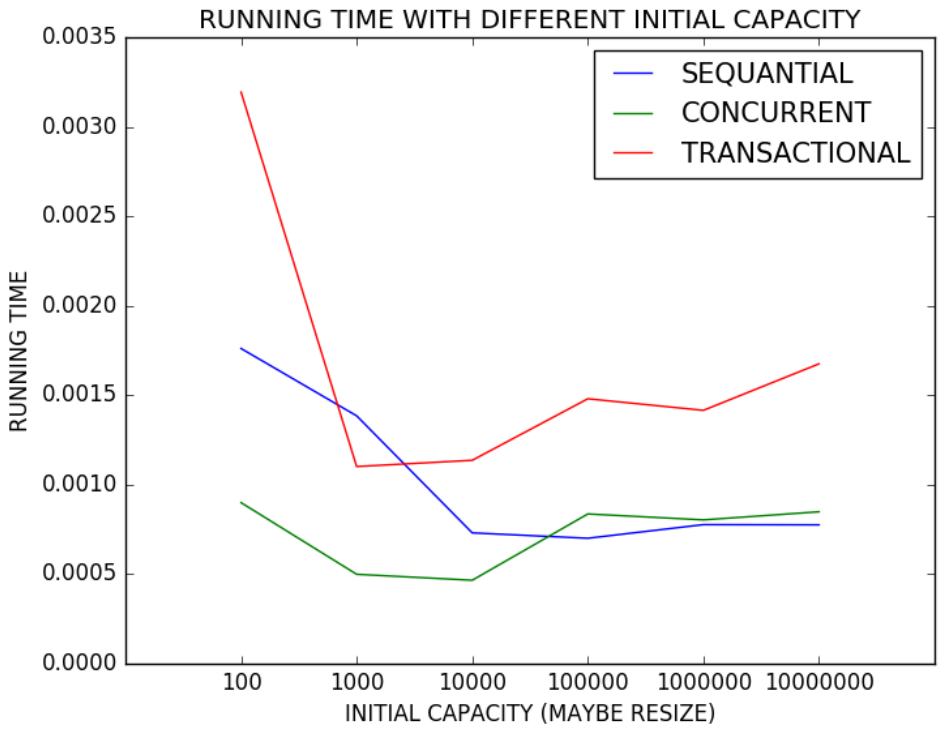
Some descriptions:

(1) When increasing the threads, it's normally that running time of concurrent > sequential > transactional, even transactional version is also a concurrent version, in all tests it's obviously slower that the other two versions.

(2) The default of operation number is 0.5*capacity, in some thread number set it may have a little modification since I need to make (operation_num/thread_num)%5==0 to ensure each thread will work with equal number of add and remove operations (in my "hashset.cpp" there is also a comment have an explanation about this issue), these slight modification doesn't affect the running time, thus we could treat operation number of each thread as 5000.

(3) There is no resize operation during these series of tests, since with resizing the transaction will not work normally, seeing "VI-3: Defects of transaction memory usage".

V-2: When only changing the initial capacity (threads number = 2, operation number = 5000, running time measured by seconds)

