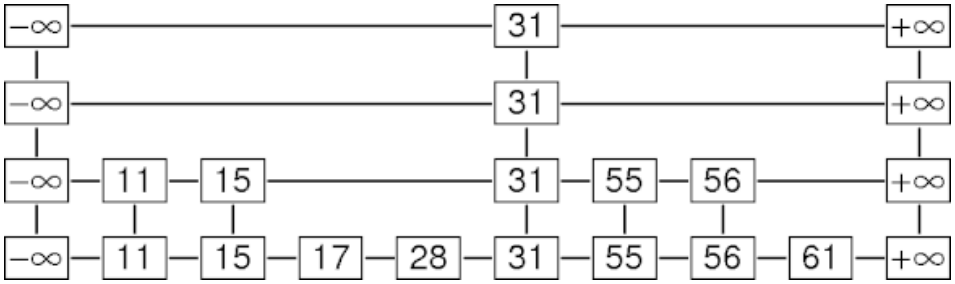
CSCI 7000-017 - Concurrent Programming

Shreyas Gopalakrishna

Project – Concurrent Skip list

Skip list

A skip list is probabilistic data structure with a logarithmic insertion, deletion and search in the average case with an ordered sequence of elements while maintaining a linked list like structure. The skip list is built on multiple layers. The bottom layer has all the elements connected and the topper layers behave as an express lane to skip few elements to transverse the skip list faster. Insert, search, delete all traverse the express lane to find the right position quickly and then perform the necessary action. A depiction of how a skip list is organized is shown in the image below.



Concurrent Skip list

A concurrent skip list is a multithreaded implementation of the skip list data structure where the insert, delete, get and range operations can be performed together across multiple threads. The implementation uses hand over **hand locking to access nodes for writing and atomic variables while reading** (no locks needed while reading).

Implementation Strategies

1. Node

```
class Node{
    public:
        KeyValuePair key_value_pair;
        vector<Node*> next;
        mutex node_lock;
        atomic<bool> marked = {false};
        atomic<bool> fully_linked = {false};
        int top_level;
};
```

The *KeyValuePair* stores a key and value for every node. In my implementation, the key is an integer, and the value is a string. The *next* member variable points to the next node at each level in the skip list. Each node uses a *node_lock* to lock the node when it is being modified. An atomic variable *marked* is used to indicate if a node is being deleted and another atomic variable *fully_linked* is used to indicate if node is completely linked to its successors and predecessors. The member variable *top_level* has the max level until which the particular node is available.

2. **Skip list – insert**

Before inserting an element into the skip list, we check if the element is already present in the skip list and if the node is marked. If the element is already present and node is unmarked, we don't insert the element since it is already present in the skip list. If the element is present and node is not fully linked, then we wait until it is completely linked before inserting. If the element is present and node is marked, it is being deleted, so we wait and try our whole insert algo again later.

To add the element after the above check, we find references to predecessors and successors of the position this element has to be inserted at each level. These references can be corrupted by the time we actually perform the insert. Since each node just has a pointer to the next node, we will only need to hold the lock of the predecessor and not the successor. But we need to be sure that both the predecessor and successor is not marked and the next of the predecessor is the successor at each level. In case these conditions are not met, we wait and try our whole insert algo again later.

To insert, we start holding lock of the predecessor node at each level simultaneously checking the above conditions, if conditions not met, we release the locks held and go for a fresh try to insert. Once the condition is met, we have the lock to all the predecessors, and we can make the insert. To insert, a new node is created by randomly choosing the top level until which it must be available. The successors of the newly created node are linked at every level and then the predecessors at each level are linked to the newly created node. Once all the links are complete the node is marked as fully linked and then we release all the locks of the predecessors held at each level. This completes the concurrent insert.

3. **Skip list – delete**

Before deleting an element from the skip list, we check if the element is present in the skip list and if the node is not present, we return. If the element is present, we check if is fully linked and unmarked if not, we try the delete algo again.

Once the above conditions are met, we find references to predecessors and successors of the position this element is present. These references can be corrupted by the time we actually perform the insert. We try to take the lock of the node to be deleted and then go ahead to try acquiring the locks of the predecessors to the node at each level. While acquiring the locks, we also check if the predecessor is not marked and also if the next element to the predecessor is the current element we are trying to delete. If the conditions are not met, we release lock of the predecessors we are holding, also release the lock of the element being deleted and try the delete algo again.

Once we have all the locks of the predecessors, the required conditions are met, so we now link the predecessors to the successors of the node to be deleted. Once the linking is done, the node is deleted from the skip list and memory for that particular node is freed. After deleting the node, the locks of all the predecessor nodes held are released. This completes the concurrent delete.

