Modified Skip List in Concurrent Environment

Ranjeet Kaur, Dr. Pushpa Rani Suri, Professor.
Kaurranjeet2203@gmail.com
Kurukshetra University, Kurukshetra

Abstract— Traditional data structures give few considerations to their execution in concurrent environments. It is not sufficient to simply move a traditional data structure into a concurrent environment and expect an improvement in performance by allocating additional resources and processing power. In that direction we present an efficient and practical lock based MSL (modified skip list). MSL is a modified version of basic skip list; we describe methods for performing concurrent access and update using MSL. Experimental result shows that MSL structure is faster than original skip list structure for representation of dictionaries

Index Terms—Concurrency, Lock, Linearizability, MSL Skiplist, Thread.

1. INTRODUCTION

In past decades, with the emergence of multiprocessing systems. There is a steady increase in the number of processors available on commercial multiprocessors. This increase in the availability of large computing platform has not been met by a matching improvement in our ability to construct new data structure. There is a requirement to shift the way we think and construct data structures.
A data structure in a concurrent environment is access by multiple computing threads (or processes) on a computer. The proliferation of commercial shared-memory multiprocessor machines has brought about significant changes in the art of concurrent programming. Given current trends towards low cost chip multithreading (CMT), such machines are bound to become ever more widespread. Shared-memory multiprocessors are systems that concurrently execute multiple threads of computation which communicate and synchronize through data structures in shared memory. Designing concurrent data structures and ensuring their correctness is a difficult task, significantly more challenging than doing so for their sequential counter parts. The difficult of concurrency is aggravated by the fact that threads are asynchronous since they are subject to page faults, interrupts, and so on. To manage the difficulty of concurrent programming, multithreaded applications need synchronization to ensure thread-safety by coordinating the concurrent accesses of the threads. At the same time, it is crucial to allow many operations to make progress concurrently and complete without interference in order to utilize the parallel processing capabilities of contemporary architectures. The traditional way to implement shared data structures is to use mutual exclusion (locks) to ensure that concurrent operations do not interfere with one another. In concurrent search structures, locks are used to prevent concurrent threads from interfering with each other. A concurrency scheme must assure the integrity of the data structure, avoid deadlock and have a serializable schedule. Within those restrictions, we would like the algorithms to be as simple, efficient and concurrent as possible.
Our concurrent MSL is using the locking techniques. This is just the beginning to see how MSL behave in concurrent environment.

2. SKIP LIST AND MODIFIED SKIP LIST

Skip-lists [1] are an increasingly important data structure for storing and retrieving ordered in-memory data. SkipLists have received little attention in the parallel computing world, in spite of their highly decentralized nature. This structure uses randomization and has a probabilistic time complexity of O(logN) where N is the maximum number of elements in the list.
The data structure is basically an ordered list with randomly distributed short-cuts in order to improve search times, see Figure 1. In this paper, we propose a new lock-based concurrent modified skip-list pseudo code that appears to perform as well as the best existing concurrent skip-list implementation under most common usage conditions. The principal advantage of our implementation is that it is much simpler, and much easier to reason about. The original lock-based concurrent SkipList implementation by Pugh [2] is rather complex due to its use of pointer-reversal,

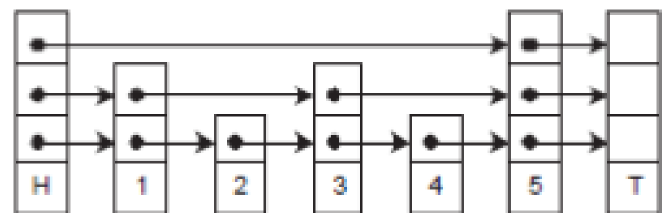
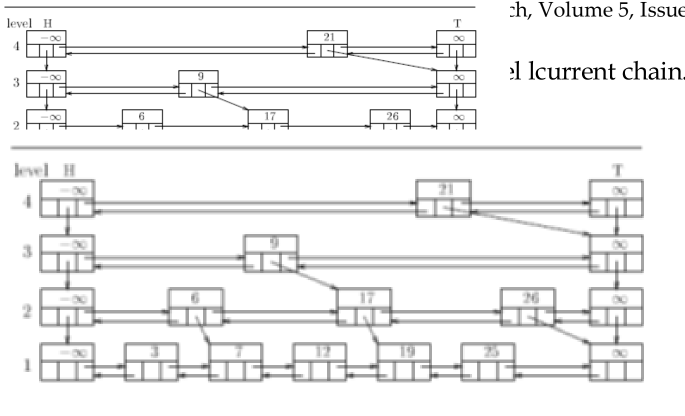