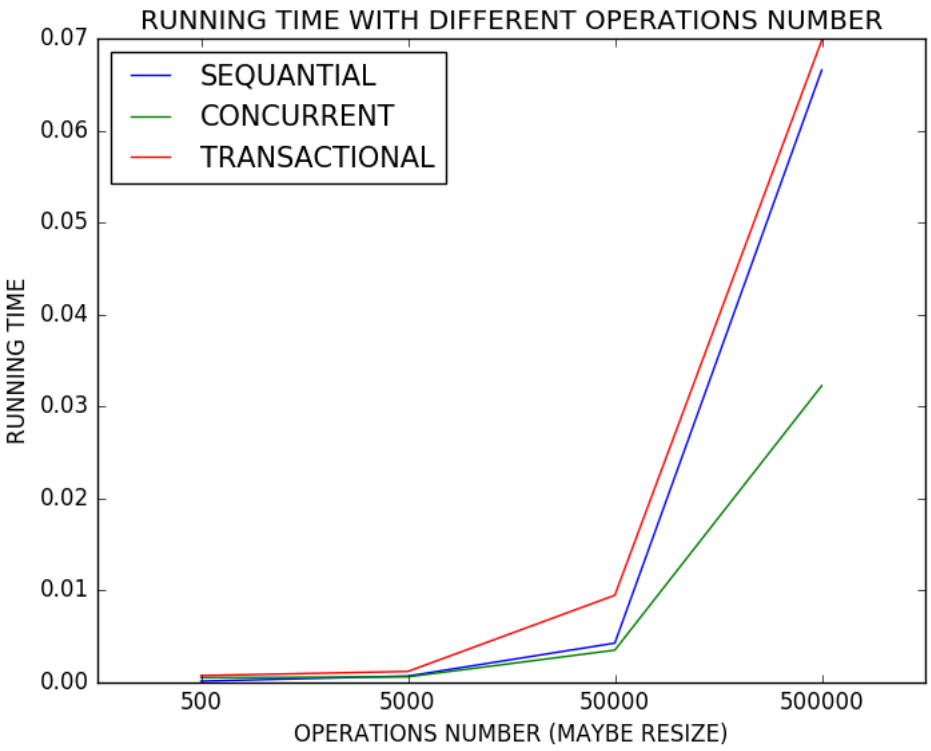


Some descriptions:

(1) The default operation number is 5000, thus when capacity = 100 or 1000 in my tests there are resizing exists, and when capacity >= 10000, there are not resizing, that's why here is a significant decreasing in this plot, since resizing really takes time.

(2) Normally, when there is no resizing and as initial capacity increasing, the running time will also increase stably.

V-3: When only changing the operation number (capacity = 100000, threads number = 4, running time measured by seconds)



Some descriptions:
Intuitively, as number of operation increasing, all run times of three versions hash set will increase, however, when number of operation is very bigger (such as million-level) and need to resize, the transactional version will not work normally (seeing "VI-3: Defects of transaction memory usage"), thus I only plot four points for each version.

< VI > Collection of findings:
This is not a tutorial and I am not an expert of transaction memory toolkits, thus here I want to share some findings during my tests with not very deep technical introduction.

VI-1: About "__transaction_atomic"
Implementation methods on "concurrent_hashset.h" and "transactional_hashset.h" are quite same but only the way to achieve lock has difference.
In "concurrent_hashset.h" I used std::mutex to lock and un_lock as textbook has introduced, in "transactional_hashset.h" the "__transaction_atomic" is a substitution of std::mutex, but during my works I find some significant difference.
The "__transaction_atomic" can be seen as allocate a fixed block of memory on one thread, therefore, any function trends to expand the memory will cause "unsafe error" when compiling,

such as vector.push_back(), vector.insert(), but other methods which not need more memory will not cause this error, such as vector.at(), vector.erase(), etc.
Thus, when using "__transaction_atomic", it's better not try to expand memory, designing a type of data structure with fixed memory occupation is important, otherwise, we may use key words "__attribute__((transaction_pure))" to label the unsafe function to safe one.

VI-2: About "__attribute__((transaction_safe))" and "__attribute__((transaction_pure))"
Initially, I want to label the functions who trend to expand memory and cause unsafe error as safe, thus I used the "__attribute__((transaction_safe))" before definition of a function, but this label not always work especially when I call vector.push_back() or vector.insert() or other functions that will expand memory, In this time, I use "__attribute__((transaction_pure))" as label instead of "__attribute__((transaction_safe))", as you can see in my codes, this works like a compulsory command to treat current function even thought there are some unsafe methods including inside.

VI-3: Defects of transaction memory usage

(1) When we need to resize (we can test this by setting initial capacity << operation number) our hash sets, it's high probability that the transactional version doesn't work, it may cause ***"segmentation fault"***.
(2) When the capacity and operation number is very large (such as million-level), it's also high probability that the transactional version will cause "segmentation fault".
(3) Transaction memory toolkits are not very stable, for example, some running outcomes may not be very accurate, especially operation scale is very large, as pictures shown below:

```
Number of thread is: 8
Initial capacity is: 1000000
Operation number is: 500000

SEQUENTIAL TEST:
sequential version takes : 0.0635129 seconds
After operations, set valid size is: 213067

CONCURRENCY TEST:
concurrent version takes : 0.0335118 seconds
After operations, set valid size is: 213067

TRANSCATION TEST:
transactional version takes : 0.0965621 seconds
After operations, set valid size is: 213070

Actual size should be: 213067
```

```
Number of thread is: 8
Initial capacity is: 1000000
Operation number is: 500000

SEQUENTIAL TEST:
sequential version takes : 0.0628007 seconds
After operations, set valid size is: 213067

CONCURRENCY TEST:
concurrent version takes : 0.0329608 seconds
After operations, set valid size is: 213067

TRANSCATION TEST:
transactional version takes : 0.0971649 seconds
After operations, set valid size is: 213072

Actual size should be: 213067
```

(4) Even though transactional version is also a version of concurrency, but its running time is very slow, it's normally running time of transactional hash set slower than even sequential version, when I test on SunLab system.