4. **Skip list – search (wait-free)**

   The search for an element in the skip list is done by traversing the entire skip list at higher level and dropping to lower levels as the search gets closer to the search key. If a key is found, we check if the node is unmarked and fully linked. It yes, then our search is successful, and we return the value associated with the key. If the node is marked or not fully linked, we return false as the node is marked for deletion or not completely linked after other operations.

   The atomic member variables of the node *marked* and *fully_linked* make sure that we don't need to lock the node to read. Hence making the read or search operation lock free. This **implementation allows multiple readers to execute in parallel**.

5. **Skip list – range**

   The range operation works similar to the search where we traverse the skip list at higher level and drop to lower level as we get closer to the start of the range. When we find key in between the range we need, we add the key value pair to a map. If we encounter a node which is marked, it is ignored. If we encounter a node which is not fully linked, we wait until completely linked and then continue the traversal until we exceed the end of range. The map now contains all the key value pairs within the range which is returned.

[As indicated by the professor, my implementation qualifies for the extra credit for the project since my implementation works by allowing multiple readers at the same time by default.]

Code Organization

The code is organized into different files and modules based on their functionality. Code is written such that each structure has different cpp file and header file.

- The *main.cpp* file handles the reading of command line arguments, initializing and using the slip list based on user input.
- The project also contains 3 unit test files which create sample skip list and run operations on it.
- The *benchmark.cpp* file is used to test the skip list for time taken for different operations.

The *skip_list class* file handles all the operations of the skip list. The usage is as shown below:

```
Skiplist s = SkipList(num_of_elements,fraction)
Skiplist s = SkipList(100, 0.5)
```

This initialization tells the total number of elements the skip list must handle and also the fraction of elements which must exist in the next level. So, 0.5 here indicates that half of the elements from level 0 exist in level 1, and half from level 1 to level 2. This makes sure that the insertion, deletion operations can be performed in $\log(n)$ complexity in the average case.

The `Skiplist` class exposes 4 operations:

- `s.insert(int key, string value)`
  - `inserts key and value to skip list, ignores if already present`
- `s.delete(int key)`
  - `inserts key from the skip list if present`
- `s.search(int key)`
  - `searches for the key and returns value found else empty`
- `s.range(int key_1, int key_2)`
  - `returns a map of key and values found between the range.`

All the above operations handle concurrent operations and are thread safe. The locks and synchronization mechanisms are abstracted in the skip list class.

Output

The project involves multiple unit test cases which can be run based on the execution instructions mentioned in sections below. A sample debug output of few unit tests is shown below

1. Skip list operations – one after the other

```
This Unit test uses 4 Threads. Numbers (1-15) are inserted into the skip list parallelly.
Then a few numbers at random are removed from the skip list parallelly. And next a few
numbers are searched in the skip list.
Finally, few Range operations are done in the skip list parallelly. The output is
dislayed after each step for verification.
Output may be interleaved since multiple threads tend to print together.

---------- Numbers inserted parallelly ----------
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

---------- Skip list after insert ----------
Level 0  -2147483648 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -> 13
-> 14 -> 15 -> 2147483647 ->
Level 1  -2147483648 -> 1 -> 2 -> 3 -> 6 -> 10 -> 13 -> 14 -> 15 -> 2147483647 ->
Level 2  -2147483648 -> 3 -> 10 -> 13 -> 2147483647 ->
Level 3  -2147483648 -> 10 -> 2147483647 ->
---------- Display done! ----------


---------- Numbers deleted parallelly ----------
3 8 9

---------- Skip list after delete ----------
Level 0  -2147483648 -> 1 -> 2 -> 4 -> 5 -> 6 -> 7 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 ->
2147483647 ->
Level 1  -2147483648 -> 1 -> 2 -> 6 -> 10 -> 13 -> 14 -> 15 -> 2147483647 ->
Level 2  -2147483648 -> 10 -> 13 -> 2147483647 ->
Level 3  -2147483648 -> 10 -> 2147483647 ->
---------- Display done! ----------


---------- Numbers searched parallelly ----------
2 11 15
Searching for 11 Search value: 11
Searching for 15 Search value: 15
Searching for 2 Search value: 2

---------- Range between random numbers in Skip list parallely ----------
Range (9, 9) =
Range (13, 13) = 13
Range (6, 14) = 6 7 10 11 12 13 14
Range (1, 2) = 1 2
```

In the above output the start and end at each level is INT_MIN and INT_MAX which are the start and end nodes. The range operation is inclusive of both the keys.

