5-1-2017

Non-blocking Priority Queue based on Skiplists with Relaxed Semantics

Ashok Adhikari
University of Nevada, Las Vegas, ashokadhikari42@gmail.com

Repository Citation

Adhikari, Ashok, "Non-blocking Priority Queue based on Skiplists with Relaxed Semantics" (2017). *UNLV Theses, Dissertations, Professional Papers, and Capstones*. 2932.
https://digitalscholarship.unlv.edu/thesesdissertations/2932

NON-BLOCKING PRIORITY QUEUE BASED ON SKIPLISTS WITH RELAXED SEMANTICS

by

Ashok Adhikari

Masters in Computer Science (MS in CS)
University of Nevada, Las Vegas
2017

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2017

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

April 26, 2017

This thesis prepared by

Ashok Adhikari

entitled

Non-Blocking Priority Queue Based on Skiplists with Relaxed Semantics

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Ajoy K. Datta, Ph.D.
Examination Committee Chair

John Minor, Ph.D.
Examination Committee Member

Yoohwan Kim, Ph.D.
Examination Committee Member

Venkatesan Muthukumar, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

Abstract

Priority queues are data structures that store information in an orderly fashion. They are of tremendous importance because they are an integral part of many applications, like Dijkstras shortest path algorithm, MST algorithms, priority schedulers, and so on.

Since priority queues by nature have high contention on the `delete_min` operation, the design of an efficient priority queue should involve an intelligent choice of the data structure as well as relaxation bounds on the data structure. Lock-free data structures provide higher scalability as well as progress guarantee than a lock-based data structure. That is another factor to be considered in the priority queue design.

We present a relaxed non-blocking priority queue based on skiplists. We address all the design issues mentioned above in our priority queue. Use of skiplists allows multiple threads to concurrently access different parts of the skiplist quickly, whereas relaxing the priority queue `delete_min` operation distributes contention over the skiplist instead of just at the front. Furthermore, a non-blocking implementation guarantees that the system will make progress even when some process fails.

Our priority queue is internally composed of several priority queues, one for each thread and one shared priority queue common to all threads. Each thread selects the best value from its local priority queue and the shared priority queue and returns the value. In case a thread is unable to delete an item, it tries to spy items from other threads' local priority queues.

We experimentally and theoretically show the correctness of our data structure. We also compare the performance of our data structure with other variations like priority queues based on coarse-grained skiplists for both relaxed and non-relaxed semantics.

Acknowledgements

”I would like to express my sincerest gratitude to my thesis advisor, Dr. Ajoy K. Datta for his continuous guidance, encouragement and support throughout my work. I would like to take this opportunity to thank him for all the knowledge, both technical and non-technical, that he has bestowed upon me during the course of my study at UNLV. I plan to carry all his teachings and spread those to the world as I go.

I would also like to thank Dr. John Minor, Dr. Yoohwan Kim and Dr. Venkatesan Muthukumar for allocating their precious time in reviewing my work and providing valuable comments. I am grateful to have people of such stature in my thesis committee.

A special mention goes to my wife Mrs. Benju Koirala for her continuous support during my thesis work and my life. I cannot imagine being where I am without her. I am also thankful to my brother Mr. Ajit Adhikari and my sister Mrs. Ajita Adhikari. Both of them have been an important part of my life.

I am also grateful to all my family and friends who helped me directly or indirectly during the course of my thesis.

And finally, last but by no means the least, I would like to thank my mother Mrs. Ganga Adhikari. Her life has always been an inspiration for me. Her unfaltering love and support is what makes me want to move forward.”

Ashok Adhikari

University of Nevada, Las Vegas

May 2017

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Necessity of Multicore

According to Moores law, the density of transistors that fit into an integrated circuit board exponentially increases each year. Without overheating, the growing number of transistors cannot be packed into the same space. What this implies is that we can no longer rely on hardware manufacturers to produce fast processors to make our computation more efficient. The luxury that we had on increased speedup because of increased clock speed is now over. Hence, we must change the way we approach problems. Instead of relying on a single fast processor to do the work for us efficiently, we must approach the problem so that multiple processors can work on it simultaneously. Of course, there is an inherent complication in this approach because one must synchronize all the different processors acting on it, but it is the only way out that we have. Without exploiting multicores, it is practically not feasible to get more efficient computations.

To clarify the intent, lets look at this simple example:

Consider a simple program which takes as input a linked list of integers and outputs the sum of all the elements present. Lets suppose there are n elements in the linked list. Suppose the processor that we are using can iterate through the elements at a speed of m elements/unit time. If we only use a single processor to do this task, then the best time we can finish the task would be n/m unit time. Now if we throw more processors with the same speed at the problem, we can divide the work such that each processor works on an equally divided part of the linked list, and then we can do the job faster. For example, if we use two processors, such that the first processor works on the first half of the list and the second processor works on the second half, then each processor will take $(n/2m)$ time unit. Hence, we can complete the task in half the time as compare to the time taken by a single processor.

1.2 Necessity of Concurrent Data Structures

Although we used multiple processors to solve the task in the example presented in section (1.1), the synchronization that we needed to implement it was very low. In fact, we had to do nothing at all, apart from possibly waiting for all the processes to end, so that the main process can sum up all the values they computed before returning to the caller. These types of problems are called embarrassingly parallel problems. They are such problems where we do not have to take care of synchronization; hence we can simply throw more processors at the problem, and we will get the speedup. There is a catch here, however. If we throw so many processors such that the time taken to manage all the processes takes longer, then it may affect or even reduce the speedup.

If the problem at hand is not embarrassingly parallel, then the problem is much more difficult. Lets look at an example to clarify the intent. Consider a different problem than section (1.1. Suppose we now have a set of tasks with priorities. The problem now is to execute all the tasks in the priority queue only once, but with an added condition that the items executed must be ordered on their priorities. At first glance, a solution might be to use a heap data structure and insert all the tasks into the heap data structure first, before deleting and executing them. If we observe carefully, this solution will only work if we use a single processor. On more than one processor, we may execute a task more than once. One such scenario is when processor `p1` and `p2` both call `delete_min` simultaneously and read the value `item1` as the item with the lowest priority and try to execute it. This happens because `p1` does not know that `p2` was calling `delete_min` concurrently.

Hence, we need to redesign sequential data structures to make them work in a concurrent environment. These modified data structures are called concurrent data structures.

1.3 Necessity of Non-blocking data structures

There are several ways we can design a concurrent data structure. If we refer to the example presented in section (1.2), we can use coarse-grained locks on the heap. What this means is that we apply locks on every operation on the heap. Hence, each processor guarantees that it deletes a unique item. However, there is a problem with this solution as well. Consider a scenario where a thread holding a lock while calling `delete_min` crashes. Now the whole program will halt as a result. Hence, using locks to design parallel data structures, although it guarantees correctness, cannot guarantee progress. We call such data structures blocking data structures. To get rid of this problem, we need to design a data structure such that failure of a thread does not impact the overall progress of the system. Such data structures are called non-blocking data structures. There are several techniques for implementing non-blocking data structures. One thing common to all non-blocking data structures is that they use some atomic primitives like compare-and-swap, fetch-and-add, etc. to coordinate the processes, instead of locks.

1.4 Necessity of Relaxed data structures

Relaxing the semantics of a data structure allows the data structure to perform operations that may not align with the sequential specifications. For example, consider a priority queue with relaxation parameter k. Whenever a thread calls a `delete_min` operation on the priority queue, it is still considered correct if it returns the `kth` smallest item in the queue. Relaxing the data structure reduces contention spots which occur because of concurrent access by multiple processes. For example, for a priority queue implemented with either skiplists or log-structured merge trees, the contention spot during `delete_min` is distributed from the head of the list to the first `k` elements in case of skiplists or the head of the first `k` blocks in case of log-structured merge trees. Relaxing may not be an option when the requirements dictate higher ordering.

1.5 Why do we care about data structures so much?

Data structures define how we store and manage information in our system so that they can be retrieved/manipulated efficiently. Data structures are present in almost every software system or program. There are also many cases where a program or algorithm relies completely on a specific data structure. For example, a breadth first traversal of a graph requires a queue data structure to work correctly. Hence, data structures are of huge importance when it comes to solving problems using computers.

1.6 Contribution

The main contribution of this thesis work is a non-blocking priority queue implementation. We implemented the priority queue using skiplists and relaxed the `delete_min` operation by a constant `k`. What it means is that, whenever a thread calls a `delete_min` operation, the priority queue can delete any of `k` smallest elements from the queue. Further, the priority queue is internally composed of several priority queues - one for each process and a shared priority queue common to all of them. The shared priority queue has a higher ordering guarantee but higher contention, whereas the local priority queues have lower contention but a lower ordering guarantee. So, by mixing the two priority queues, and by altering values for relaxation constant `k`, we can get a priority queue with relatively low contention and a good ordering guarantee. Further, we also guarantee local ordering of items, which means that the order in which a thread inserts items into its local queue matches the order in which it deletes them.

1.7 Outline

In Chapter 2, we will give a background on various terminologies in shared memory systems which are required to understand the work presented. We discuss concurrent data structures and their properties. We also give an insight into various priority queue implementations like heaps, skiplists, etc.

In Chapter 3, we will present a literature review on the topic of concurrent data structures. We will talk about the recent work being done in and around the topic. We will discuss various blocking / non-blocking implementations of data structures like binary trees, heaps, tries, skiplists, log-structured merge trees, etc. We will focus more on priority queues, however. We will look at the current state of the art implementation of priority queues, analyze the data structures used and the performance as claimed by the authors.

In Chapter 4, we will dive into the implementation of our priority queue. We will discuss the high-level design of our algorithm as well as give pseudo-code for it. We will work through the algorithm in detail as well as highlight cases where synchronization is tricky.

In Chapter 5, we will show that our priority queue is correct theoretically. For theoretical correctness, we give proofs that our queue operations, i.e. insert and `delete_min`, are lock-free. For linearization, we only give the linearization points without proving them.

In Chapter 6, we compare the performance of our priority queue with other variations of priority queues like priority queue based on coarse-grained locking, priority queue based on shared skiplist and priority queue based on the combination of the two queues just mentioned.

In Chapter 7, we will give concluding remarks which include a summary of our work as well as talk a bit about future enhancements.

Chapter 2

Background

Technology nowadays is shifting from using uni-processors to multicores. The shift is not just a choice; it is a must. We can no longer expect processor manufacturers to produce processors with higher clock speed. Processor manufacturers will soon, as Moores Law predicts, be unable to pack more transistors into a single processor to increase speed without overheating. So, the only way to gain efficiency is using multicores.

According to [Ven11], a **multicore** processor is typically a single processor which contains several cores on a chip. Each core on a multicore consists of both a computational unit and a cache. Together, the multiple cores combine to replicate the performance of a uni-processor. A single core is not necessarily as potent as a single uniprocessor, but when multiple cores coordinate and do the task in parallel, they can usually outperform a uni-processor. Now, the key to multicore machines being faster than uni-processor machines can be understood by highlighting how the two types of processors execute programs. In the case of a single processor, the processor would assign a time-slice to each process sequentially. At any time, only one process will have the processor; hence if a process takes longer to complete, then all other processes start lagging. However, this is not the case in multicore systems. In multicore systems, each core is assigned multiple processes, hence at a single instant, various processes can be executed in parallel, thus boosting the performance.