Figure: 1

While the search, insert, and delete algorithms for skip lists are simple and have probabilistic complexity of O (log n) when the level 1 chain has n elements. with these observations in mind [3] introduced modified skip list(MSL) structure in which each node has one data field and three pointer fields :left, right, and down. Each level l chain worked separate doubly linked list. The down field of level l node x points to the leftmost node in the level l-1 chain that has key value larger than the key in x. H and T re-



el lcurrent chain.

Figure: 2

Now in this paper we describe the simple concurrent algorithms for access and update of MSL. Our algorithm based on the lazy-list algorithm of [5], a simple concurrent linked-list algorithm with an optimistic fine-grained locking scheme for the add and remove operations, and a wait-free contains operation.

3. CONCURRENT OPERATIONS ON MSL

We now describe a method for performing concurrent operations' on MSL. In MSL elements of the list are represented by a node. The left pointer of a node points to previous node and right pointer points to the next node in the list and the nodes are kept in sorted order according their keys. The key of anode is given by x→key, the value is given by x→value and the left pointer is given by x→left. The head (H) and tail (T) of a list l is treated as a node and is given by l→H and l→T. For purposes of reasoning about our invariants, the H and T have the sentinel value (-infinity). The term thread refer to a task or process operating concurrently with other threads. A thread obtains a lock on a field only when updating a field that other threads might be attempting to update. While searching for an element, no locks are needed. Only a single thread may hold a lock on a field, and by convention a thread only updates a field if it already holds a lock on the field. If a thread attempts to lock a field that is already locked, that thread is blocked until the lock can be obtained.

4. ALGORITHM

We present a modified skip list algorithm in the context of an implementation of n set objects supporting three methods, search_node, insert_node, remove_node:search_node (key) search for a node with key k equal to key, and return true if key found otherwise return false. Insert_node (key, d) inserts adds d to the set and returns true iff d was not already in the set; remove (v) removes v from the set and returns true iff d was in the set. This paper also shows that the implementation is deadlock-free.the below figure 3 shows the field of a node.

```
node {
int key,
node ** left,
node **right,
node **down,
bool marked,
bool fullylinked,
lock lock
}
```

Figure: 3

4.1. SEARCH_NODE

Searching in MSL is accomplished by search_node procedure see figure 3, which takes a key v and search exactly like a searching in sequential linked list, starting at the highest level and proceeding to the next down level each time it encounters a node whose key is greater than or equal to v.the search process also save the predecessor and successor of a searched node v for further reference.

```
procedure search_node (int key):bool
{
p=h
while(p#NULL)
{
While(p→value→key<key)do
{
p=p→right
}
if(p→value→key==key) then
return true and break
else
p=p→left→down
}
return false
}
```

Figure: 4

Note that search_node does not acquire any locks, nor does it retry in case of conflicting access with some other thread. We now consider each of the operation in turn.

4.2. THE INSERT_NODE OPERATION

The algorithm calls search_node to determine whether a node with the key is already in the list. If so, and the node is not marked, then the add operation returns false, indicating that the key is already in the set. However, if that node is not yet fully linked, then the thread waits until it is (because the key is not in the abstract set until the node is fully linked). If the node is marked, then some other thread is in the process of deleting that node, so the thread doing the add operation simply retries.

