Using Layering and Data Partitioning Techniques to Increase NUMA-Locality and Performance in Concurrent Data Structures

Davidson College

Sam Thomas

May 5, 2020

Abstract

Data structures are fundamental components of efficient computing. With the ever-increasing role of multiprocessing in computing, the need for thread-safe concurrent data structres is paramount. In particular, Non-Uniform Memory Access (NUMA) machines are more efficient in accessing particular partitions of memory, so modern concurrent data structures should be built with NUMA architecture in mind.

This thesis proposes a technique for designing concurrent data structures that optimizes memory accesses for hierarchical NUMA machines by utilizing a novel partitionable skip list variant called a skip graph. It also proposes a technique that layers thread-local sequential data structures on top of the partitioned skip graph to minimize traversals across synchronized variables. These techniques allow the construction of NUMA-aware data structures that are highly adaptable to different implementations that will increase performance in several different concurrent environments.

Dedication

To anyone who's thought *why me? why now?* and made it out the other side to tell the tale. You're capable of so much more than you give yourself credit for.

Declaration

I declare that the work that went into this research and thesis was my own with the help of Dr. Hammurabi Mendes and Dr. Jonad Pulaj, who co-advised the writing of this thesis, and upheld the standards of the Davidson College Honor Code.

Acknowledgements

I want to thank my thesis committee for taking the time to work through the materials presented in this thesis even with the challenges that come with being a professor during COVID-19.

I want to thank Dr. Mendes for all of his help with research and beyond. Trying to quantify his guidance over the past three years or so in words would not do justice to the love of research he fostered in me or the personal growth he helped me reach.

I want to thank Ana Hayne for all of her help with proof-reading this thesis and being a source to commiserate over the thesis writing process with. If you're reading this final draft, we did it!

I want to thank all of my friends who have been so supportive of me over the past four years - even when I was hidden in the library late at night instead of hanging out. To name a few; thank you to the members of the men's and women's track teams - especially the seniors, Fresh Kitties forever - for always having my back through thick and thin; thank you to my first-year hallmates on base rich for making me feel at home and always making me laugh; thank you to the pep band for escape from reality at basketball games - crazy how shouting can relieve stress.

I also wanted to give a special shoutout to Alex Hazan and Jake Clary for everything over the past few years. I really cannot imagine having done college - let alone this thesis - without you both. It's going to be a very great day when we're all together again.

Contents

Chapter 1

Introduction

Concurrent programming is a technique that utilizes multiple *threads* in parallel with shared resources to achieve increased performance as compared to sequential programming alternatives. Threads consist of individual processes acting independent of one another to achieve their own ends, namely by making a call to a thread function on which it will run. The need for concurrent programming grows inexorably as multicore computers become bigger and more commonly accessible. Yet, as a whole, the task of concurrently programming algorithms that are correct and run fast introduces a variety of challenges.

Ideally, seeing as threads act in parallel, it would be expected that each additional thread added to an environment would increase the overall *throughput* by 100%. Throughput is defined as the quantity of operational performance. That is, suppose there is a single thread executing a process at 1000 operations per millisecond. If a second thread were to be added, a naïve expectation may be that there would now be 2000 operations per millisecond. Introducing another thread would give 3000 operations per millisecond and so on. In practice, there must be some form of *synchronization* between threads in order to ensure that all data is correctly preserved in a way that makes sense. These synchronization mechanisms degrade from the idealized performance of parallel computing on shared data, which will be introduced and discussed throughout this chapter.

1.1 Synchronization

Synchronization refers to the notion of preserving a global definition for the state of any particular data at any point in time. The effects of the lack of synchronization on the correct preservation of data are first displayed in the linked list Fig. 1.1. The figure shows two threads operating on a list that originally is only made of three nodes, 7, 16, and 31. There are three critical frames of the

Notice that in state (a), the linked list is made up of nodes for the values of 7, 16, and 31. In state (b), notice that Thread 1 has already began a call to `insert(20)` into the linked list. In particular, Thread 1 has allocated a new node with value 20 and pointed it to 31. Thread 1 still has to update 16's pointer from 31 to 20. However, notice that in state (b) this operation is happening in parallel with thread 2, which makes a call to `remove(16)`. In order to do such an operations, Thread 2 will need to "relink" the pointer from node 7 from 16 to 31. Both operations, having completed these steps, should return **`true`**.