2.1 Shared Memory System

There are various ways in which different processors working on a problem can interact. One of the methods is the use of shared memory.

Definition 2.1.1 *Shared Memory: A memory that may be simultaneously accessed by multiple* *programs with the intent to provide communication among themselves or avoid redundant copies.*

A system that uses shared memory as a communication bridge for different processors is a shared memory system. A shared memory system can be either of the following [ERAEB05]:

2.1.1 Uniform Memory Access (UMA)

If each shared memory location is equidistant from every processor, then such architecture is called UMA. Here, the access time of a memory location is independent of the processor requesting it.

2.1.2 Non-Uniform Memory Access (NUMA)

In the case of non-uniform memory access architecture, the distance between memory locations and processors may not be uniform. Hence, the memory access times for the same location by different processors may be different.

2.1.3 Cache-Only Memory Architecture (COMA)

In the case of UMA and NUMA, local memories associated with each processor are the actual main memory. However, in the case of COMA, local memories are used as cache. So for COMA, access to a remote data by a processor may cause the complete data to migrate to its cache. Although the migration of data reduces the number of redundant copies as compared to NUMA architecture, it puts forward other coherence problems such as, how to find a data if it migrated to someones cache, what strategy to apply if the local memory fills up, etc.

2.2 Cache Coherence Protocol

In shared memory systems, where each processor may have a separate cache, it is possible to have multiple copies of the same data in multiple cache locations, i.e. one copy of the data in the main memory and another copy in the local cache of each processor that requested the memory location. Hence, whenever the data is updated, the change must propagate properly in all the copies. The protocol that ensures this timely propagation of shared data in a timely fashion across the system is called a cache coherence protocol.

Definition 2.2.1 *Cache Coherence Protocol: A protocol that ensures a timely propagation of shared data in a timely fashion across the system is called a cache coherence protocol.*

There are various models and protocols for implementing cache coherence, some of which are MSI, MESI, MOSI, Synapse, Berkeley, etc.

2.3 Atomic Primitives

Atomic instructions are such instructions that can temporarily impede CPU interrupts, such that the currently running processor is not context switched. What this means is that, in the time frame of an atomic instruction, only one processor is in charge. Atomic primitives are one of the building blocks of non-blocking algorithms. In this section, we go through some of the most common and widely available primitives.

2.3.1 Compare and Swap (CAS)

CAS atomically attempts to update the value of a variable with a new value, only if the variable had an expected value stored in it. The following code clarifies the functionality of CAS:

Algorithm 1: Possible implementation of CAS operation.

```
Data: address, expected, newvalue
Result: boolean
if $address == expected$ then
    address = newvalue;
    return true;
else
    return false;
end
```

All the statements inside the function get executed in one atomic step. There are other flavors of CAS which act on multiple addresses at the same time. Specifically, a CAS that acts on two memory locations atomically is called Double Compare and Swap (DCAS). Use of such primitives can help implement certain data structures such as dequeues [ADF$^+$00] with relative ease, but it is not always the case. The algorithms are as complex and error-prone as they would be when implemented using just CAS. The reason such primitives are not in widespread use is that they have slow performance. Further, they are not natively supported in any of the widespread CPUs in production.

2.3.2 Test and Set (TAS)

TAS is an atomic instruction which writes 1 to the memory location specified and returns the old value. All the steps are performed in one atomic step. Following is a possible implementation of TAS:

Algorithm 2: Possible implementation of TAS operation.

```
  Data: address
  Result: boolean
1 initial = address;
2 address = true;
3 return initial;
```

All the statements inside the function are executed in one atomic step.

2.3.3 Fetch and Add (FAA)

2.3.3 Fetch and add (FAA) Also, called get-and-increment, this instruction adds given value to the contents of the memory location referenced. Following is its possible implementation:

Algorithm 3: Possible implementation of FAA operation.

```
  Data: address, value
  Result: old-value
1 initial = address;
2 address += value;
3 return initial;
```

All statements inside the function are executed in one atomic step.

2.3.4 Load-Linked / Store-Conditional (LL/SC)

A load-linked instruction, given an address to a memory location, returns the contents of the memory location. A store-conditional instruction comes paired with a load-linked instruction. If the contents of the memory location have not changed since a load-linked, a store conditional will store the new value to the new location and succeed. It will fail otherwise. An LL/SC pair is stronger than a read followed by a CAS because the latter succeeds even if the memory locations value has been changed and restored (ABA problem). LL/SC, however, will fail in this case.

2.4 Concurrent Data Structures

Like normal data structures, concurrent data structures' purpose is to store and manage data such that they can be efficiently accessed and manipulated. The main difference here is that they must work with multiple threads, which may access the data simultaneously. As such, a concurrent data structure must provide not just a safety property, as done by sequential versions, but also provide a liveness property. A safety property ensures that something bad never happens, whereas a liveness property ensures that something good keeps on happening.

A method is blocking if it causes some other thread, which has called the same method, to block until the thread currently calling the method is done. As such, lock-based implementations of concurrent data structures are blocking. Once a thread holds the lock on a method, other threads wanting to execute the method must wait for the thread to finish its execution and release the lock. There are two types of liveness properties associated with these sort of implementations:

Definition 2.4.1 *Deadlock-freedom: An implementation is deadlock-free if some thread trying to acquire the lock eventually succeeds.*

Definition 2.4.2 *Starvation-freedom: An implementation is starvation-free if every thread trying to acquire the lock eventually succeeds.*

In the case of non-blocking algorithms, liveness properties can be any of the following:

Definition 2.4.3 *Lock-freedom: An implementation is lock-free if some thread eventually makes progress.*

Definition 2.4.4 *Wait-freedom: An implementation is wait-free if all threads eventually make progress.*

Definition 2.4.5 *Obstruction-freedom: An implementation is obstruction-free if some thread allowed to run in isolation eventually makes progress.*

To summarize, deadlock freedom and lock-freedom guarantee system-wide progress, whereas starvation-freedom and wait-freedom guarantee per-thread progress.

There are various ways to analyze the correctness of a concurrent data structure. As described by Herlihy and Shavit in [HS08], we can use the following consistency notions to verify the correctness of a concurrent data structure:

Definition 2.4.6 *Sequential Consistency: A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each processor appear in this sequence in the order specified by its program [Lam79].*

Definition 2.4.7 *Quiescent Consistency: An execution of a concurrent program is quiescently consistent if the method calls can be correctly arranged retaining the mutual order of calls separated by quiescence, a period where no method is being called in any thread.*

Definition 2.4.8 *Linearizability: An execution history (sequence of method invocations and responses) is linearizable if the invocations and responses can be reordered to yield a correct sequential history, such that if a response preceded an invocation in the original history; it still precedes it in the reordered sequential history.*

2.5 Relaxed Data Structures

The semantics of a data structures are relaxed if some method calls in the data structure can return values that may not necessarily be in order as mandated by the original data structure. For example, a relaxed stack may pop an item that was not most recently added. For each relaxed data structure, there is a parameter that defines the relaxation, generally called a relaxation constant. This constant determines the bound that the operations can deviate from the normal behavior. Continuing with the above example, a relaxed stack with relaxation constant, say 2, can pop the last two recently added items.

2.6 Work Stealing and Work Spying

In multithreaded applications, where each processor is assigned a list of tasks, various schemes define the behavior of a process when it runs out of its task list. One such technique is work-stealing. In this scheme, whenever a processor runs out of its task list, it picks up a random processor and moves some items to its local list and then executes them. Another scheme, which is non-destructive i.e. it does not remove the items from the task list of the victim processor, is called work-spying [WCV$^+$13]. The only difference is that, in this scheme, the elements are just copied to the local task list; they are not moved. Work-spying schemes are applied in cases where local-ordering semantics need to be guaranteed.

2.7 Skiplists

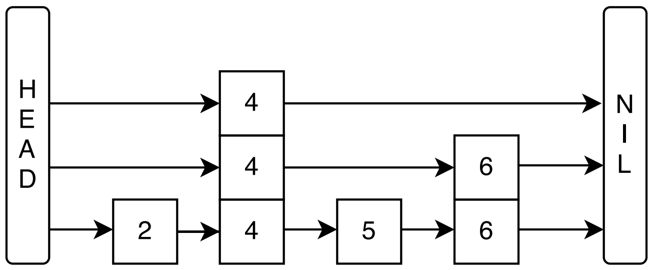

Figure 2.1: Structure of a skiplist.

Skiplists [Pug90] are probabilistic data structures that allow efficient searching within an ordered sequence of elements. A skiplist has two sentinel nodes: `head` and `NIL`. In between `head` and `NIL`, all the elements of the skiplist are stored. Each node has a level that is randomly chosen during its insertion. Some skiplists have a fixed maximum level defined, whereas other variants grow the maximum level dynamically. The bottom layer of the skiplist is a sorted linked list (can be doubly linked or singly linked). Each subsequent upper level serves as a shortcut into the lower level layer, hence find/insert/erase operations can be performed much faster than in a simple linked list. To search for an element in the skiplist, a thread traverses the skiplist from the maximum level. It can be shown that at each level, almost half the elements are skipped over, with high probability, hence the runtime for all the operations in the skiplist is, with high probability, `O(log n)`.

A node of the skiplist consists of the key, value and an array of forward pointers that point to other nodes in the skiplist. Following is a possible representation of a node in a skiplist:

Algorithm 4: Possible representation of a SkipNode.

```
Data: key
Data: value
Data: array of pointers to SkipNode
```

2.7.1 Predecessors and Successors

For each node, two arrays of pointers to nodes can be defined. One of them is predecessors, and the other is successors. Both are crucial in the implementation of all the methods of the skiplist, namely, find, insert, erase. For a node, ”predecessors” is the array of links that are incoming towards the node, whereas ”successors” is the array of links that are outgoing from the node. To give an example, in Fig. 2.1, the predecessor for node four would be `[head, head, 2]` and successor for it would be `[NIL, 6, 5]`. There are three items in the array because the level of node 4 is 3.

Chapter 3

Literature Review

The need for concurrent data structure arises from the fact that current technology trend is switching towards multicore computers. Sequential data structures such as stacks, queues, linked list, etc. do not work as they are in the concurrent environment. Since multiple processes may access the same part of the data structure simultaneously, race conditions may arise. These conditions need explicit addressing when designing such data structures.

Much research is being done in the area of concurrent data structures because they are at the heart of every algorithm. In this section, we will go through some important works done in the area in recent years.

3.1 Lock based Concurrent Data Structures

Locking is one of the widely used and simplest ways of creating a parallel data structure. If we just put locks on all the operations of a sequential data structure [HS08], we typically get a concurrent version of the data structure. For example, if we use locks for pops and pushes operations of a stack, we will have a concurrent version of the stack which can work with multiple numbers of processes safely. Although the concurrent version created this way produces correct results, it cannot scale with increased number of threads. The reason is that only one process will be accessing the data structure at a time, hence, rendering the data structure equivalent to a sequential one. A workaround to this problem is a fine-grained locking scheme.

