The SprayList: A Scalable Relaxed Priority Queue

COP 6616: Multicore Programming

Data Structure Project Final Report

Rajat Bhattacharjee *(PID: 4191543)*
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, United States of America
rajat@Knights.ucf.edu

***Abstract*— There are applications such as task scheduling and discrete event simulation where high-performance concurrent priority queues are essentially used. However, with the increase in the number of threads, the performance of such an abstract data type decreases with respect to the data structure used. It happens because of the sequential bottleneck in accessing the elements at the head of the queue while performing a DeleteMin operation. This project implements the SprayList algorithm which uses the Skiplist data structure to implement the priority queue data type. It is a scalable priority queue with relaxed ordering semantics. The main innovation behind this design proposed by the authors is that the DeleteMin operations avoid a sequential bottleneck by "spraying" themselves onto the head of the SkipList list in a particular fashion.**

The spraying is implemented using a carefully designed random walk, so that DeleteMin returns an element among the first O(p log3 p) in the list, with high probability, where p is the number of threads. The authors prove that the running time of a DeleteMin operation is O(log3 p), with high probability, independent of the size of the list. The paper presents with experiments that show that relaxed semantics allow the data structure to scale for very high thread counts, comparable to a classic unordered SkipList.

Keywords—Priority Queue; SkipList Algorithm; SpayList.

I. INTRODUCTION

One of the main software trends of the past decade is the increased demand for efficient, scalable concurrent data structures. Several impossibility results suggest that not all data structures can have efficient concurrent implementations, due to an inherent sequentiality which follows from their sequential specification. An example of such a data structure is the priority queue, which is widely used in applications such as scheduling and event simulation.

A priority queue stores a set of key-value pairs, and supports two operations: Insert, which adds a given pair to the data structure and DeleteMin, which returns the key-value pair with the smallest key currently present. Sequential priority queues are easy to understand, and are usually implemented using a binary heap.

Unfortunately, heap-based concurrent priority queues suffer from both memory contention and sequential bottlenecks, not only when attempting to delete the single minimal key element at the root of the heap, but also when percolating small inserted elements up the heap.

Implementations such as a SkipList data structure was proposed in order to reduce overheads. SkipLists are randomized list-based data structures which classically support Insert and Delete operations without traversing all possible nodes. This data structure is composed of several linked lists organized in separate levels, each skipping over fewer elements for higher levels of the list. SkipLists are desirable because they allow priority queue insertions and removals without the costly percolation up a heap or the rebalancing of a search tree.

Highly concurrent SkipList-based priority queues have been studied extensively and have relatively simple implementations. Unfortunately, an exact concurrent SkipList-based priority queue, that is, one that maintains a linearizable (or even quiescently-consistent) order on DeleteMin operations, must still remove the minimal element from the leftmost node

in the SkipList. This means that all threads must repeatedly compete to decide who gets this minimal node, resulting in a bottleneck due to contention, and limited scalability.

In this paper, the authors introduce the SprayList, a scalable relaxed priority queue implementation based on a SkipList data structure. The SprayList provides probabilistic guarantees on the relative priority of returned elements, and on the running time of operations. At the same time, it shows fully scalable throughput for up to 80 concurrent threads under high-contention workloads.

The main limitation of past SkipList-based designs was that all threads clash on the first element in the list. Instead, the idea is to allow threads to "skip ahead" in the list, so that concurrent attempts try to remove distinct, uncontended elements. The obvious issue with this approach is that one cannot allow threads to skip ahead too far, or many high priority (minimal key) elements will not be removed.

II. Implementation

In my implementation of SkipList using C++ I am creating a class with defines the structure of a single node containing the key-value pair along with its mark, level of the node and its next pointer which is double pointer(accessed using array brackets), it points to all nodes connected to it i.e. the next node in the same level, node in the upper level and the node in the lower level. The key change from the conventional Skiplist node structure is the inclusion of markers used for insert, deleteMin and find operation.

The purpose of the marker is that when implementing lock-free lists there might be some nodes that might get deleted concurrently while some thread is trying access them. This paper solves that problem by creating a marker in each pointer field of the SkipList. A node with a marked bit is itself marked. The bit is always checked and masked off before accessing the node for physical deletion.