If no node was found with the appropriate key, then the

thread locks and updates all the preceding and succeeding nodes returned by search_node, whose pointers are associated with the new node. The randomLevel function is used in the beginning of insert_node operation to determine the level at which new node to be. If the new node to be inserted in between of existing level then there is need to updates the predecessor, successor and the adjacent down level nodes, all depends on the various check applied on new node. If validation fails, the thread encountered a conflicting operation, so it releases the locks and retries.

```
procedure insert_node(key,d)
{
int max=50,temp,current_level
k=randomlevel ()
node * found_node=null,*x,*h1,*t1
node *save[max] ,*t ,*s ,*d ,*u
while(true)  {
//search the msl for a key k of   d and save the necessary pointers.
p=h
i=current_level
while (p#null) do
{
while(p→data→key<key) do
{
save[i]=p
p=p→right
}
If(p→data→key==key)
{
found_node=p
Write "searched node is found at p"and break
 }
else
{
p=p→left→down
i=i-1
}
}
If (found_node != NULL)
{
if( ! found_node→marked)
{
while (found_node→fullylinked)
{ }
}
continue
 }
else
{
//create a new node x and set its value
//connect the new node at level returned by random function
temp=current_level
current_level=current_level+1
if(k>temp)
{
try  {
s=save[temp]
t=save[temp]→right
if  (s!=null)
s→lock.lock()
valid = ! s→marked && ! t→marked  && s→right==t
} }
if (! Valid)  continue
else
{
// create new Head(H1) and tail(T1)
h1→data=∞
h1→right=x
h1→left=null
h1→down=h
h=h1
t1→data=∞
t1→right=null
t1→left=x
t1→down=t
t=t1
//update the new node pointers
x→left=h1
x→right=t1
if((save[temp]→right→down)==NULL && (save[temp]→right→data>x→d))
{
x→down= save[temp]→right
}
}
elseif (k==1) // k is from existing levels
try
{
s=save[k]
t=save[k]→right
d=save[k+1]→right
if  (s!=null&& d !=Null)
s→lock.lock()
d→lock.lock()
valid = ! s→marked &&  ! t→marked&& !d→marked&& s→right==t

}
if (! valid)  continue
else
{
x→left=save[k]
x→right=save[k]→right
save[k]→right→left=x
if((save[k-1]→right→down)==NULL && (save[k-1]→right→data>x→d)) then
x→down= save[k-1]→right
else
x→down=NULL
}
else        //k is in between the current level ,the else of if-then-elseif-else
{
try
{
s=save[k]
t=save[k]→right
```

IJSER

ISSN 2229-5518

```
d=save[k+1]→right
u=save[k-1]→right
if  (s!=null && d !=Null&& u!=null)
{
s→lock.lock()
d→lock.lock()
u→lock.lock()
}
valid = ! s→marked && ! t→marked && !d→marked
&&u→marked&& s→right==t
}
if (! valid)  continue
else
{
node *x
x→left=save[k]
x→right=save[k]→right
save[k]→right→left=x
if((save[k-1]→right→down)==NULL    OR    (save[k-
1]→right→data>x→d)) then
x→down= save [k-1]→right
if    ((save[k+1]→right→down)==NULL    OR
(save[k+1]→right→data<x→d)) then
x→down= save [k+1]→right
else
x→down=NULL
}
}
x→fullylinked=true
return true
finally
{
Unlock(s,d,u)
}
}
```

IJSER

is the linearization point of the remove operation. The remaining part of the procedure accomplishes the "physical" deletion, removing the node from the list by first lockingits predecessors, upward, and downward node. As in the insert_node operation, before changing any of the deleted node's predecessors, the thread validates that those nodes are indeed still the deleted node'spredecessors. This is done using the weak Validate function, which is the same as validate except that itdoes not fail if the successor is marked, since the successor in this case should be the node to be removed thatwas just marked. If the validation fails, then the thread releases the locks on the old predecessors (but not thedeleted node) and tries to find the new predecessors of the deleted node by calling find_Node again. However, at this point it has already set the local isMarked flag so that it will not try to mark another node. Aftersuccessfully removing the deleted node from the list, the thread releases all its locks and returns true.

```
procedure remove_node (int v)
node *delete_node = null;

bool ismarked = false;
int found_node= -1
node *save,*t,*prev, *succ,*up
//search the msl for a key k of   d and save the necessary
pointers.
p=h
i=current_level
while (p#null) do
{
While (p→data→key<key) do
{
p=p→right
}
If (p→data→key==key&&found_node== -1)
{
found_node=i
Write "searched node is found at i" and stop
 }
else
{
save[i] =p
p=p→left→down
I=i-1
}
}
while (true)
{
if ( isMarked || (found_node! = -1 &&right_to_delete
(p,found_node) ) )
{
i f (! isMarked )
{
delete_node=p
delete_node→lock.lock ()
if(delete_node→marked)
{
delete_node→lock.unlock ()
return false
}
delete_node→marked=true
ismarked=true
}
try
{
prev=delete_node→left
succ=delete_node→right
upp=save [i+1]→left→down
if (prev! =null &&succ! =null &&upp!=null)
{
prev→lock. lock ()
succ→lock. lock ()
upp→lock. lock ()
}
valid= !prev→marked && !succ→marked &&upp→marked
if (! valid) continue
delete_node→left→right=delete_node→right
delete_node→right→left=delete_node→left
if (save [i+1]→left→down==p)
save [i+1]→left→down=p→right
}     //end of try
delete_node→lock.unlock ()
```

```
return true
}
finally {
unlock (prev, succ, upp)
}
else
return false
}
}
```

```
Bool right_to_delete(node *c ,int f)
{
return(c→fullylinked &&
c→marked)
}
```