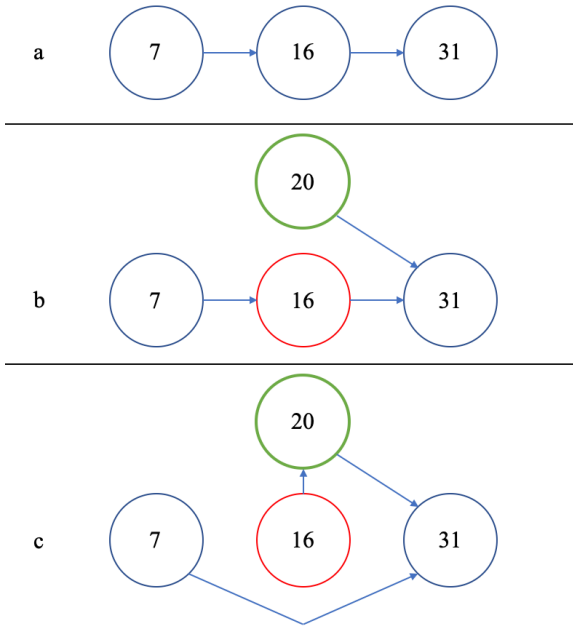

Figure 1.1: A concurrent sequence of operations on a linked list. Step (a) is the original state of the list. Step (b) shows Thread 1 inserting 20 in green and Thread 2 removing 16 in red. Step (c) shows the final state of the linked list, where only 7 and 31 are discoverable.

Note that state (c) shows that this removes both 16 and 20 as 20 is only discoverable through 16. Yet, based on the return types of Thread 1 and Thread 2's operations, the expected state of the linked list would be to have 7, 20, and 31, as 20 was successfully inserted and 16 was successfully removed. At least, this was what was expected based on the returned values by the thread functions. This inconsistency between expected and observed outcomes warrants the implementation of synchronization mechanisms to preserve correct outputs.

Lock-Based Synchronization

One way to achieve synchronization is through a *lock-based* implementation. Locks are objects that ensure that only a single thread can proceed through a certain portion of the code at a time while other threads attempting to acquire the lock are left "waiting" or "spinning"[1] until it can be acquired. Though there are many potential implementations of locks, they all promote the same general idea. Namely, they ensure that only one thread can operate within a *critical section*, or the code between obtaining and releasing a lock object, at a time. That is, locks can be used to avoid the situation where thread 2 called `remove(16)` in state (b) of Fig. 1.1 and removed more than one node on a single remove call from the linked list.

The following is a naïve way to implement *mutual exclusion*, or a situation where only one thread can operate on a shared variable at a time. Acquire a global lock before making any changes to the linked list. Only once the lock is acquired can you make any changes. Release the lock when finished. This can be seen in Alg. 1.

[1]This is implementation dependent. Generally, wait-based locks are better for longer periods of being locked out and spinning is better for shorter periods of being locked out.

Algorithm 1 Global Lock

```
1: ▷ Lock has lock() and unlock() capabilities
2: lock = GlobalLock()
3: procedure INSERT(key)
4:     toInsert = Node(key)
5:     ▷ Assume search gives access to the node immediately larger and smaller
   than the desired key
6:     if search(key) is false then
7:         lock.lock()
8:         toInsert.next = next
9:         previous.next = toInsert
10:        lock.unlock()
11:        return true
12:    return false
13: procedure REMOVE(key)
14:    ▷ Assume search gives access to the node immediately larger and smaller
   than the desired key
15:    if search(key) then
16:        lock.lock()
17:        previous.next = next
18:        lock.unlock()
19:        return true
20:    return false
```

The problem with using a single global lock to implement mutual exclusion is that it causes *starvation*. Starvation is the idea that certain threads will never have the opportunity to act on the object. Imagine the following execution of two threads operating on the linked list in Fig. 1.1 with the above implementation. First, Thread 1 calls `Insert(20)` and then Thread 2 calls `Remove(16)`. Suppose Thread 1 wins the global lock. That is, Thread 2 has to wait for Thread 1 to release the lock in order for it to proceed with its execution. However, suppose in this situation that the operating system temporarily deschedules Thread 1 so it can perform other system-wide tasks or run other threads. Now, all of Thread 2's execution time is spent waiting on Thread 1 to release the lock while Thread 1 is dormant. In this situation, Thread 2 is *blocked* from execution. This is more likely to occur as contention is increased, which happens by increasing the number of threads and/or reducing the element space in which threads can operate.