The search operation in its sequential implementation looks for a left and right node at each level in the list. These are the nodes adjacent to it, with key values either less-than or greater than-equal-to the search key. The search loop checks whether nodes are marked and skips over them, since they obviously have been logically removed from the list.

The search procedure also cleans up marked nodes using the compare and exchange operation from the list: if the thread encounters a sequence of marked nodes, these are removed by updating the unmarked successor to point to the unmarked predecessor in the list at that level. If the node accessed by the thread becomes marked during the list traversal, the entire search is re-started from the beginning. The operation returns the node with the required key, if found at some level of the list, as well as the list of predecessors and successors of the node.

Insertion: A new node is created with a randomly chosen height. The node's pointers are unmarked, and the set of successors is set to the successors returned by the search method on the node's key. Next, the node is inserted into the lists by linking it between the successors and the predecessors obtained by searching. The updates are performed using compare-and-exchange. If a compare-and- exchange fails, the list must have changed, and the call is restarted. The insert then progressively links the node up to higher levels. Once all levels are linked, the method returns.

DeleteMin operation traverses the bottom list attempting to acquire a node via compare-and-exchange. Once acquired, the node is logically deleted and then removed via a search operation.

Deletion of a node with key k begins by first searching for the node. If the node is found, then it is logically deleted by updating its mark field to NULL. This operation as discussed earlier will prevent new nodes from being inserted after the deleted node. Hence, all references to the deleted node are removed by simply performing a search for the key in its next operation. The search procedure swings list pointers over marked nodes.

III. Spray Parameters

To avoid congestion at various levels, we need to use the right combination of parameters. The authors had to idea to vary the starting height, the distribution for jump lengths at each level, and how many levels to descend between jumps. The authors however needed to balance the average length of a Spray with

the expected number of thread collisions on elements in the bottom list.
Following is an overview of the parameter choices for the implementation. For simplicity, consider a SkipList on which no removes have yet occurred due to Spray operations. We assume that the data structure contains n elements, where n >> p.

Starting Height: Each Spray starts at list level H = log p + K, for some constant K. I chose this value to be equal to 3 such that I could avoid unexpected errors. Intuitively, starting the Spray from a height less than log p leads to a high number of collisions, while starting from a height of C log p for C > 1 leads to Sprays which traverse beyond the first O(p log3 p) elements.

Jump Length Distribution: The authors chose the maximum number of forward steps L that a Spray may take at a level to be L = M log3 p, where M > 1 is a constant. However, I had to assign M equal to 1 since otherwise the jump length exceeded the length of the list. The number of forward steps at level l, is uniformly distributed in the interval [0;L].

The intuitive reason for this choice is that a randomly built SkipList is likely to have chains of log p consecutive elements of height one, which can only be accessed through the bottom list. The algorithm should be able to choose uniformly among such elements, and we therefore need L to be at least log p. (While the same argument does not apply at higher levels, the authors' analysis shows that choosing this jump length j yields good uniformity properties.)

Levels to Descend. The final parameter is the choice of how many levels to descend after a jump. A natural choice, which the authors used in the implementation, is to descend one level at a time, i.e., perform horizontal jumps at each SkipList level.

In the analysis however they considered a slightly more involved random walk, which descends D = max(1; blog log pc) consecutive levels after a jump at level `. It must always traverse the bottom level of the SkipList (or we will never hit SkipList nodes of height 1) so they round H down to the nearest multiple of D. Note that the authors found empirically that setting D = 1 yields similar performance.

In the following, we parametrize the implementation by H, L and D such that D evenly divides H. The pseudocode for Spray(H; L;D) is given below:

```
x = head        // pointer to current location
                // Assume D divides H
L = H           // L is the current level
while L > 0 do
        Choose j <- Uniform[0;L]
        // random jump
        Walk x forward j_L steps on list at
        height L
        // traverse the list at this level
        L = L - D // descend D levels
return x
```

IV. Implementation

My implementation of this algorithm is done by using C++ and will use POSIX threads in order to be compatible with the RSTM library.

As till now, I have created a structure for the SkipList for its sentinel and regular nodes. Along with its node creation, find() operation and add() operation for searching and insertion operations respectively. Many changes are yet to be made to the code.