Herlihy and Shavit in [HS08] show a different version of concurrent linked list implementations. They start off with a coarse-grained version as described above where they lock all the methods of the linked list i.e. add, remove, find with the same lock variable. They then go on to describe a finer grained lock whereby all the nodes in the linked list have an internal lock with them. Any process trying to do an add has to lock the new node, its predecessor, and successor on the list. Further, on remove, the process has to make sure it locks the node it is removing as well as the successor node. The key here is that the locks acquired by the processes must be in order, i.e., if one of the processes starts acquiring locks from the beginning of the list while the other starts from the end, then deadlock can occur. This technique of acquiring locks is called hand-over-hand locking [HS08]. They also present another optimistic scheme of implementing the non-blocking list. Here, instead of traversing the list by successively acquiring and releasing the locks on each node, a process traverses the list without caring about the locks. Once it gets to the desired node, it first validates that the node is present, and then only tries to acquire locks. One of the demerits of this method is that contains needs to acquire locks, which is unattractive. They also present another approach which lazily removes nodes from the list. The main concept here is that a deletion operation has two steps: logical deletion and physical deletion. In logical deletion, a thread sets a marked flag present in the node to true, meaning that it is no longer a member of the list. In physical deletion, a thread shifts the node's links to bypass the deleted node. They then move on to show a non-blocking implementation of a linked list, which we will discuss in section (3.2).

Lotan and Shavit in [SL00] present a priority queue based on skiplist. Both insert and delete operation on the priority queue use locks, but on a finer level. They use a similar technique of lazy synchronization presented in [HS08]. Insertions move upward whereas deletions move downwards. Upon insertion, a thread acquires locks only on the current level. As in lazy synchronization [HS08], three locks need to be acquired to safely insert the new node: current node, predecessor node, and successor node. Once a thread is done linking the new node at a level, it then moves upwards. For deletion, which works top-bottom, a thread acquires locks on the node to be deleted and its successor. Once done, it unlinks the node at that level. It then moves downwards until all the levels are exhausted. Only when a node is unlinked from the bottom level, is it considered to be not present in the skiplist. Now, the `delete_min` operation simply traverses the bottom level list of nodes and marks the first unmarked node as deleted. Then it uses the skiplist delete method to remove the node from the data structure. Starvation is possible in this implementation, but it is highly unlikely.

3.2 Lock-free / Wait-free Concurrent Data Structures

The necessity of non-blocking data structures arises from the fact that lock-based data structures cannot guarantee progress when a thread holding a lock crashes. Much research has been done on this very topic. Herlihy and Shavit present a non-blocking linked list in [HS08] where they use a concept of `AtomicMarkableReference`. It is a combination of a boolean mark flag and a reference. Setting of the mark flag and updating the reference can be done atomically. Languages like Java provide library functionality for such variables, but in languages like C/C++ where we can do pointer manipulations, we can simply use the last bit of the pointer as the mark bit. The necessity of using `AtomicMarkableReference` arises from the fact that, a lookup followed by a CAS while inserting on a linked list may cause lost inserts/deletes. The implementation technique follows closely to Lazy Synchronization [HS08], the only difference being, links are updated using CAS on `AtomicMarkableReference`, instead of using locks.

Crain et al. [CGR12] present a contention-friendly non-blocking skiplist that relies on CAS ensure non-blocking property. In the implementation, they divide insertions into two phases: abstract modification, meaning insertion at the bottom level, and structural adaption, meaning updating pointers at higher levels. Similarly, they divide deletions into two phases: logical deletion, meaning marking of nodes, and physical deletion, meaning actual switching of links and garbage collection. Links are switched using CAS during deletion. To avoid the lost update problem, they carry out physical removal in two steps. First, the nodes value is CASed from null to itself; then only it is deleted. Further, they mark the node's next field. Now, other inserting threads ensure that no one else has marked their prev and next nodes before doing the insertion operation. Instead of using a separate "marked" field for each node, the data structure uses null marker for logical deletion. A thread nullifies a node's value reference to delete the node logically. In the algorithm, structural adaptation may happen only after many abstract modifications. Such delay in structural adaptation hinders logarithmic time complexity of operations, but as it only happens under contention, performance is not affected. Delaying structural adaptation has one interesting advantage. If a deleted node is added again before structural adaptation happens, insertion can simply unmark the logically deleted node to make it alive. Subsequent structural adaptation does not need to do work on the node. One another interesting property of the skiplist is worth mentioning: a thread only deletes the bottom level of any node while deleting it, it leaves the higher levels as they are. It is only when the deleted nodes' count reaches some threshold that the thread wipes out the bottom level index list. Apart from this, there is no other mechanism to remove the index level nodes.

Liu et al. show an implementation of a non-blocking hash table [LZS14], which is both lock-free and wait-free. The hash table they presented is resizable, dynamic and unbounded. They use freezable set objects [LZS14] as building blocks of the hash table. To copy the keys from old buckets of the hash table to the new one during the resize operation, each bucket is first frozen. Freezing ensures that the logical state of a bucket stays the same during the transfer. They also present a specific lock-free freezable set implementation using CAS. The wait-free version of the implementation uses helping mechanism, which is similar to the doorway stage of Lamports Bakery algorithm. An array is used to announce tasks. Each task is assigned a priority using a strictly increasing counter variable. Once a task is assigned to the announce array, a thread scans the announce array to see if there is any task with higher priority than the one it has. If it finds one, it helps it first.

Jayanti and Tarjan present a non-blocking algorithm for disjoint set union [JT16]. They again us CAS operation in the "unite" operation when merging two trees to one. They also present three different implementations of find method using three separate path compression techniques: compression, splitting, and halving. Compression replaces the parent of every node on the path by the root of the current tree; splitting replaces the parent of every node on the path by its grandparent and halving does splitting on every other node on the path.

Ellen et al. present a non-blocking binary search tree in [EFHR14]. They start the paper by showing their previous implementation of the same problem which had poor amortized time complexity. There were three main problems with their previous implementation. Firstly, whenever an update operation failed, search restarted from the beginning. Secondly, traversals may pass through marked nodes multiple times because no helping mechanism was present. Thirdly, helping was recursive, which was inefficient. In their new implementation, they address all the above problems. Instead of starting from the root in case of failed update operation, each operation backtracks up the tree, until it finds an unmarked node. For the second problem, each marked node is helped first, ensuring they will not be traversed again unnecessarily. Finally, they omit the recursive helping business, as it turns out that it is unnecessary. Every update on the tree nodes is done atomically using CASes.

Kallimanis and Kanellou propose wait-free graph objects with dynamic traversals [KK16]. Their main contribution is a graph model with addition and removal functionalities of any edge of the graph, and atomic traversal of a part or the entirety of the graph. They also present a library called

Dense, which provides the functionalities mentioned above. The main idea behind the wait-free implementation is the use of two arrays, namely `done[]` and `ann[]`. Each process announces its operation to the announce array by marking its position in the array to true. Once a process completes an operation, it marks its location in the done[] array by setting it to true. If `done[p] == ann[p]`, it implies that an operation by p has completed. Otherwise, it is pending. Each process alternates between two phases `AGREE` and `APPLY`. In `AGREE` phase, processes detect pending operations using the `ann[]` and `done[]` arrays, whereas, in `APPLY` phase, the pending operation is carried out. The implementation uses LL/SC primitives.

Shafiei presents a non-blocking doubly-linked list in [Sha14]. In the algorithm, the list is accessed using cursors. The data structure supports cursor movements like move-left and move-right. A thread can traverse the list and land in the desired position to start updating the list by using a cursor. An interesting technique given in the paper is the way helping is carried out. Each node in the list has a descriptor field. If the descriptor for a node is null, it implies that no thread is operating on the node. If a node has a descriptor attached to it, then we know that some thread is working on the node. For example, if a node has a descriptor attached with information `delete`, this means that some thread is trying to remove the node. Once some thread, while traversing the list, finds that some node has a descriptor attached to it, it helps complete the nodes operation first, before moving on. The implementation only uses CASes as atomic primitive.

Petrank and Timnath show an implementation of lock-free data structure iterators in [PT13]. Their main contribution is a technique for a wait-free iterator for both lock-free and wait-free implementations of a set. To implement an iterator, they first obtain a consistent snapshot of the data structure. At the heart of the algorithm is the snap-collector object. A snap-collector object contains a list of node pointers and a list of reports. Whenever any process wants a snapshot, it creates an iterator and calls its `TakeSnapshot` method. The `TakeSnapshot` method, in turn, acquires a snap-collector object and then uses the object to capture snapshot by traversing the graph and adding unmarked nodes on its way. Finally, the iterator reconstructs the snap-collector data, and the process gets a consistent snapshot over which it can iterate. While a snapshot collection is ongoing, and if another process requests a snapshot, then the same snap-collector object is returned, instead of creating one. If no thread has currently allocated a snap-collector object, then a new instance is created, and the reference to the object is stored in the iterator atomically using CAS.

3.3 Work Stealing / Work Spying Concurrent Data Structures

Work stealing is a load balancing technique used in the context of multiple processes, where each process maintains a local set of executable tasks. The owner takes items from its task list and executes them. Once an owner is done with all of its tasks, to keep itself busy, it tries to steal tasks from other processes task list. There has been much research going on in the field of work-stealing. Many authors have published papers proposing various work-stealing algorithms, some of which we will discuss in this section.

Michael et at. propose idempotent work stealing algorithm in [MVS09]. They present three variations of their algorithm. First one is a LIFO work-stealing algorithm, where tasks are always extracted from the tail of the queue and inserted from the head. The second implementation is a FIFO work-stealing algorithm, where tasks are both inserted and extracted from the head of the queue. Finally, the third algorithm is a double-ended work-stealing algorithm. In the algorithm, owner thread extracts items from the tail of the queue, whereas the stealing threads extract items from the head of the queue. The algorithm presented exploits the relaxed semantics of the problems: the algorithm guarantees that each inserted task is extracted at least once. The algorithm works correctly only in problems where the application ensures that each task is executed only once, e.g. by a mechanism that checks the completion of a task before executing, or in problems where repeated tasks are allowed. It turns out that the criteria mentioned above apply to most of the problems. The authors try to minimize the use of atomic instructions to improve the performance of the algorithms. Each thread has its local queue with put, take, steal and expand methods. Only the owner is allowed to call put, take and expand methods; it is the thieves that call steal method when they run out of their tasks. It is only the steal method in their algorithm that uses CAS. The lower number of CASes is because their algorithm heavily relies on ordering writes.

Wimmer et al. provide another technique similar to work-stealing, called work spying [WCV$^+$13]. Work spying is closely related to work stealing in that it copies tasks from the victim list instead of moving them. This technique apart from simplifying synchronization helps maintain local ordering.

3.4 Relaxed Concurrent Data Structures

Relaxed data structures allow some deviation in the standard semantics of a data structure. For example, a relaxed priority work-scheduler can return an item more than once and still be considered correct; a relaxed stack can pop `kth` most recently pushed item and still be considered correct. The motivation behind relaxing the semantics of a data structure is to make it more favorable for a concurrent environment. Relaxation sometimes spreads contention spots, which otherwise would be a single spot in the conventional data structure, and sometimes reduces synchronization overhead. Research on how a data structure behaves under relaxation has been done these days [[WGTT15], [WCV+13], [MVS09]] massively.

Michael et al. present a work-stealing algorithm [MVS09] exploiting relaxed semantics. In the algorithm, every thread maintains its task list, which can be viewed by other threads to steal tasks. A standard semantics of the problem would mandate that each item in the task list is extracted only once, but in the implementation presented by the authors, they relax the semantics and allow multiple extractions of the same item by different threads. Relaxation works because the program using this algorithm can externally check for the completion of the work before executing it.