It is important to note that, although Thread 2 is subject to starvation, the overall correctness of the structure remains is preserved at all times. The challenge presented by starvation is one of performance rather than correctness. While correctness and performance are important both in concurrent programming, correctness trumps performance to ensure that the program can run successfully and execute as expected. As such locks are essential over no synchronization mechanisms.

The program's throughput can be improved with a more strategic use of locks. For instance, consider the following implementation of "fine-grained locking" as is demonstrated in Alg. 2. The idea is to embed a lock in each node, so that no two threads can operate on the same node at the same time. This will allow for more threads to operate in parallel if they operate on different parts of the data, while

still ensuring that the data is preserved.

Algorithm 2 Fine Grained Locking

```
1: ▷ Node is a structure that holds data and a lock
2: procedure INSERT(key)
3:     toInsert = Node(key)
4:     ▷ Assume search gives access to the node immediately larger and smaller
   than the desired key
5:     if !search(key) then
6:         toInsert.next = next
7:         previous.lock()
8:         previous.next = toInsert
9:         previous.unlock()
10:        return true
11:    return false
12: procedure REMOVE(key)
13:    ▷ Assume search gives access to the node immediately larger and smaller
   than the desired key
14:    if search(key) then
15:        previous.lock()
16:        current.lock()
17:        next.lock()
18:        previous.next = next
19:        previous.unlock()
20:        current.unlock()
21:        next.unlock()
22:        return true
23:    return false
```

In the general case, this implementation of lock-based synchronization improves upon the throughput of a singular global lock because more threads can operate in parallel. Again, consider the case of Fig. 1.1 where Thread 1 calls `Insert(20)` and Thread 2 makes a call to `Remove(16)`. In this situation, Thread 1 and Thread 2 can both operate on the structure in a thread-safe manner due to the limited *contention* between threads. That is, Threads 1 and 2 are only contending for a singular shared lock, which is 16. Furthermore, the data remains protected because two threads cannot operate on the same data at the same time.

Lock-based synchronization is not relevant for the purposes of this thesis. This largely comes from the inevitable bottlenecking that occurs under high-contention environments. While it is inevitable that this bottleneck will occur in high contention, lock-based synchronization renders threads totally dormant, which is an ineffective use of CPU resources and intuitively defeats the purpose of having multiple threads. Such a situation motivates the alternative utilization of a lock-free form of synchronization.

Lock-Free Synchronization

Compare-and-swap (CAS) is a lock-free atomic hardware operation on an atomic variable that acquires a cache line in exclusive mode and flushes the updated variable

to other CPUs in the event that its present state matches the provided expected state on comparison. To demonstrate this, suppose that there was an unsigned, 64 bit integer `atomic<uint64_t>x=5` and a process wanted to, in a thread-safe and lock-free manner, increment it to `6`. To do so, that process would call `x.CAS(5,6)` and `CAS` would only return true if the value of `x` was `5` and that this current thread was the one that could push `6` to the value of `x`.

CAS is the most commonly and successfully used lock-free synchronization technique. It is an *atomic operation*, meaning that it is indivisible by any other operation. That is, it appears in-place in context of the surrounding code regardless of compiler optimizations. Furthermore, the nature of CAS means that it operates on *atomic variables*, or variables that are visible to all threads at all times. Note that operating on atomic variables is much slower than operating on local memory in that local variables must only be flushed to the local cache whereas atomic variables must flushed to the cache of each CPU.

In this sense, CAS operations are particularly expensive in the number of clock cycles needed to perform the overall operation. Furthermore, each atomic variable must be written to by a CAS operation individually, which can create a lot of overhead when making many write calls. As such, the most effective utilization of CAS as a lock-free synchronization mechanism still minimizes its usage.