4.4. CONTAINS OPERATION

This operation just calls search_node and returns true if and only if it finds a unmarked, fully linked node with the appropriate key. If it finds such a node, then it's mean the key is in the abstract set. However, as mentioned above, if the node is marked, it is not so easy to see that it is safe to return false.

```
bool contains ( int v )
int  item_found = search_Node ( v  ) ;
 return ( Item_found ≠ -1)
&& item_found→right→fullylinked
&& item_found→right→marked ) ;
```

5. CORRECTNESS

This section, sketch a proof for concurrent modified skip-list algorithm. There are three properties to prove algorithm correctness: that the algorithm implements a linearizable set, that it is deadlock-free, the contains operation is wait-free, which we define more precisely below

5.1. LINEARIZABILITY

Linearizability [4] is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a highdegree of concurrency, yet it permits programmers to specify and reason about concurrent objectsusing known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response.

For the sake of linearizable proof, few assumptions are made like:

i) Nodes are initialized with their keys
ii) Next pointers of nodes are initialized with null
iii) The fullylinked and marked fields of nodes are initilized with false value

With these assumptions in mind we can drive the following lemma:

Lemma for a node n and $0<=j<=$ n→toplayer:

n→right [j]!=null ➔n→key<n→right[j]→key

we can define a relation $\rightarrow_I$ so that $x\rightarrow_i y$ if x →right[i]=n or there exists x′ such that $x\rightarrow_i x'$ and x′→right[i]=n; that is $\rightarrow_i$ is the transitive closure of the relation that relates nodes to their immediate successors at level i

using these observations, we can show that if $x\rightarrow_i y$ in any reachable state of the algorithm ,then $x\rightarrow_I y$ in any subsequent state unless there is an action that remove n out of the level-i list, claim is already proved by[4], and that can also be applicable on our algorithm. Because n must already be marked before being removed out of the MSL, and the fullylinked flag is never set to false value, this claim implies that a key can be removed from the abstract set only by marking its node.

5.2. DEADLOCK FREEDOM

The algorithm is deadlock free because a thread always acquires locks on nodes with larger keys first ,if a thread holds a lock on a node with k then it will not attempt to acquire a lock on a node with key greater than or equal to k.

5.3. WAIT-FREEDOM

The contains operation is wait-free because it does not acquire any locks, nor does it go for retry, it searches the MSL only once

6. CONCLUSION

We introduced a concurrent modified skiplist using a remarkably simple algorithm. Our implementation is raw, various optimization to our algorithm are possible like we can replace the locking with lock free techniques. The wait free traversal in concurrent MSL leads to simpler and possibly more efficient algorithms for related data structures such as dictionaries.

7. REFERENCES

[1] Pugh, W. Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM 33, 6 (June 1990),

[2] Pugh, W. Concurrent maintenance of skip lists. Tech. Rep. CS-TR-2222, 1990.

[3] S. Cho and S. Sahni. Weight-biased leftist trees and modified skip lists. ACM J. Exp. Algorithmics, 1998.

[4] M. Herlihy's and J. Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, July 1990.

[5] Heller, S., Herlihy's, M., Luchangco, V., Moir, M., Shavit, N., and Scherer III, W. N. A lazy concurrent list-based set algorithm. In Proceedings of 9th International Conference on Principles of Distributed Systems (Dec. 2005).

IJSER