The following code defines the structure of the node of the SprayList. It holds a key-value pair, it's highest level value, a mark value (couldn't implement bit-stealing since I got too many errors and wasn't getting an output). The code still holds linearizability conditions without bit stealing by working around it. Two constructors for a regular node and a sentinel node, sentinel nodes for head and tail. Most importantly, the next pointer that holds multiple pointers to various levels.

```
struct Node
{
    public:
        std::atomic<int> key;
        std::atomic<int> NodeLevel;
        std::atomic<int> value;
    // Array to hold pointers to node of different level
        std::atomic<Node**> next;
        std::atomic<bool> mark;
        Node(int key);
        Node(int key, int level);
};
```

The constructor are defined:

```
Node::Node(int key, int level)
{
        this->key = key;
        this->value = rand() % 50 + 1;
        this->NodeLevel = level;
        this->mark = false;
        // Allocate memory to next
        next = new Node*[level+1];

        // Fill next array with 0(NULL)
        memset(next, 0, sizeof(Node*)*(level+1));
}

//Sentinal Node
Node::Node(int key)
{
        this->key = key;
        this->value = -1;
        this->mark = false;
        this->NodeLevel = 5;                    // maxLevel
        // Allocate memory to next
        next = new Node*[(this->NodeLevel)+1];

        // Fill next array with 0(NULL)
        memset(next, 0, sizeof(Node*)*((this->NodeLevel) +
1));
}
```

The first constructor is used for regular nodes, which accepts the key and highest level of the node as parameters.
Note: The Skiplist is organized with respect to keys, not values according to the book 'The Art of Multiprocessor Programming' and the paper from which the data structure is inspired.
It assigns the key and level values, assigns a random value between 1 to 50. Initializes mark to false and allocates memory dynamically to the next double pointer. The only difference with the sentinel node is that' its value is -1, highest level is always the maximal level, according to the SkipList data Structure.

```
class SkipList
{
    // Maximum level for this skip list
    int maxLevel;

    // current level of skip list
    int level;

    // pointer to /*header*/head and tail node
    Node *head;             /*Node *header*/
    Node *tail;
    Node *endTail;
    public:
        SkipList(int, float);
        int randomLevel();
        Node* createNode(int, int);
        bool find(int, Node **, Node **);
        bool add(int);
        Node* DeleteMin();
        void displayList();
};

//Constructor
SkipList::SkipList(int maxLevel)
{
    this->maxLevel = maxLevel;
    level = 0;

    // create header node and initialize key to -1
    /*header*/
    head = new Node(-1);
    tail = new Node(999);
    endTail = new Node(999);



    for (int i = 0; i <= head->NodeLevel; i++)
    {
        head->next[i] = tail;
    }
    for (int i = 0; i <= tail->NodeLevel; i++)
    {
        tail->next[i] = endTail;
    }
};
```

Next, we create a Skiplist class which holds the instance of one SprayList data structure. We only deal with only one such instance. In its constructor, we initialize its max level that it can hold, holds the current level value in variable 'level'. Initializes the head and tail pointers. Now since the add() and find() method uses three pred, curr and succ, hence we need 3 sentinal nodes to start in case of an empty list, so we have another tail called 'endTail'. Key values for head and tail nodes are -1 and 999 respectively. No node can contain key value less than or greater than these respectively.

```
// create new node
Node* SkipList::createNode(int key, int level)
{
    Node *n = new Node(key, level);
    return n;
};

// Display skip list level wise
void SkipList::displayList()
{
    cout<<"\n*****Skip List*****"<<"\n";
    for (int i = 0; i <= level; i++)
    {
        Node *node = head;
        cout << "Level " << i << ": ";
        while (node != NULL)
        {
                cout << node->key <<" ";
```

```
                node = node->next[i];
        }
        cout << "\n";
    }
};
```

Now we move on to the general functions such as CreateNode() and displayList(). The CreateNode() function just create calls the regular node constructor. The displayList() displays the entire Spraylist starting from the lowest to highest level.

bool SkipList::find(int x, Node *preds[], Node *succs[]) {} is the function that returns true if node with key x is found and fills up the *pred[] and *succ[] with the predecessor ad succor at all levels of the node found. If the node is not found, then it returns the preds and succs of the largest element which is lower than x.