Alistarh et at. present a scalable relaxed priority queue based on skiplists called SprayList [AKLS15]. The main idea behind SprayList is that `delete_min` operation starts at a randomly chosen height `h`, instead of traversing the bottom level list. It then moves forward skipping few nodes. The number of nodes that it skips horizontally at a level is also determined randomly. Once done, it then moves to a lower level and continues the above process, until it reaches the bottom level. Upon reaching the bottom level, it checks to see if the node is available for deletion or not. If it is, then the operation returns the value, and it completes; otherwise, it retries the Spray operation just mentioned or becomes a clear thread and traverses the bottom level list linearly until it finds an unmarked node. The algorithm hence relaxes the `delete_min` operation to allow removal of first $O(plog^3p)$ highest priority elements, where p is the number of processes concurrently accessing the SkipList. The advantage of relaxing the `delete_min` operation is that the algorithm spreads the contention spot across the first $O(plog^3p)$ items and h levels instead of just the bottom level list and its first item.

Wimmer et al. show a lock-free k-LSM relaxed priority queue in [WGTT15]. The priority queue is composed up of two internal priority queues: one local to a thread, other global to all threads. Each priority queue uses log-structured merge trees internally. In their implementation, both the insert and delete operations are relaxed. For inserts, each thread can hold up to `k`-insert operations, before that are made globally available. For `delete_min`, it is allowed to return any of `k+1` smallest key visible to all threads.

Byrnes et al. propose a `k`-relaxed lateness queue implementation in [AB]. The queue is closely related to a FIFO queue, but the dequeue operation is relaxed such that, it does not need to remove the head for every dequeue operation, but it must delete it in no more than `k` dequeues. Each process maintains the count of elements dequeued since the head was last dequeued. If the count is less than a relaxation threshold, the dequeue operation takes a fast route and simply returns an arbitrary available element in the local queue. On the other hand, if the number of items dequeued since the dequeue of last head is greater than the relaxation threshold, then a slow dequeue is called, where it dequeues the head, and resets the count to zero.

3.5 Transactional Memory

Transactional Memory (TM) allows atomic execution of any arbitrary operation. Transactional memory has been a hot topic of research because the use of TM makes parallel programming easier.

Timnat et al. describe a transactional memory interface in [THP15]. The interface provided MCMS operation, an abbreviated form of Multiple Compare Multiple Swap. The operation is atomic and different from MCAS (Multi-word Compare And Swap) only in that it allows addresses to be compared without being swapped. The implementation of MCMS uses Hardware Transactional Memory (HTM) whenever it is present in the system, but in case HTM is not present, which is more common, it uses multiple CASes to do the job. The authors also present an implementation of two data structures namely, linked lists and binary search trees using the TM interface they proposed, to show its correctness.

3.6 Skiplists / Priority Queues

Skiplists have gained popularity because they are easy to implement and are favorable for concurrent accesses. They are a nice alternative to balanced binary search trees because they provide a logarithmic time insertion, deletion and search functionalities.

Priority queues are another data structures that many problems including priority work-schedulers, shortest path algorithms, minimum spanning tree algorithms, and so on use. As many algorithms use them as a core data structure, researchers have focused on efficient implementation of priority queues. Priority queues can be implemented using various data structures like: heap, sorted lists, skiplists [[HS08], [LJ13], [SL00], [CMH14], [AKLS15]], log-structured merge trees [WGTT15], etc. We now take a look at some priority queue implementations proposed by various authors in recent years.

Wimmer et al. propose a priority queue based on log-structured merge trees [WGTT15] with `k`-relaxed semantics. Their priority queue is internally composed up of multiple numbers of priority queues. Each thread owns one priority queue each, called local priority queue. There is also another priority queue that is visible to all the threads, called a shared priority queue. Both insert and delete operations are relaxed. Upon a call to a `delete_min` operation, a thread selects the best result from its local queue and the shared queue before returning the value. If a thread finds out that both its local queue and the shared queue are empty, then it tries to spy [WCV+13] items from other threads local task list.

Alistarh et al. propose a scalable relaxed priority queue based on skiplists called SprayList [AKLS15]. It also has relaxed semantics, meaning every `delete_min` operation is allowed to remove one of $O(plog^3p)$ highest priority elements in the list, where `p` is the number of processes currently accessing the skiplist. The algorithm spreads the contention at the head of the skiplist by starting a `delete_min` operation at a randomly chosen height `h`. The thread then moves forward at a particular level for a random number of nodes before descending one level. The thread continues the above operation until it reaches the lowest level. Once at the lowest level, the thread tries to delete the item. If it succeeds, the operation is complete, whereas if the operation is unsuccessful, it either retries the whole operation or it turns to a cleaner thread, where it traverses the bottom level list cleaning the marked nodes until it finds an unmarked node.

Calciu et al. present an adaptive priority queue with elimination and combining. The presented data structure is composed up of three layers. An elimination layer is at the front of the data structure. It serves the purpose of matching inserts and `delete_min` operations. A dedicated server thread collects the unmatched inserts and moves them into the second layer - the sequential part. The sequential part consists of items with lower keys, and the concurrent threads do not traverse it. However, it serves as a source of values to pending `delete_min` operations in the elimination layer. It is only when the sequential part exceeds in size by a predefined threshold, the server thread merges it onto the third part - the parallel part. It is the parallel part where the higher keys in the data structure are present. It is accessed by multiple threads and uses locks to obtain synchronization.

Linden and Jonsson propose a priority queue based on skiplists [LJ13], with less memory contention. One of the main technique described in this algorithm is the concept of batch deletions. A thread performing a `delete_min` operation only logically deletes a node; physical deletion is done by another restructuring thread only when a threshold number of nodes are marked. The algorithm proposed maintains two properties which are essential to the correctness of the algorithm. First, each node maintains its deletion mark along with the "reference" of its preceding node, and not the node itself. This ensures that the deleted nodes form a prefix of the skiplist, hence a batch deletion is possible. Second, the list contains at least one logically deleted node. An exception to this is at the beginning when no one has yet deleted a node. This guarantee inserts to be correct. Since nodes are not immediately physically removed from the skiplist, one can argue that it compromises performance because threads must traverse marked nodes multiple times. However, this is not the case, since reads are very cheap. It is instead the reads conflicting with concurrent physical deletions that hinder the performance. As with non-blocking skiplist implementation in [HS08], insertions move upwards, whereas deletions move downwards. The algorithm only uses CAS to synchronize multiple threads. The restructure operation, which a thread calls when the number of marked nodes in the skiplist crosses a threshold value, simply moves the head past all the marked nodes to point to the first unmarked node, at each level. Since the deleted nodes are guaranteed to form a prefix of the skiplist, this operation yields a correct result.

Wimmer et al. presented three different data structures for task-based priority scheduling in [WCV+13]. The first one uses work-stealing scheme [WCV+13], where processes try to steal tasks from other processes task list when they run out of tasks. The synchronization overhead needed in this case is tiny because different processes only interact with each other only when they need to steal tasks from others. The second one is a centralized `k`-priority data structure. The data structure is shared by all the processes. Hence, it requires more synchronization overhead. The priority queue offers global ordering of tasks on all the tasks present in the system. The ordering problem of the work-stealing priority queue is solved by the centralized implementation but at the cost of higher contention. Finally, the authors present a hybrid model of a priority queue, which uses both work-stealing and centralized priority queue, to obtain a queue with a higher guarantee and lower contention, which can again be controlled by a variable at runtime.

Chapter 4

Implementation

Skiplists [Pug90] were first described by W. Pugh. They are an alternative to balanced search trees because they provide logarithmic time insertion, deletion and search operations, with high probability. A skiplist is composed of a sorted linked list, which comprises the bottom level of the list. Each subsequent upward level acts as a shortcut to the elements in the bottom level. We first look at how various operations are carried out in a sequential skiplist.

4.1 Sequential Skiplist

4.1.1 insert

An insert operation expects a `(key, value)` pair as its input and adds an item with priority `key` and information `value` to the skiplist. If an item with key `key` is already present in the list, the operation updates the item's value.

Algorithm 5: Implementation of `insert` operation on a sequential skiplist.

```
   Data: key, value
   Result: None
1  found, preds, succs = find(key);
2  if found then
3      succs[0].value = value;
4      return;
5  end
6  height = generate_random_height(MAXLEVEL);
7  new_node = allocate_new_node(key, value, height);
8  foreach level from 0 to height-1 do
9      new_node.forward[level] = preds.forward[level];
10     preds.forward[level] = new-node;
11 end
```

As shown in the listing (5), an insert first checks to see if any item with the same key is present in the skiplist (line 2). If such an item exists, which is present in `succs[0]`, it simply replaces its value with new value `value`, and the operation returns. In case no item with key `key` is present in the skiplist, it allocates a new node with given `key`, `value` and a randomly determined `height` (lines 6-7). It then links the node to the skiplist using the `preds` values computed using the find method (lines 8-10).

Fig. (4.1) illustrates the insertion of a node with *key* 5 on the skiplist. The dashed lines show nodes that were unlinked during the insertion and the red lines show the newly added links.

4.1.2 delete

A delete operation expects `key` of the node to be deleted. If such an item with key `key` is present in the skiplist, the operation removes it and returns `true`. Otherwise, the operation simply returns `false`. As shown in the listing (6), the algorithm first calls a find operation to compute `preds` and `succs` of the node with key `key`. If it is not present, there is nothing to delete; hence it returns `false` (line 3). If such a node is present, which will reside in `succs[0]` (line 5), the operation switches its forward references from its `preds` to its `succs` (lines 6-8).

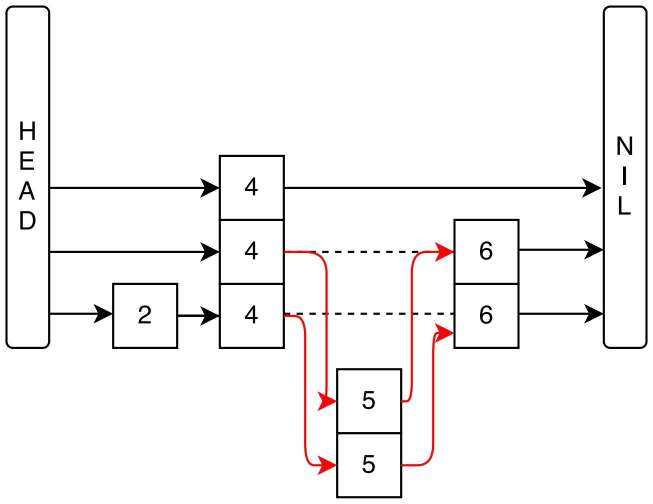

Figure 4.1: Insertion of node with *key* 5 in a skiplist.

Algorithm 6: Implementation of `delete` operation on a sequential skiplist.

```
Data: key
Result: bool
1 found, preds, succs = find(key);
2 if !found then
3     return false;
4 end
5 node_to_delete = succs[0];
6 foreach level from 0 to node_to_delete.height-1 do
7     preds[level].forward[level] = node_to_delete.forward[level];
8     preds.forward[level] = new_node;
9 end
```

Fig. (4.2) illustrates the deletion of a node with *key* 5 on the skiplist. The dashed lines show nodes that were unlinked during the deletion and the red lines show the newly added links.