1.2 Linearizability

In order to have a properly correct concurrent application, there must be some standard of correctness in place. The standard for judging whether or not a concurrent application is correct is defined as whether or not its operations are *linearizable*. That is, a program is linearizable when all operations that happen concurrently have an ordering. Each operation must have definably happened before or after all other operations to make this ordering. This definition of linearizability suggests that each operation must be defined by certain checkpoints in either the code or in the logical understanding of how the code operates called *linearizability points*.

To demonstrate exactly how linearizability and linearizability points work in practice, consider the example of two threads inserting to and one thread removing from a queue, implemented by an array, concurrently. Suppose performing an insertion into the structure is done by (i) successfully being given access to a location in the queue[2] and (ii) adjusting the state of that location in the queue. Similarly, suppose removals are done by (i) examining the first location of the queue and, if that location is not empty, (ii) removing the element in that location. In order for inserting threads to linearize with one another, they will be ordered based upon the real-time ordering of being granted access to a location in the queue. Inserting threads will linearize with the removing thread at the point at which the cell is updated with the value.

To see how these definitions look in practice, consider example one of Fig. 1.2. Thread 1, thread 2, and thread 3 all perform their operations with a period of quiesence between them. As such, it must have been the case that thread 1 was given access to its location of the array before thread 2. Furthermore, seeing as the

[2]There are several ways this can be done. For the sake of this example, assume it is done by performing a CAS operation on an atomic boolean flag.

first location of the queue was updated before thread 3 began its remove call, it must be the case that thread 3 was able to successfully remove an element, and that element must have been (a).

Now consider example 2 in Fig. 1.2. Notice how thread 1 and thread 2 execute concurrently. Further, notice that thread 3 is invoked strictly after both thread 1 and thread 2 finish their execution. It follows that thread 3 will remove an element because the first location of the array will have been updated by either thread 1 or thread 2. Though, the question is raise - which element will thread 3 remove? There is a linearizable explanation for either element (a) or (b). If thread 3 removes (a), it must have been the case that thread 1 was granted access to the first position in the queue by finishing step (i) of the insertion method before thread 2. If thread 3 removes (b), then the same must have been true with thread 2 being granted access to the first position. Also notice how it is possible that such a situation could happen regardless of the fact that thread 1 began its method before thread 2. Thread 2 may still have reached that line first by benefitting from cached variables, thread 1 may have been momentarily descheduled by the operating system, or several other possible explanations. Either way, if thread 3 removes (b), it must have been the case that thread 2 completed the inserting linearizability point before thread 1.

In example 3 of Fig. 1.2, thread 1 overlaps with thread 2 and thread 3, where as threads 2 and 3 only overlap with thread 1. As such there are three possible outcomes of thread 3's call to remove: it could remove (a), it could remove (b), or it could remove nothing. Let's first consider the case where thread 3 removes the element (a). If thread 3 removes (a) it must have been the case that thread 1 was able to successfully be given access to the first position of the queue before thread 2 and must have put (a) in that location before thread 3 was able to check the location. It also must be the case that (b) is in the queue before thread 3 removes (a) in such a situation because thread 2 finished its call before thread 3's remove method was invoked.

If thread 3 removes (b), it must be the case that thread 2 was granted access to the first location of the queue before thread 1 was able to do so. Since thread 2 finishes its execution before thread 3 starts, it must be the case that (b) was in the queue when thread 3 checked the first location, so thread 3 removes the element, which is (b). Notice, however, that no determination can be made as to whether or not (a) is in the queue at the moment thread 3 begins its call to remove. That is, it may have been the case that thread 1 was not able to update the position it was