2. Skip list operations – all together

```
This Unit test uses 8 Threads. Numbers (1-20) are used for this unit test.
All threads call a function which performs insert, delete, search and range with random
numbers (1-50).
This simulates all operations parallelly on the skip list.
Output may be interleaved since multiple threads tend to print search value found
together.

The final elements in the skip list will be based on if insert or delete got scheduled
and executed first for any given element.


---------- Vector of numbers used for insert ----------
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

---------- Vector of numbers used for delete ----------
3 8 9 17 18

---------- Vector of numbers used for search ----------
12 13 14

Searching for 12 Search value: 12
Searching for 13 Search value: Not Found
Searching for 14 Search value: Not Found
Searching for 17 Search value: 17
Searching for 12 Search value: 12
Range (6, 15) = 12 13 14 15
Searching for 13 Search value: 13
Searching for 17 Search value: Not Found Searching for 12 Search value: 12
Range (12, 18) = 12 13 14 15 16 18
Range (11, 15) = 11 12 13 14 15
Searching for 14 Search value: 14
Range (10, 18) = 10 11 12 13 14 15 16
Range (4, 13) = 4 5 7 10 11 12 13
Searching for 14 Search value: 14
Range (11, 19) = 11 12 13 14 15 16


---------- Skip list after all operations ----------
Level 0  -2147483648 -> 2 -> 4 -> 5 -> 7 -> 10 -> 11 -> 12 -> 13 -> 14 -> 15 -> 16 -> 20
-> 2147483647 ->
Level 1  -2147483648 -> 5 -> 7 -> 11 -> 12 -> 16 -> 20 -> 2147483647 ->
Level 2  -2147483648 -> 12 -> 2147483647 ->
---------- Display done! ----------

We can see that few of the numbers from the 'delete vector' are deleted from the skip
list and few still exist since insert got scheduled after delete for these numbers.

But we can notice that none of the numbers not present in the 'delete vector' are
deleted. And our traversal of the skip list is sorted. This verifies our skip list to be
accurate.
```

Experimental results, Perf Statistics and Insights

The benchmark.cpp file was used to take the readings below.

When using 4 threads, the skip list implementation handles 1000000 inserts in 0.6 *seconds* and 1000000 deletes in 0.8 *seconds*. While using 8 threads, the skip list implementation handles 1000000 inserts in 3.8 *seconds* and 1000000 deletes in 4.3 *seconds*.

The increase in time is due to the contention which takes place as multiple threads try to contend on recourses as many are trying to modify the skip list and everyone tries to acquire the lock of nodes at every level which they are trying to insert or delete.

When using 4 threads, performing search for 1000000 elements in the skip list took around inserts in $0.3\,seconds$. The same search using 8 threads, for 1000000 elements took about $0.5\,seconds$. The time taken for search is quite comparable due to the lock free implementation. This way, multiple readers read the value and are only waiting if there are incomplete links in the skip list.

The results of time taken in seconds for insert, delete and search individually and all operations performed together are tabulated below

No. of Elements	No. of Threads	Insert	Delete	Search	All operations
100000	4	0.058	0.06	0.02	0.2
100000	8	0.050	0.06	0.03	0.38
100000	12	0.14	0.16	0.08	1.92
1000000	4	0.6	0.8	0.3	1.7
1000000	8	3.8	4.3	0.5	6.1
1000000	12	5.2	7.8	1.9	17.3

We can notice that as the threads increase the time for insertion and deletion increases even though we have same number of elements. This is due to more threads competing for the locks of multiple nodes across different levels in the skip list. We also notice that search takes significantly lesser time as multiple readers can execute in parallel without trying to acquire locks.

High contention

In order to simulate high contention, the benchmark code executes such that it continuously tries to add the same element and delete the same element for certain number of iterations. Doing this make all the threads try to acquire the lock of the element to be deleted and its predecessor. The results of time taken, and page faults are tabulated below

No. of Iterations	No. of Threads	High contention time(s)	No. of Page faults
100000	4	0.9	2685
100000	8	15	4913
100000	12	54.3	7242
1000000	4	9.6	16037
1000000	8	165	27520
1000000	12	611	30435

We can notice that as the threads increase, the execution time increases when benchmarking the high contention simulation. Using more threads increases the number of resources contending for the lock of just the few nodes. This makes all the other nodes wait for the resource. Using higher iterations and high threads increases the execution time when the same few nodes are being modified by everyone.

The number of page faults have a light increase when more threads but a spike in increase when the iterations are increased. This includes the soft and hard page faults. One possible reason for this is, every node which fails to insert or delete and retires actually should read the predecessors and successors needed again due to a page fault during the previous tries. This way as threads and iterations increase high contention scenario might lead to more page faults.