4.1.3 find

A find operation, given a `key`, computes the predecessor and successor values for a node with key `key`. It returns `true`, if it successfully finds a node with key `key`. Otherwise, it returns `false`. The operation starts at the `MAXLEVEL` of the skiplist. Until it finds a node with a key greater than `key`,

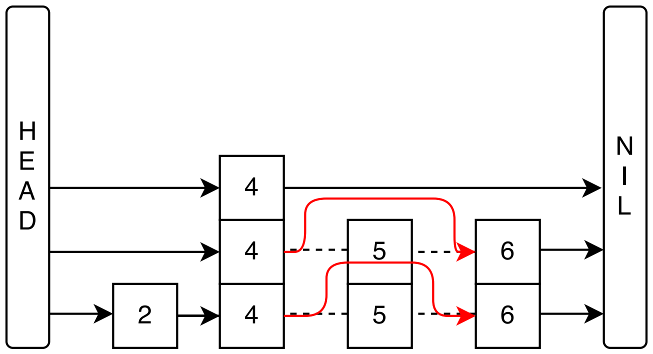

Figure 4.2: Deletion of node with *key* 5 in a skiplist.

it traverses the skiplist forward at the current `level` (lines 4-12). After that, it descends a `level` and continues the same operation until it reaches the bottom level. The size of both `preds` and `succs` arrays is equal to `MAXHEIGHT` of the skiplist. Listing (7) shows the details of a find operation.

Algorithm 7: Implementation of `find` operation on a sequential skiplist.

```
   Data: key, preds, succs
   Result: bool, preds, succs
1  level = MAXLEVEL - 1;
2  while level >= 0 do
3      cur = pred.forward[level];
4      while true do
5          if cur.key < key then
6              pred = cur;
7              cur = succ;
8          end
9          else
10             break;
11         end
12     end
13     preds[level] = pred;
14     succs[level] = cur;
15 end
16 return cur.key == key;
```

Since skiplists store data in sorted order, they are naturally suited for implementing priority queues. Further, as skiplists have multiple links to reach to the same node, multiple threads accessing the list will incur lower contention. Hence, they are one of the popular choices [[WGTT15], [LJ13], [SL00], [CMH14], [AKLS15]] for implementation of a priority queue. As skiplists are easy to implement, provide logarithmic time insertion, deletion and search operations, and perform better in a concurrent environment, we based all the priority queues on skiplists.

4.2 Distributed Priority Queue

Our distributed priority queue is internally composed of skiplists. Each thread that accesses the priority queue gets an unshared local priority queue. Each thread inserts elements into its local priority queue and deletes elements from the same queue unless it runs out of elements. Once a thread runs out of elements in its local priority queue, it selects a random thread from the available threads to `spy`. The main advantage of a distributed priority queue is that it has very low synchronization overhead as well as low contention. The only synchronization needed is during the spy operation, which we will discuss in section (4.2.5). The main disadvantage of using a distributed priority queue is that we do not have any global ordering guarantees associated with it. Since a priority queue is all about the removal of lowest priority elements, a distributed priority queue does not serve the purpose if used solely. It, however, has a local ordering guarantee, i.e. the order in which a thread inserts elements matches the order in which it removes them. Our distributed priority queue is similar to the one proposed by Wimmer et al. in [WGTT15], but our main contribution is that we use skiplists instead of log-structured merge trees in our implementation. Before going through the operations of the distributed priority queue in detail, let's revisit the `SkipNode` structure (section (4)) as it requires a small change to work in a concurrent environment.

4.2.1 Item

We first define an item structure and wrap the key, value pair with it. We additionally add an atomic boolean variable called `taken` in it. Since our implementation does not use locks, concurrent threads try to set the boolean `taken` flag atomically to logically delete the item, and hence the SkipNode.

```
Algorithm 8: Definition of an Item for distributed priority queue.
    Data: key
    Data: value
    Data: atomic<bool> taken
```

4.2.2 SkipNode

A SkipNode is composed of an `Item` (listing (8)) and an array of pointers to other SkipNodes. Listing (9) shows the definition of a SkipNode used in a distributed priority queue.

```
Algorithm 9: Definition of a SkipNode for distributed priority queue.
    Data: pointer to an Item
    Data: array of pointers to SkipNodes
```

It is also important to note that we maintain a separate list that contains all the items in a SkipList. SkipNodes simply point to entries in the item list. Figure (4.3) illustrates the idea. A red colored dot on the bottom-right represents a marked item, and hence a marked SkipNode.

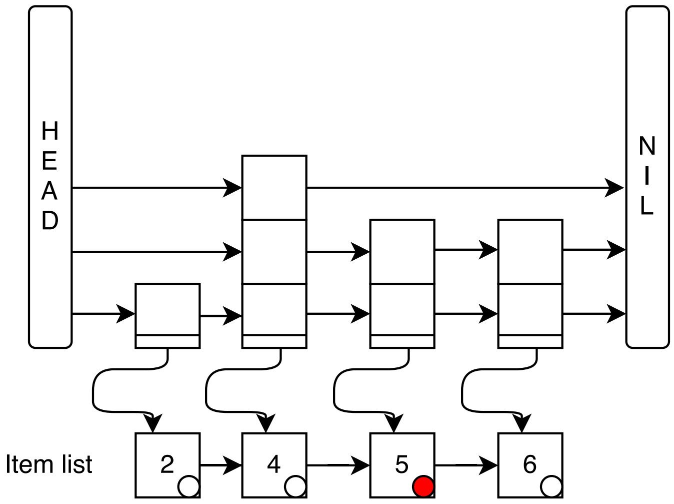

Figure 4.3: Representation of Items and SkipNodes in a distributed priority queue.

4.2.3 insert

Since the distributed priority queue uses skiplists internally, a call to insert on the distributed priority queue calls the underlying skiplist's insert method. Listing (6) shows the details of the implementation of an insert method on a distributed priority queue.

Algorithm 10: Implementation of `insert` operation on a distributed priority queue with shared priority queue as a parameter.

```
   Data: key, value, shared_pq
   Result: None
1  if shared_pq != nullptr and local_pq.size() > k then
2      shared_pq.insert(key, value);
3  end
4  else
5      insert(key, value);
6  end
```

If a shared priority queue is provided as a parameter to the insert method, and if the size of the local queue is greater than a threshold `k`, then the insert operation tries to insert the element into the shared priority queue instead (lines 1-3). However, if no shared priority queue is provided to the insert method, then the element is inserted into the local priority queue as shown in listing (5). Since our priority queue allows keys with duplicate priorities, the check for whether an element with the same key exists in the queue is not done, i.e. the insert method does not use lines 2-4 in listing (5).

4.2.4 delete-min

A `delete_min` operation only logically deletes the first node with an unmarked item in the bottom level list. Physical deletion is left to a subsequent find method.

Algorithm 11: Implementation of `delete_min` operation on a distributed priority queue.

```
   Data: parent_dist_pq
   Result: boolean, value
1  item = find_min(value);
2  if item == nullptr then
3      if spy(parent_dist_pq) > 0 then
4          item = find_min(value);
5      end
6      else
7          return false;
8      end
9  end
10 if item == nullptr then
11     return false
12 end
13 success = TAS(item.taken);
14 find(item.key, preds, succs);
15 return success
```

A `delete_min` operation calls a `find_min` operation internally. A `find_min` operation starts at the head of the skiplist and traverses the bottom level list until it finds a node with an unmarked item. It returns the node without deleting it. If a `find_min` operation fails to find an unmarked item, it returns a `nullptr`. In this case, the `delete_min` operation tries to `spy` on other threads and copy their data onto its local queue. If the `spy` operation is successful, the `delete_min` operation tries `find_min` one more time. If `find_min` returns a `nullptr` again, then the operation returns false, meaning it failed to delete an item. However, if an item is successfully peeked using `find_min`, the thread tries to set the node's taken flag to true atomically. If it succeeds, the operation succeeds and returns the value associated with the item; otherwise, the operation fails. The call to `find_min` in line 14 is an optimizing call. As a side-effect the find method will physically delete the node from the skiplist.

4.2.5 spy

A spy operation, given a distributed priority queue, simply copies all the unmarked items from the distributed queue's item list to its local item list.

Algorithm 12: Implementation of `spy` operation on a distributed priority queue.

```
Data: dist_pq
Result: int
victim = dist_pq.get_random_victim();
count = 0;
foreach item in victim.task_list do
    if item and !item.taken then
        insert(item);
        count++;
    end
end
return count
```

One should note that a `spy` operation does not copy items from the victim's task list. The operation simply points to the item in the victim's task list. We, however, allocate a new SkipNode while inserting a spied item to the local list. The SkipNode will have a reference to the spied item. As depicted in Figure (4.4), the `spy` operation only considers unmarked items. One should note that multiple threads can spy the same item, and the thread which successfully sets the `taken` flag to true returns the value.

For space efficiency, we can allow threads to `spy` only up to `k` elements from the victim list instead of all the available items.

4.2.6 find

A find method is similar to that of a sequential skiplist (listing (7)). The difference here is that, whenever it finds a marked node on its way, it physically deletes them. Since the owner thread is the only one who can delete nodes on the skiplist, the ABA problem does not occur.

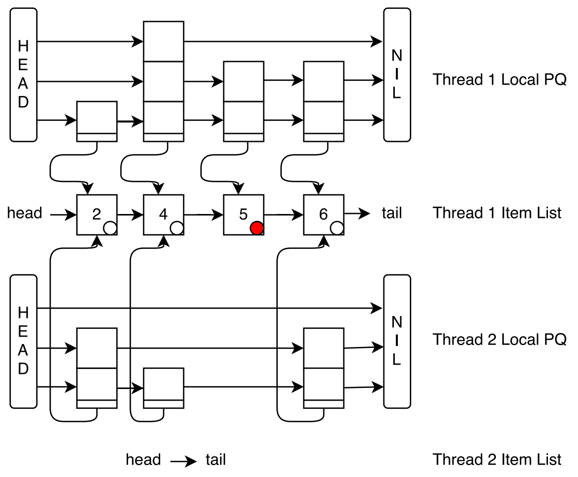

Figure 4.4: Thread 2 spying items from Thread 1.

Algorithm 13: Implementation of find operation on a distributed skiplist.

```
   Data: key, preds, succs
   Result: bool, preds, succs
1  level = MAXLEVEL - 1;
2  while level >= 0 do
3      cur = pred.forward[level];
4      while true do
5          succ = cur.forward[level];
6          while cur.item.taken do
7              Physically delete cur, and move forward.
8          end
9          if cur.key < key then
10             Move forward i.e.  pred=cur and cur=succ. Break the loop otherwise.
11         end
12     end
13     preds[level] = pred;
14     succs[level] = cur;
15 end
16 return cur.key == key;
```

In listing (13), lines 6-11 implement the physical removal of a node found during traversal.

4.3 Shared Priority Queue

The main idea behind a shared priority queue is that it provides a strong guarantee on the global ordering of the items deleted. As a result, it incurs higher contention. Our implementation of a shared priority queue is based on the skiplist proposed by Herlihy and Shavit in [HS08], with few changes. Some properties of the skiplist they proposed are as follows:

1. The skiplist is non-blocking and only uses `CAS`.
2. Insertion moves upwards while deletion moves downwards.
3. Deletion is done in two phases: logical deletion and physical deletion.
4. All forward references are `AtomicMarkableReferences`. A thread atomically marks the forward references of a node at a level to delete it logically at that level.
5. The skiplist does not store elements with duplicate keys.