We start by initializing the startingHeight which is the variable starting height as described in the SprayList Algorithm. This height differs for different threads accessing it. Since we assign the k value a random number between 3 and 5 (based on intuition and less errors). Each thread gets a different value for starting height. Hence there is less contention for add and deleteMin operations which call find(). This was intentional by the authors to reduce the congestion at the starting nodes if all threads started at the same height. I tried this first, but with the change in the number of threads, it effected the log value and changed the level value in such a way that it caused segmentation fault with thread count higher than 7. Hence I had to use maxLevel.

We also descend the levels one by one for our implementation since the authors did the same for their implementation. However, the algorithm suggests changing the number of levels to more than 1 or random; at a time, such that there are no congestion between threads. This number is assigned to the variable ‘levelsToDescend’.

We assign the start traversing from the head (assigned to pred), curr becomes the next node to pred and succ next to curr; pred, curr and succ denote the predecessor, current and successor threads. Then we check if the next pointer of current node (i.e.succ node) is marked or not. If yes, we move to remove the marked nodes. As the paper explained, the find() method is a cleaner function that removes all the marked nodes physically while searching for an element. The DeleteMin() operation marks the minimum node and calls the find() function to clean it up. That is the function of find(). Now if the (curr->next[level])->mark is true, then we need to remove the curr node.

```
while (true)
{
int k = = rand() % 5 + 3;
int startingHeight = log (NUM_THREADS) + k;
int levelsToDescend = 1;
// In the paper, for implementation D = 1
pred = head;
for (int level = maxLevel/*startingHeight*/; level >=
bottomLevel;
level = level - levelsToDescend)
{
    curr = pred->next[level];
    while (true)
    {
        succ = curr->next[level];
        mark1 = (curr->next[level])->mark;
        if(level <= curr->NodeLevel &&
        level >= succ->NodeLevel)
            mark1 = true;
        while (mark1)
        {
            if((pred->next[level])->mark == false)
            {
        std::atomic<Node*> pnext(pred->next[level]);
        snip = pnext.compare_exchange_weak(curr, succ);
        pred->next[level] = pnext;
            }
            if (!snip)
                continue;
            curr = pred->next[level];
            succ = curr->next[level];
            mark1 = (curr->next[level])->mark;
        }
        if (curr->key < key)
        {
        int jumpLength = 1 * pow (log (NUM_THREADS),
3.0);
        /*std::random_device rd;
        std::mt19937 generator(rd());
        std::uniform_int_distribution<> distribution(0,
jumpLength);
        int jump = distribution(generator);*/
        int jump;
        if(level == 0)
            jump = 1;
        else
            jump = rand() % jumpLength + 1;
        for(int i = 0; i < jump; i++)
        {
            if(pred->key == 999 || curr->key == 999)
                break;
            pred = curr; curr = succ;
            succ = curr->next[level];
        }
```

```
        }       // End of IF 'if (curr->key < key)'
                else
                        break;
                }//End of 2nd while loop
                preds[level] = pred;
                succs[level] = curr;
        }
    return (curr->key == key);
    }//End of 1st while loop
};
```

Now, we check whether the pred->next[level] is marked or not, if it is the it means that some other thread marked the pred thread while performing this current operation. So, we assign the pred, curr and succ nodes again, and repeat the while loop. This ensures that if some other thread changes the pred node, we restart the operation and consistency is maintained. If pred->next[level] is not marked, we move on, to point pred's next node to succ. This is the physical deletion step of any node, where the curr node just got deleted. If successful, we assign the new pred, curr and succ nodes and repeat till we find a node with no mark. If unsuccessful, we skip the assignment and repeat the loop using the continue statement.

After the physical deletion step, we check to see if the curr node's key value is less than the desired node's key, if yes then we keep on traversing the list. In traditional SkipList, we just ahead only one node at a time. However, in SprayList we assign the jump length a random number so that we can avoid contention at the same level, if multiple threads compete. Since all threads start from the head node and with a jump length 1, they keep on contending for the same nodes, we change that by assigning different jump lengths. This was explained by the paper and one of the key parameters of the algorithm. However, if the jump causes the thread to skip or reach the tail, we break; if pred->key == 999 or curr->key == 999. After that we assign the pred and curr nodes of each level to the node array *preds[] and *currs[] that will help in the add() and DeleteMin() operations.