Low contention

In order to simulate low contention, the benchmark code executes such that it inserts to the skip list a wider range of values such that the multiple inserts do not try to acquire lock of the same nodes but try to uniformly acquire locks. This is simulated by inserting all values from 1 to num_iterations so that there is uniform insert and holding locks or contention for resources also takes place uniformly. But, it is to be noted that the higher the level a node is available the higher the chance of threads being contending for it since threads try to acquire lock of element if they are a predecessor in higher levels. The results of time taken, and page faults are tabulated below

No. of Iterations	No. of Threads	Low contention time(s)	No. of Page faults
100000	4	0.06	4,732
100000	8	0.05	4,748
100000	12	0.04	4,760
1000000	4	0.6	15,801
1000000	8	3.9	15,813
1000000	12	5.9	15,831

We can notice that as the threads increase, the execution time has very light increase. Also compared to the high contention scenario the time taken is very less. Since the resources are uniformly distributed the contention is also reduced. The increase in iterations as well as the number of threads don't seem to affect the run time much during the low contention scenarios. The page faults have increased as iterations increase as more resources may move out of memory when the skip list expands with more data. Whereas the increase in threads don't seem to affect the page faults if the iterations remain the same.

Files

File	Description
main.cpp	Based on the input, creates skip list, performs random inserts, deleted, search and range operations.
key_value_pair.cpp	Stores a key and value pair. Here key is an integer and value a string.
key_value_pair.h	Header file with includes the necessary packages and defines the function headers for *key_value_pair.cpp*
node.cpp	Implements the content of a single node in the skip list.
node.h	Header file with includes the necessary packages and defines the function headers for *node.cpp.*
skip_list.cpp	Implements the skip list data structure with insert, delete, search and range operations.
skip_list.h	Header file with includes the necessary packages and defines the function headers for *skip_list.cpp*

benchmark.cpp	Performs benchmark testing using all the operations and simulates high and low contention
unit_test_1,2,3.cpp	Unit tests to verify the working of skip list.
makefile	Makefile to compile code and clean up binaries

Compilation instructions

The package includes a makefile which compiles code based on the following command.

g++ main.cpp key_value_pair.cpp node.cpp skip_list.cpp -o skiplist -pthread
g++ benchmark.cpp key_value_pair.cpp node.cpp skip_list.cpp -o skiplist -pthread
g++ unit_test_1.cpp key_value_pair.cpp node.cpp skip_list.cpp -o skiplist -pthread

Compiling can be done just by running *make*

Execution instructions

The ***main.cpp*** program can be executed as shown below. The main program prints the debug output on the console. Using lower values will help in a readable output on the console.

```
./skiplist [--name] -i <iterations> -t <num_threads> --operation=<combined,
                                separate> [--help]
```

The 3 unit tests files cover a wide range of cases and each file can be run the following way.

./unit_test_1 *or* *./unit_test_2* *or* *./unit_test_3*

The Unit test 1 file prints to console with lower values. The unit test file 2 and 3 are automated and verifies the operations on the skip list.

The ***benchmark.cpp*** file can be executed to perform benchmark on different operations and also to simulate high and low contention. It can be executed based on following syntax with perf for performance analysis.

```
perf stat -d /benchmark [--name] -i <max_number> -t <num_threads> --benchmark=<insert,
   delete, search, range, all_operations, high_contention, low_contention> [--help]
```

Bugs

Some of the known bugs / input / conditions this code will not handle are listed below

1. If code not run using the requited files or commands or wrong input provided.
2. If the input has wrong values such as float or characters or negative values.
3. If the key inserted is not integer and the value is not a string and if numbers provided is more than or equal to the INT_MAX or less than or equal to INT_MIN.
4. If the input is very large to exceed the memory a system can handle.
5. If the number of threads input is less than 1 or a very large number such that the resources aren't available
6. The key and value pairs used for the skip list has to be of type int and string respectively.

References

- Book - Shared-Memory Synchronization - Michael L. Scott
- Book - The Art of Multiprocessor Programming - Maurice Herlihy and Nir Shavit
- http://www.cplusplus.com/reference/vector/vector/
- https://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Options.html
- https://stackoverflow.com/questions/18079340/using-getopt-in-c-with-non-option-arguments
- https://en.cppreference.com/w/cpp/atomic/atomic
- http://www.cplusplus.com/reference/atomic/
- https://bluehawk.monmouth.edu/~rclayton/web-pages/f10-305/skiplistsf01.png
- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/SkipList.html