Following are the contributions that we made in our work:

1. We changed the skiplist to allow duplicates.
2. We built a relaxed priority queue on top of the skiplist implementation.
3. We changed the `SkipNode` structure so that it pictorially looks like the one in Figure (4.5)(b). The figure shows a skipnode with `item.key` 2 and `level` 3. The bottom right red dot on a skipnode indicates a marked item. A white dot instead would indicate an unmarked item. This mark is used by a `delete_min` operation to claim the node for deletion. The dots connected with the forward references of the node which indicate the mark of the references, however, is the concept borrowed from [HS08]. If the dot is red, then the reference is marked; if the dot is white, the reference is unmarked.

In this section, we will only go through the changes that we proposed on top of the implementation given in [HS08].

4.3.1 Allowing Duplicates

To allow duplicates, we simply remove the check for whether a node exists in the skiplist or not before doing an insertion. The deletion proposed in [HS08] however, needs a little more attention.

As proposed by Herlihy and Shavit [HS08], the delete operation first searches for a node with the given key in the skiplist, then logically deletes it by marking all of its forward references. A subsequent find call then deletes the node physically from the list. Figure (4.5) illustrates the problem of this approach when duplicates are allowed. Consider a find-min operation traversing the skiplist. Suppose it successfully marks the node with key 4 and level 2. Now, to prepare the node for physical deletion, if we call delete(4), then as shown by the red arrow, the operation will incorrectly choose the node with key 4 and level 3 and start marking its forward references. What this means is that the delete physically removes the node with key 4 and level 3 without ever returning it from the list. To remedy this problem, we change the signature of the delete operation to take a SkipNode as its parameter. Hence, we do not need to do a search on the skiplist as we already have the node to delete at hand. We then simply use the technique proposed by Herlihy and Shavit in [WGTT15] to mark the forward references of the node and prepare it for physical deletion.

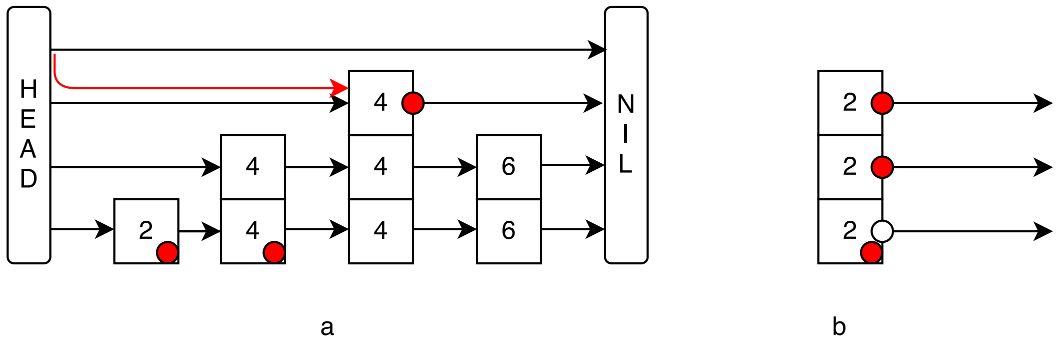

Figure 4.5: (a) Logical deletion of a wrong element. (b) Pictorial representation of a SkipNode for Shared Priority Queue. The bottom-right dot represents the *taken* flag for the item in the SkipNode, while the dots attached with the references represent the mark of the references.

4.3.2 Relaxed find-min operation

A `find_min` operation traverses the bottom level list and returns one of the first `k` unmarked elements from the list. The details of the implementation is shown in listing (14). The algorithm starts at the head of the skiplist. It chooses a random number `num_elements_to_skip` in range `(0, k]` (line 6). It then traverses the bottom level list until it skips `num_elements_to_skip` number of unmarked elements (lines 9-12). It then checks the mark of the element. If the element is marked, it moves forward until it lands on an unmarked element. It then returns the node. In case the algorithm cannot find a marked node before reaching NIL, it can mean two situations. First is that the list does not have any unmarked elements. In this case, the value of skip_count is 0, in which case, the algorithm returns nullptr. In the second case, where skip_count is greater than 0, it means that the number of unmarked elements in the list is less than num_elements_to_skip. Hence, the algorithm now picks a random number in range (0, num_elements_to_skip]. The algorithm retries this procedure max_retries number of times before failing and returning nullptr.

Algorithm 14: Implementation of relaxed `find_min` operation on a shared priority queue.

```
Data: None
Result: min_value, peeked_node, item
skip_count = k; // Relaxation constant
max_attempt = 2;
bottom_level = 0;
repeat
    cur = head.forward[bottom_level];
    num_elements_to_skip = random number in range [0, skip_count);
    skip_count = 0;
    while cur != NIL do
        if skip_count < num_elements_to_skip then
            cur = cur.forward[bottom_level];
            if !cur.item.taken then
                skip_count++;
            end
        end
        else
            if cur.item.taken then
                cur = cur.forward[bottom_level];
            end
            else
                // Found an unmarked node.
                peeked_node = cur;
                value = cur.item.value;
                return value, peeked_node, cur.item;
            end
        end
    end
    return nullptr;
until --max_attempt > 0 and skip_count > 0;
```

4.3.3 Delete-min operation

A `delete_min` operation uses the `find_min` operation to find an unmarked node in the skiplist. It then uses `TAS` to atomically set the nodes `item.taken` flag to true. If it succeeds, it calls a delete method on the skiplist with the skipnode returned by `find_min` operation to mark the forward references of the skipnode. If it fails, it retries the operation by calling the `find_min` operation once more. If it fails this time too, it gives up and returns `false`. Listing (15) shows the details of the algorithm.

Algorithm 15: Implementation of `delete_min` operation on a shared priority queue.

```
Data: None
Result: min_value
min_value, peeked_node, item = find_min();
if TAS(item.taken) succeeds then
    return delete(peeked_node)
end
else
    // Retry once.
    min_value, peeked_node, item = find_min();
    if delete(peeked_node) succeeds then
        return true;
    end
    else
        return false;
    end
end
```

4.4 Combined Priority Queue

The need for a combined priority queue arises from the fact that a distributed priority queue has a low guarantee on the ordering of the elements but low contention on `delete_min` and a shared priority queue has a higher guarantee but low contention on `delete_min`. By combining the two priority queues, we can get an adjustable priority queue with relatively low contention and relatively higher ordering guarantees. We define a constant `k` which determines the relaxation threshold for our queue. The priority queue is composed of a shared queue common to all the threads accessing the queue, and a separate local priority queue for each thread. The combined priority queue provides two main functions: `insert` and `delete_min`. An `insert` operation simply calls the local distributed queue's insert method with the shared queue as a parameter (listing (6)). A `delete_min` operation, however, tries to return the best value from both the queues. Listing (16) shows the details of the algorithm. If the algorithm succeeds in peeking items from both its queues it selects the best one from them. In case an element from the distributed queue is selected, then the algorithm tries to set its `item.taken` to true atomically to delete it logically (lines 5-8). Otherwise, if an element from the shared queue is selected, it first tries to set the item's `taken` flag to true atomically and then calls the shared queue's delete method to mark the forward references of the skipnode (lines 9-14). In case both the elements peeked are equal, the algorithm gives priority to the local element (line 5). If however, the algorithm succeeds in peeking an element from only one queue (lines 17-27), then it returns the element from that queue, but only if it succeeds in atomically setting the `item.taken` flag to true and marking the forward references in case of shared queue. Finally, if the algorithm finds that both the queues are empty, then it tries to `spy` elements from other threads and retries the operation if spying succeeds.

Algorithm 16: Implementation of `delete_min` operation on a combined priority queue.

```
Data: k
Result: min_value, bool
repeat
    dist_value, dist_item = dist_pq.find_min();
    shared_value, shared_item, shared_node = shared_pq.find_min();
    if dist_item and shared_item then
        if dist_value <= shared_value then
            if TAS (dist_item.taken) succeeds then
                return dist_value, true;
            end
            else
                if TAS (shared_item.taken) succeeds then
                    bool success = shared_pq.delete(shared_node);
                    return shared_value, success;
                end
            end
        end
    end
    if dist_item then
        if TAS (dist_item.taken) succeeds then
            return dist_value, true;
        end
    end
    if shared_item then
        if TAS (shared_item.taken) succeeds then
            bool success = shared_pq.delete(shared_node);
            return shared_value, success;
        end
    end
until dist_pq.spy() > 0;
return None, false;
```

4.4.1 Relaxation

Since our combined priority queue is relaxed by a constant `k`, a `delete_min` operation does not necessarily remove an item with the lowest global priority. If there are `N` threads concurrently accessing the priority queue, then our relaxation ensures that a call to a `delete_min` removes any of the first `T*k` elements from the queue. The reason is that a thread does not consider keys from other threads' local priority queue allowing it to skip a total of `(N - 1)*k` elements. Further, as our shared queue is also relaxed by the same constant `k`, it can skip up to `k` elements from the shared queue. Hence, the total number of elements that can be skipped is `(N - 1)*k` + `k` = `N*k`. To give an example, if the relaxation constant `k` is 10, and the number of threads concurrently accessing the queue `N` is 10, then the `delete_min` operation can remove any of the first 100 smallest elements from the queue.

Chapter 5

Correctness

In this section, we show that our proposed priority queues are lock-free and linearizable according to our `k` relaxation semantics. We start by showing the correctness of our distributed priority queue and then move on to the shared priority queue before moving to the combined priority queue.

5.1 Distributed Priority Queue

Correctness proofs for various operations on a distributed priority queue are as follows:

Lemma 5.1.1 *The `insert` operation on a distributed queue is lock-free.*

Proof: Since only one thread can insert elements into the distributed queue, it is trivially wait-free. ■

Lemma 5.1.2 *The `find_min` operation on a distributed queue is wait-free.*

Proof: Since a `find_min` operation only loops through the local skiplist and does not mark the unmarked item found, it is not influenced by the operation of other threads. Hence, it is wait-free. ■

Lemma 5.1.3 *The `delete_min` operation on a distributed queue is lock-free.*

Proof: A spying thread can mark an item returned by a `find_min` operation, hence causing the thread to call `find_min` again, and restart the operation. This process can repeat an arbitrary number of times, and the operation can fail. However, this means that other threads are successfully marking the item and hence succeeding. This proves that a `delete_min` operation is lock-free. ■

Lemma 5.1.4 *The `spy` operation on a distributed queue is wait-free.*

Proof: A spy operation simply iterates through the victim's item list. It creates a new SkipNode for each unmarked item in the victim's task list. While copying, the spying thread does not try to mark the item. Hence, multiple threads can spy the same element from a single victim and succeed. Hence, there is no interference with a spy operation by the operations of other threads. This concludes that the operation is wait-free. ■

Lemma 5.1.5 *The `insert` operation on a distributed queue is linerizable.*

Proof: The linearization point for an `insert` operation is when it physically links the bottom level of the new element to the lowest level of the skiplist. ■

Lemma 5.1.6 *The `find_min` operation on a distributed queue is linearizable.*

Proof: The linearization point for a successful `find_min` operation is when a thread successfully finds an unmarked element. In case of an unsuccessful `find_min` operation, it linearizes at the point when it reaches the skiplist's `NIL` and returns a `nullptr` value. ■

Lemma 5.1.7 *The `delete_min` operation on a distributed queue is linearizable.*

Proof: For a successful `delete_min` operation, the linearization point is when a thread successfully sets the item's `taken` flag to true. In case of an unsuccessful `delete_min` operation, the linearization point is when it returns a `nullptr` value. ■

Lemma 5.1.8 *The `spy` operation on a distributed queue is linearizable.*

Proof: For a spy operation, the linearization point is when the spying thread completes iterating the victim's task list. ■

5.2 Shared Priority Queue

The shared skiplist which is used to implement our shared priority queue is lock-free and linearizable [HS08]. In this section, we will only show proofs for the methods that we added on top of Herlihy and Shavit's skiplist [HS08].