Now, we move on to the bool SkipList::add(int x) method:

We start by searching the list for a node with the same key as we're about to add. If it exists, we exit the method since we cannot have any duplicate elements. If it doesn't then we proceed. Note that here the find method returns the *preds[] and *currs[] which contain the future predecessor and successors nodes of different levels. This helps in adding the node. We start by creating a new node The topLevel variable is assigned a random integer for its levels. As we know that a SkipList is a randomized data structure that enables less congestion. This is how we build we random structure. Then we traverse all the levels and link the current nodes with the successor nodes first. This method sets its next references to the successors before it links the node into the bottomlevel list, meaning that a node is ready to be removed from the moment it is logically added to the list. The next step is to try to logically add the new node to the abstract set of the list by linking it into the bottom-level list. After assigning the next pointers to succs[], we move on to link the pred->next[] to the current node. Here we use the atomic compare_exchange_weak() method which checks if the pred->next[bottomLevel] still points to succ[bottomLevel] node at the bottom level. If yes then points it to the new node 'newNode'. If a following thread tries to do the same thing, then it is stopped since the value of pred->next[bottomLevel] has changed, and it fails. If so, then the continue statement starts the entire while loop again, and the addition operation is started from the start because of the old values the interrupting thread contains. This is all for the bottom level. Now we move on to link the rest of the pred->next[] nodes to the newNode. We apply the similar compare_exchange_weak(). If successful we move on to the next level, else we call the find() method again to update the preds[] and succs[] values; this shows that some other thread must have changed the preds[] or the succs[] nodes.

```
while (true)
{
        bool found = find(x, preds, succs);
        if (found)
        {
                return false;
                // Linearization Point for unsuccessful add()
        }
        else
        {
                Node *newNode = createNode(x, topLevel);
                key++;
                for (int level = bottomLevel;
                        level <= topLevel; level++)
                {
                        Node *succ;
                        succ = succs[level];
                        newNode->next[level] = succ;
                }
                Node *pred;
                pred = preds[bottomLevel];
                Node *succ;
```

```
                succ = succs[bottomLevel];
                newNode->next[bottomLevel] = succ;
    std::atomic<Node*> pnewnext(pred->next[bottomLevel]);

    if (!(pnewnext.compare_exchange_weak(succ, newNode)))
          continue;    // Linearization Point for Successful add()

                pred->next[bottomLevel] = pnewnext;
                if(topLevel > level) {
                        level = topLevel;
                }
                for (int level = bottomLevel + 1; level <=
topLevel; level++)
                {
                while (true)
                {
                        pred = preds[level];
                        succ = succs[level];
    std::atomic<Node*> pnewnext2(pred->next[level]);
                        if(
    pnewnext2.compare_exchange_weak(succ, newNode))
                        {
                        pred->next[level] = pnewnext2;
                        break;
                        }
                        find(x, preds, succs);
                }// End of while()
                }// End of for()
                return true;
          }
    }
```

Next is the DeleteMin() method. We start by assigning the head value to curr (for the bottom level). The purpose of the DeleteMin() is to remove the first element, which is right after head. So, this method tries to remove the first node which is not marked. It doesn't have to deal with levels. So, if the first node is not marked, we atomically set its flag to true. This flag set is done by the atomics library's std::atomic_flag_test_and_set() function. This sets the flag to true and returns the old value. So if the mark was false, it enters the if body and set the curr->next[0]->mark as true. I had to do this because, changing the std::atomic_flag curr_flag_test variable does not set the (curr->next[0])->mark. But we still maintain the linearizability since if some other thread follows right behind and tests the if statement, its atomic curr_flag_test will return true since the previous thread set to true atomically. So if std::atomic_flag_test_and_set() returns true, it won't enter the if body, and won't update its mark value. Hence no collision. After that, we just call the find() method to remove the marked node physically.

```
    curr = head->next[0];
    while (curr != tail)
    {
          if (!((curr->next[0])->mark))
          {
    std::atomic_flag curr_flag_test((curr->next[0])->mark);
          if(!(std::atomic_flag_test_and_set(&curr_flag_test)))
          {
                (curr->next[0])->mark = true;
    // Linearization Point for Successful DeleteMin()
                find(curr->key, preds, succs);
                return curr;
          }
          else
                curr = curr->next[0];
          }
    }
```

The code is uploaded in WebCourses along with this report.