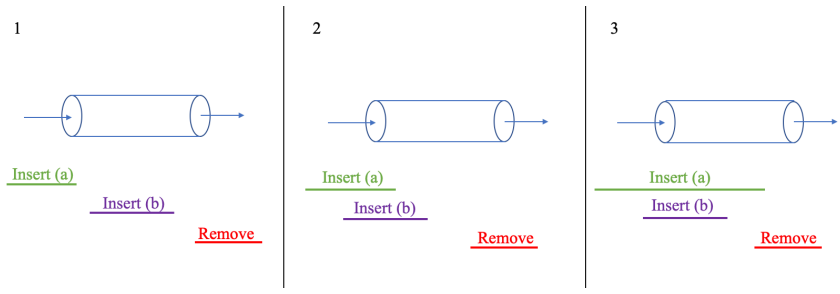

Figure 1.2: Three potential cases of linearizability. Thread lifespans are represented by colored line underneath the figure. Thread 1 inserts a node with value (a), Thread 2 inserts node with value (b), and Thread 3 removes the head element of the queue.

granted in the queue before thread 3 made its call to remove, but no determination can be made with any certainty.

Now consider the case where thread 3 removes nothing. If this were to happen, it must have been the case that thread 1 was granted access to insert into the first location of the array, but was unable to physically insert (a) into that location before thread 3 checked the first location of the queue. Notice that, in such a situation, thread 2 would have been granted access to the second position of the queue and finished inserting (b) into that location before thread 3 was able to start its execution. That is, the queue was not empty before thread 3 made a call to remove, but still returned false. This outcome may be undesirable, but is still perfectly legal within these definitions of linearizability. As such, building provably linearizable concurrent data structures is a highly non-trivial task.

These cases demonstrate the importance of having very clearly defined linearizability points in a concurrent environment to prove its correctness. The correctness of certain outcomes are highly dependent upon a particular implementation of an application. As such, discussing both the particularities of the implementation and its linearizability points are crucial.

1.3 Skip List

Much of the work in this thesis is built upon a *skip list*. A skip list is a probabilistic data structure that behaves and operates as an ordered set of nodes. They were first introduced by [17]. Skip lists allow for searching for a node in $O(log(n))$ time. In this sense, it is a very fast data structure and is frequently utilized in the development of concurrent algorithms because it provides tree semantics. Trees are not used in concurrent environments because their balancing requires a maintenance of a "root" element that will be subject to high contention. A skip list is made up of a series of linked lists that are organized in *levels*. Note that each level is itself a linked list, but the upper-level linked lists are part of the same node. For example, `13` belongs to three linked lists, but `13` is a single node. Levels are organized hierarchically and can be seen in Fig 1.3.

For a node to be "in" a skip list means for it to exist at the bottom level. Each node has numbered keys which can be seen in Fig 1.3 and an associated value; a maximum height, which is the number of lists it exists in; and its next pointers, which refer to the next node in each linked list. Note that this means that nodes will be have multiple references as possible "next" nodes in Fig 1.3. This makes the skip lists that are used for the purposes of this thesis *singly linked*. The maximum height of the node is established on the node's construction. It entails a "coin flip" at each level, and there is a 50% chance that a node will exist at the second level, 25% chance that it will exist at the third level, etc.

Searching for a node in a skip list starts with the head element at the maximum height of that skip list. From there, it traverses forwards in the current linked list as much as possible until it has reached a node greater than or equal to the current key. If it is equal to the current integer, it will return true. Otherwise, it will simply go down a level and repeat the process from that current node. As it goes down a level, it stores the current node in the predecessors array and the next node in the successors array. If it makes it to the bottom level and finds a node that is greater than the key it is searching for without finding the key, then it returns false.

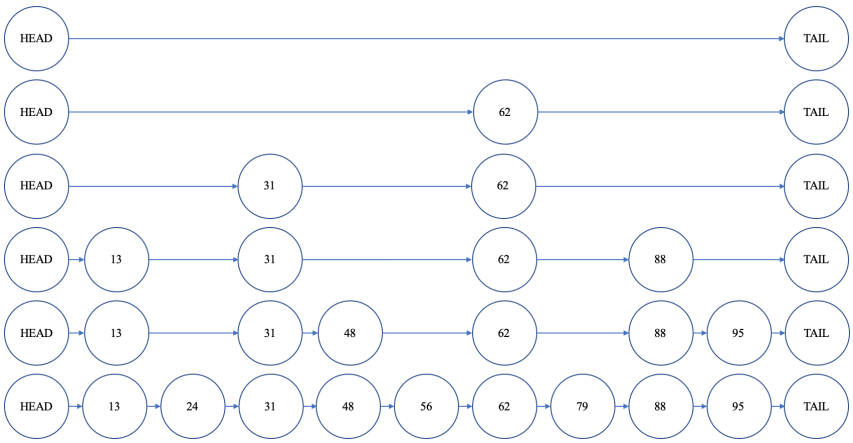

Figure 1.3: A skip list with probability $p = .5$ that a node will advance to the next level.