Lemma 5.2.1 *The `find_min` operation on a shared queue is wait-free.*

Proof: Since a `find_min` operation just scans the bottom level list of the shared skiplist without trying to mark an element, there is no interference by other threads in its operation. A thread calling `find_min` operation will traverse the bottom level list in search for an unmarked node for a maximum of `max_attempt` times. Hence, it is wait-free. ■

Lemma 5.2.2 *The `delete_min` operation on a shared queue is lock-free.*

Proof: A `delete_min` operation calls a `delete` operation on the node peeked by a `find_min` operation. Since the `delete` operation is lock-free [HS08], `delete_min` operation is also lock-free. ■

Lemma 5.2.3 *The `find_min` operation on a shared queue is linearizable with `k` relaxation semantics.*

Proof: A `find_min` operation first chooses a random number in the range [0, `k`). It then skips `k` number of unmarked elements in the queue and starts finding an unmarked element. This means a `find_min` operation selects one of the first `k` unmarked elements, which is in accordance to the `k` relaxation semantics. The linearization point for a successful `find_min` operation is when the thread finds an unmarked element. For an unsuccessful `find_min` operation, the linearization point is the point when it returns a `nullptr`. ■

Lemma 5.2.4 *The `delete_min` operation on a shared queue is lineriazable with `k` relaxation semantics.*

Proof: A `delete_min` operation calls a `find_min` operation to get an unmarked node in the skiplist. It then calls the shared skiplist's `detele` method to mark the node's forward references. Since both `find_min` and `delete` operations are linerizable with `k` relaxation semantics, a `delete_min` operation is also linearizable with `k` relaxation semantics. The linearization point for a successful `delete_min` operation is when the underlying `delete` method successfully marks the bottom level forward reference of the node being deleted. For an unsuccessful `delete_min` operation, which occurs as a result of failed `TAS` on the peeked item's `taken` flag, the linearization point is where the `delete_min` operation returns `false`. A `delete_min` can also fail as a result of a concurrent thread trying to delete the same element. In case the competing thread succeeds in deleting the element, the operation fails and the linearization point is the point when the succeeding thread succeeds in marking the bottom level forward reference of the node being deleted. ■

5.3 Combined Priority Queue

We now show that the operations `insert` and `delete_min` on a combined priority queue is lock-free and linearizable with `k` relaxation semantics.

Lemma 5.3.1 *The `insert` method on a combined priority queue is lock-free.*

Proof: An `insert` method on a combined priority queue calls the distributed queue's `insert` method with the shared priority queue as a parameter. If the number of elements in the queue is less than relaxation parameter `k`, then the element is inserted into the distributed priority queue, which is wait-free. If, however, the number of elements in the combined queue is greater or equal to `k`, then the element is inserted into the shared queue. As mentioned in section (5.2), the shared priority queue is lock-free. Hence, an `insert` operation on a combined priority queue is also lock-free. ■

Lemma 5.3.2 *The* `delete_min` *operaton on a combined priority queue is lock-free.*

Proof: A `delete_min` operation calls `find_min` operation on both distributed queue and shared queue. It then tries a `TAS` on the `taken` flag of the element with the lower `key` to claim it for deletion. In this process, it can be beaten by other threads concurrently trying to mark the element, arbitrary number of times. Although the thread is unable to make progress, it means that other threads are making progress by marking the element. Hence, the operation is lock-free. ■

Lemma 5.3.3 *The* `insert` *operation is linearizable according to* `k` *relaxation semantics.*

Proof: An `insert` method can insert the new element in any of the two internal priority queues. If the size of distributed queue is less that `k`, insertion is done in the local queue, which is wait-free as shown in Lemma(5.1.1). If, however, the size of the distributed queue is greater or equal to `k`, then the insertion is done in the shared queue, which is linearizable [HS08]. The linearization point, in this case, is the point when the operation successfully links the bottom level of the new node in the skiplist. ■

Lemma 5.3.4 *The* `delete_min` *operation is linearizable with* `k` *relaxation semantics.*

Proof: A `delete_min` operation can select an element from either the distributed queue or the shared queue. In case an element from the distributed queue is selected, the operation linearizes at the point where it successfully sets the peeked item's `taken` flag to `true`. If it fails `TAS` on the item's `taken` flag, the operation fails, and the linearization point is where it returns `false`. In case an element from the shared queue is selected, the linearization point for a successful operation is when it marks the bottom level reference of the element in the shared skiplist. In case the operation is unable to mark the peeked item, it fails, and the linearization for an unsuccessful `delete_min` operation is where it returns `false`. A `delete` operation on a shared queue can also fail as a result of other threads concurrently trying to delete the element under consideration. In this case, the operation fails, and the linearization point is the point where the successful thread succeeds in deleting the element.

A `delete_min` operation on a combined priority queue skips a total of `(N - 1).k` elements from other threads' local queues, where `N` is the number of threads. This is because `k` bounds the size of the distributed priority queue. It can also skip up to `k` elements from the shared priority queue. Hence, the operation can skip a total of `Nk` number of elements, which is the relaxation bound of the algorithm.

The operation, however, fulfills the local ordering semantics. This is because, a thread checks both the local queue and the shared queue, which may contain elements added by the given thread, before deleting an element. ■

Chapter 6

Experimental Results

We base our performance on `throughput` i.e. the number of operations completed (`insert` and `delete_min` combined) within a certain timeframe. As in [GTW16], we prefill each priority queue with 10^6 elements before starting the benchmark. We then measure the throughput for 10 seconds and then finally report on the number of operations performed per second. The metric is similar to the one proposed by Wimmer et al. in [GTW16].

The behavior of the throughput is controlled only by the `workload` parameter (currently). Workload may be,

- `uniform`, meaning that each thread may execute roughly equal number of `insert` and `delete_min`, or,
- `split`, meaning that half the threads `insert`, while half of them `delete`, or,
- `alternating`, meaning that each thread alternates between insertions and deletions strictly.

Our benchmark uses 32 bit ascending integers as keys. What this means is that each next key we feed the priority queue is one more than the previous one. The current benchmark only uses integers, but we can easily extend it to use other data types like floats, doubles, or other custom data types.

We use seven different queues for the benchmark.

- `LockedPQ`: A `LockedPQ` serves as a baseline for acceptable performance of a priority queue. It is based on skiplist and coarse-grained locking.
- `DistPQ`: It is the priority queue proposed in section (4.2).

- `SharedPQ32/32`: It is the priority queue proposed in section (4.3). For benchmark purposes, we relax the shared queue by `k` = 32.
- `CombinedLockedPQ`: A priority queue composed of `LockedPQ` and `DistPQ`. Each thread owns a local `DistPQ`, and all of them share a global `LockedPQ`. This is also a relaxed priority queue. A `delete_min` operation can skip up to $N - 1 * k$ elements, where N is the number of threads concurrently accessing the priority queue, and k is an upper bound on the size of each `DistPQ`.
- `CombinedNBPQ32/32`: It is the priority queue proposed in section (4.4) with `k` = 32.
- `CombinedNBPQ128/128`: It is the priority queue proposed in section (4.4) with `k` = 128.
- `CombinedNBPQ4096/4096`: It is the priority queue proposed in section (4.4) with `k` = 4096.

The queues were benchmarked on a Macbook Pro (3.1GHz Intel Core i7) with 8GB 1867 MHz DDR3 RAM with **4 logical cores**.

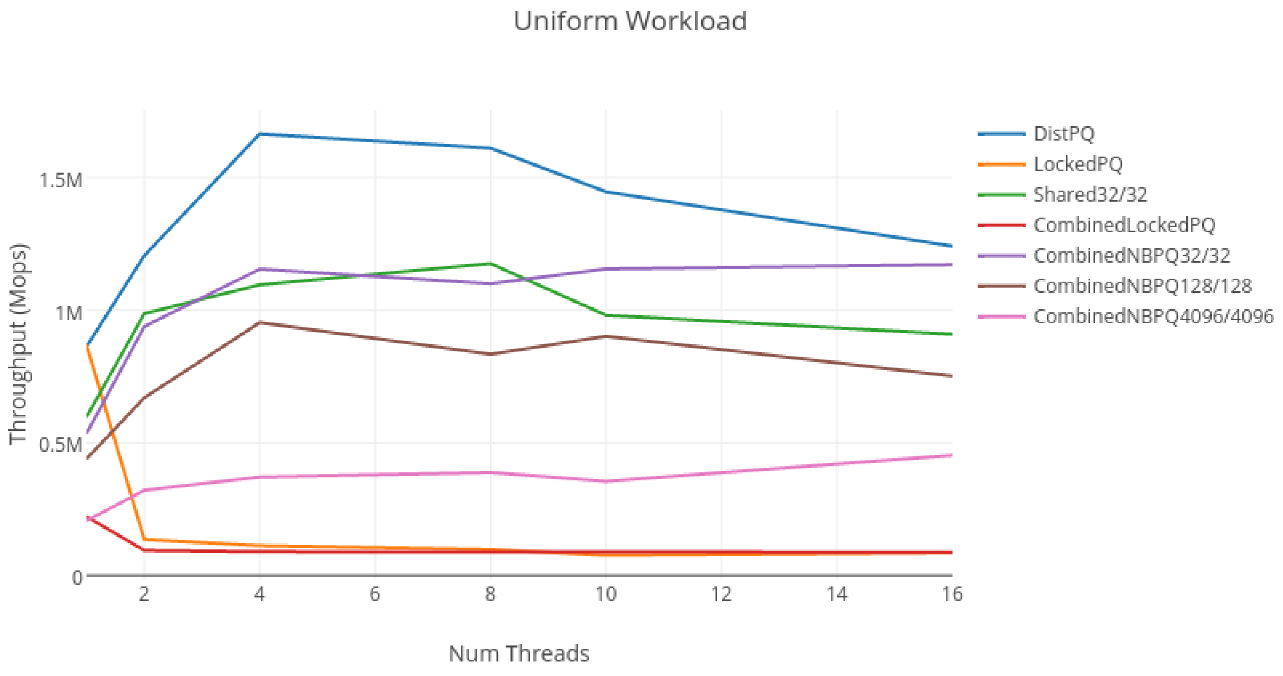

Figure 6.1: Throughput for uniform workload.

Figure (6.1) shows the throughput in Mops of various queues for uniform workload distribution. As in the figure, the `DistPQ` has the highest Mops, which peaks to almost 2 Mops for four threads.

Since a `DistPQ` has low contention, it has higher throughput than others. The `SharedPQ32/32` has a relatively lower throughput than the `DistPQ`. It is because of the contention in it's `delete_min` operation. A `CombinedNBPQ32/32`, which is the combination of a `DistPQ` and a `SharedPQ32/32`, has a throughput somewhere between the above two queues. A combined priority queue is sensitive to the relaxation parameter, the number of threads and the machine on which it is run. As depicted in the figure, as we increase the relaxation parameter to 128 and 4096, the performance of the queue deteriorates. This is partly because of the relaxed `find_min` operation of the `shared_pq` and the `spy` operation of the `dist_pq`. A relaxed `find_min` tries to skip a random number of nodes between 0 and `k` even if an unmarked node is on its way and a `dist_pq` spy operation tries to copy more elements to the local queue as the relaxation parameter increases.