V. Performance Tests

The graph with execution time with respect to number of threads is provided in the Zip file.
Number of threads = 5, Execution Time = 0.00260611 seconds
Number of threads = 10, Execution Time = 0.00470701 seconds
Number of threads = 15, Execution Time = 0.00287702 seconds
Number of threads = 20, Execution Time = 0.0107531 seconds
Number of threads = 25, Execution Time = 0.00402057 seconds
Number of threads = 30, Execution Time = 0.00469464 seconds
Number of threads = 35, Execution Time = 0.00491326 seconds
Number of threads = 40, Execution Time = 0.00996837 seconds
Number of threads = 50, Execution Time = 0.0139782 seconds

VI. Related Work

The primary reference for this paper provided is the 'The SprayList: A Scalable Relaxed Priority Queue' by Dan Alistarh, Justin Kopinsky, Jerry Li and Nir Shavit. My report is the implementation of this algorithm using C++ and POSIX threads in order to be compatible with the RSTM library.

The first concurrent SkipList was proposed by Pugh, while Lotan and Shavit were first to employ this data structure as a concurrent priority queue.

They also noticed that the original implementation is not linearizable, and added a timestamping mechanism for linearizability. Herlihy and Shavit give a lock-free version of this algorithm.

Sundell and Tsigas proposed a lock-free SkipList-based implementation which ensures linearizability by preventing threads from moving past a list element that has not been fully removed. Instead, concurrent threads help with the cleanup process. Unfortunately, all the above implementations suffer from very high contention under a standard workload, since threads are still all continuously competing for a handful of locations.

Recently, Lind´en and Jonsson presented an elegant design with the aim of reducing the bottleneck of deleting the minimal element. Their algorithm achieves a 30-80% improvement over previous SkipList-based proposals; however, due to high contention compare-and-swap operations, its throughput does not scale past 8 concurrent threads. To the best of our knowledge, this is a limitation of all known exact priority queue implementations.

Other recent work by Mendes employed elimination techniques to adapt to contention in an effort to extend scalability. Even still, their experiments do not show throughput scaling beyond 20 threads.

Another direction by Wimmer presents lock-free priority queues which allow the user to dynamically decrease the strength of the ordering for improved performance. In essence, the data structure is distributed over a set of places, which behave as exact priority queues. Threads are free to perform operations on a place as long as the ordering guarantees are not violated. Otherwise, the thread merges the state of the place to a global task list, ensuring that the relaxation semantics hold deterministically. The paper provides analytical bounds on the work wasted by their algorithm when executing a parallel instance of Dijkstra's algorithm, and benchmark the execution time and wasted work for running parallel Dijkstra on a set of random graphs. Intuitively, the above approach provides a tighter handle on the ordering semantics than ours, at the cost of higher synchronization cost. The relative performance of the two data structures depends on the specific application scenario, and on the workload.

REFERENCES

[1] Alistarh, D., Kopinsky, J., Li, J., & Shavit, N. (2015). The SprayList: A scalable relaxed priority queue. ACM SIGPLAN Notices, 50(8), 11-20.

[2] ' The Art of Multiprocessor Programming': by Maurice Herlihy, Nir Shavit

[3] 'Skip List | Set 2 (Insertion)': http://www.geeksforgeeks.org/skip-list-set-2-insertion/

[4] 'C++ Multithreading': https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm

[5] MULTI-THREADED PROGRAMMING III - C/C++ CLASS THREAD FOR PTHREADS – 2017: http://www.bogotobogo.com/cplusplus/multithreading_pthread.php

[6] Online C++ compiler: https://www.onlinegdb.com/

[7] std::atomic C++: http://en.cppreference.com/w/cpp/atomic/atomic

[8] std::atomic::compare_exchange_weak, std::atomic::compare_exchange_strong: http://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange

[9] 'POSIX thread (pthread) libraries': http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html