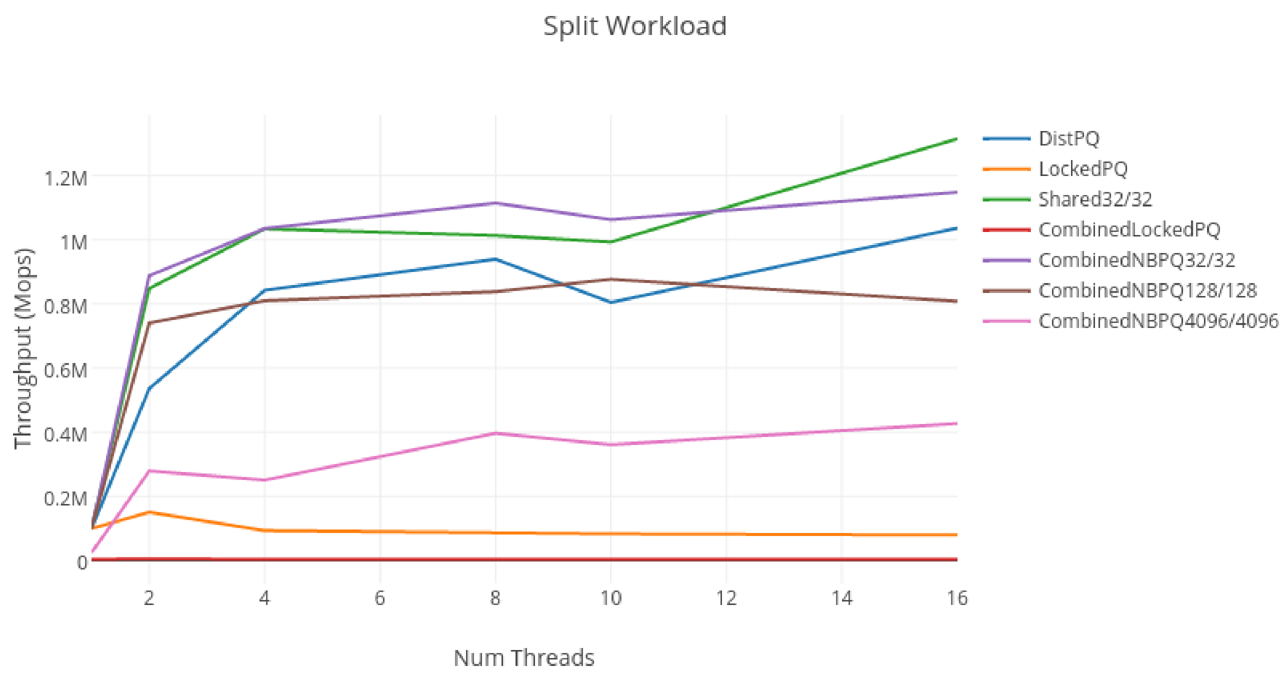

Figure 6.2: Throughput for split workload.

For split workload, as depicted in Figure (6.2), the scenario is a little different. Here, a `SharedPQ32/32` and a `CombinedNBPQ32/32` show higher performance as the number of threads increase. A `DistPQ` has a throughput lower than both `SharedPQ32/32` and `CombinedNBPQ32/32`. The reason behind this is again the spy operation. Since half the threads are inserting threads, the items in their local queues can be deleted only by the deleting threads' spy operation. The same reason applies for the decreasing throughput with an increase in relaxation parameter `k`.

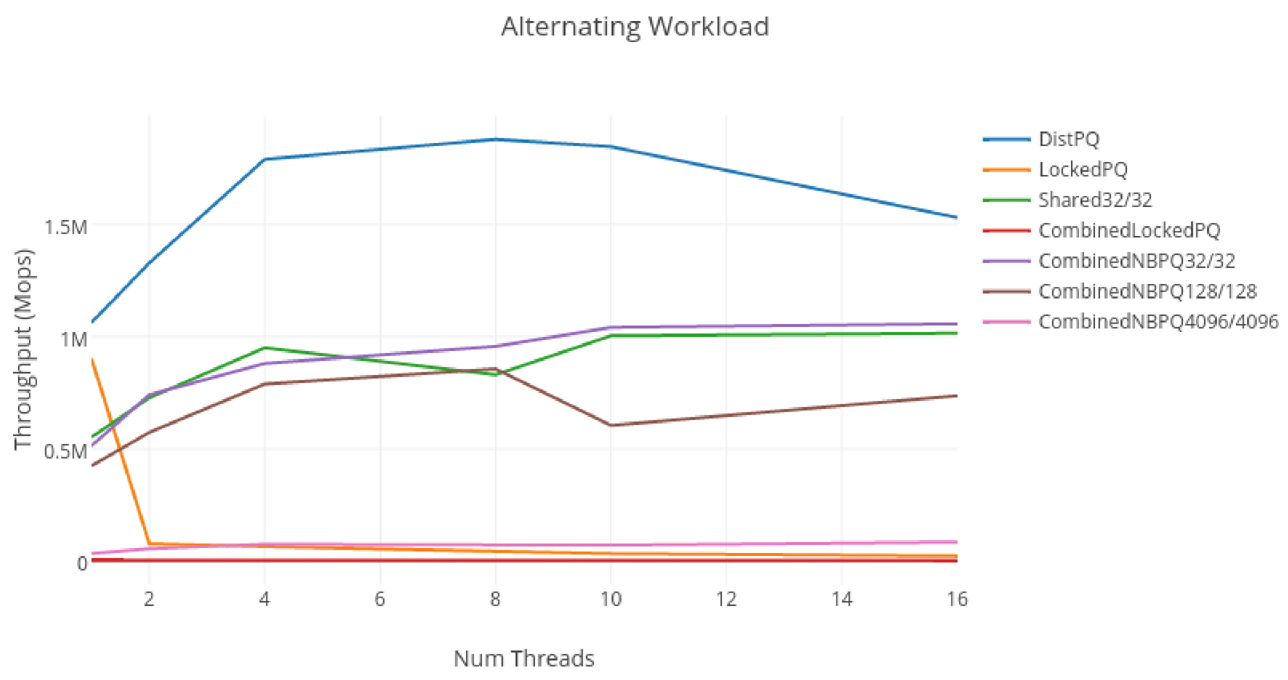

Figure 6.3: Throughput for alternating workload.

Finally, for alternating workload, as expected, a `DistPQ` has the highest throughput. As threads have items in their local queues most of the time, the spy operation is minimized, hence increasing the performance. A `CombinedNBPQ32/32` shows a slightly higher throughput than a `SharedPQ32/32` for alternating workload.

All the combined priority queue variations are well above the accepted performance given by the `LockedPQ`.

Chapter 7

Conclusion and Future Work

We presented a non-blocking priority queue based on skiplists with relaxed semantics. Our priority queue was internally composed of two priority queues with completely different characteristics. One of them was a distributed priority queue, which had high throughput but a very low ordering guarantee, while the other was a shared priority queue, which had a higher ordering guarantee, but relatively lower throughput. By combining the two queues, we were able to get a queue with higher throughput and good ordering guarantee, i.e., relaxed by `k` allowing it to remove any of `N * k` lowest priority elements. We gave theoretical correctness for our algorithm, which proves that our algorithm is lock-free and linearizable. We also benchmarked our priority queue with other variations of priority queues like locked, shared, combined, distributed, etc. and showed that our proposed queue performs as expected.

One of the places that needs improvement in the proposed algorithm is the relaxed `find_min` operation of the shared queue. Instead of choosing a random number in range `[0, k)` and then doing a bottom-level walk skipping all the nodes (both marked and unmarked) on the way, we could use a technique similar to the `spray` operation as described in [AKLS15]. Further, we could replace the skiplist implementation by the one proposed in [LJ13]. One of the advantages of [LJ13] is that it allows for batch deletions, hence making deletions super fast. The difficulty in using it with our implementation, however, is that it is hard to construct a relaxed priority queue on top of it.

Trying out the priority queue proposed with a mixture of various data structures instead of just skiplists would be an interesting future work. One such example would be to use a heap in the local queue implementation and a skiplist in the shared queue implementation.

Currently, our benchmarks only run on a machine with 4 logical cores. It would be interesting to run the benchmarks on a more powerful machine with a higher number of logical cores and see the results.

Another significant area to work on is the integration of a concurrent memory management scheme. Implementing a memory management scheme and exposing the priority queue as a library for public use would be another interesting future work.

Bibliography

[AB] Faculty Mentor: Jennifer Welch Alyssa Byrnes, Graduate Mentor: Edward Talmage. Dreu research.

[ADF$^+$00] Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele, Jr. Dcas-based concurrent deques. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 137–146, New York, NY, USA, 2000. ACM.

[AKLS15] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. *SIGPLAN Not.*, 50(8):11–20, January 2015.

[CGR12] Tyler Crain, Vincent Gramoli, and Michel Raynal. *Brief Announcement: A Contention-Friendly, Non-blocking Skip List*, pages 423–424. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[CMH14] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The adaptive priority queue with elimination and combining. *CoRR*, abs/1408.1021, 2014.

[EFHR14] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 332–340, New York, NY, USA, 2014. ACM.

[ERAEB05] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2005.

[GTW16] Jakob Gruber, Jesper Larsson Träff, and Martin Wimmer. Benchmarking concurrent priority queues: Performance of k-lsm and related data structures. *CoRR*, abs/1603.05047, 2016.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[JT16] Siddhartha V. Jayanti and Robert E. Tarjan. A randomized concurrent algorithm for disjoint set union. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 75–82, New York, NY, USA, 2016. ACM.

[KK16] Nikolaos D. Kallimanis and Eleni Kanellou. Wait-Free Concurrent Graph Objects with Dynamic Traversals. In Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[LJ13] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*, OPODIS 2013, pages 206–220, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[LZS14] Yujie Liu, Kunlong Zhang, and Michael Spear. Dynamic-sized nonblocking hash tables. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 242–251, New York, NY, USA, 2014. ACM.

[MVS09] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 45–54, New York, NY, USA, 2009. ACM.

[PT13] Erez Petrank and Shahar Timnat. *Lock-Free Data-Structure Iterators*, pages 224–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[Sha14] Niloufar Shafiei. Non-blocking doubly-linked lists with good amortized complexity. *CoRR*, abs/1408.1935, 2014.

[SL00] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 263–268, 2000.

[THP15] Shahar Timnat, Maurice Herlihy, and Erez Petrank. *A Practical Transactional Memory Interface*, pages 387–401. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[Ven11] Balaji Venu. Multi-core processors - an overview. *CoRR*, abs/1110.3535, 2011.

[WCV$^+$13] Martin Wimmer, Daniel Cederman, Francesco Versaci, Jesper Larsson Träff, and Philippas Tsigas. Data structures for task-based priority scheduling. *CoRR*, abs/1312.2501, 2013.

[WGTT15] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. *CoRR*, abs/1503.05698, 2015.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Ashok Adhikari

Degrees:

Bachelor of Computer Engineering 2011

Tribhuwan University, Nepal

Thesis Title: Non-blocking Priority Queue based on Skiplists with relaxed semantics

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.

Committee Member, Dr. John Minor, Ph.D.

Committee Member, Dr. Yoohwan Kim, Ph.D.

Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.