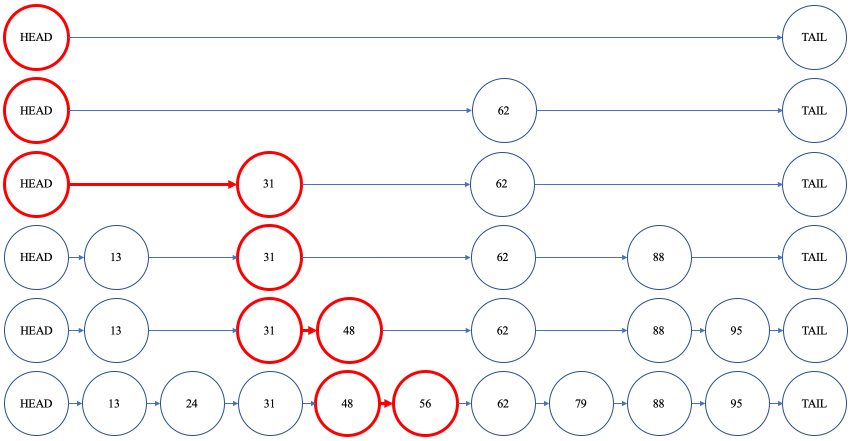

Figure 1.4: The path to search for 56 in the skip list from Fig. 1.3.

Inserting a node into a skip list entails determining the new node's maximum height, collecting a stack of predecessors and a stack of successors, and linking the inserted node to its successors and its predecessors to the inserted node. Obtaining the predecessors and successors happens by creating an array for each and filling it when searching for the node's position.

Removing a node from a skip list entails searching for the existence of that node in the structure, then physically unlinking itself from all of its predecessors and relinking the predecessors with the removed node's successors at each level.

A skip list can find a node in expected $O(\log n)$ running time. The expected maximum height of the structure is $O(\log n)$ and that the expected number of nodes traversed per level is constant by considering the worst case running time analysis of searching for `56` in Fig. 1.3. Note that the path to find `56` is demonstrated in Fig. 1.4. The analysis will hold true for what is considered to be a *perfect skip list*. That is, a skip list that will have at least one node with maximum height n within the first n nodes of the skip list. The skip list in Fig. 1.3 and Fig. 1.4 serves as an example of a perfect skip list. This is the anticipated shape of a skip list in the general case. At a high level, this makes sense intuitively. If there are a constant number of nodes traversed at each level and $O(\log n)$ levels, then the overall running time should be $O(\log n)$.

The expected number of nodes traversed per skip list level will be constant. Let `X` be a node found at level n in a skip list search. This node is either the beginning of a linked list search for a node greater than or equal to the searched node, in which the search would have just come down from an upper level of the skip list, or that there were other nodes from which have been traversed at the current linked list level. Suppose that $n = 1$, namely that searches are traversing the bottom level of the skip list.

The expected number of nodes is at most a fixed constant, 2. Consider the case in which `A` was found at an upper level. The search has searched less than 2 nodes at level n. Now consider the case in which `A` is not the first node in the current linked list traversal. As such, the node must have come from the nearest preceding node that exists at the next level up. Seeing as all nodes have a 50% chance of existing at the next level up upon creation, it would be expected that no more than one preceding node in that current linked list without being at the next level up. As such, including the current node, no more than 2 nodes to be traversed at any given level of a perfect skip list.

1.4 Concurrent Skip List

A concurrent skip list is a linearizable implementation of a skip list. As has been demonstrated, building a linearizably correct concurrent data structure is highly non-trivial, particularly when it is non-blocking. As such, it is one of the most fundamental concurrent sets and set-based maps. Most state-of-the-art concurrent sets and set-based maps use these or closely related to these algorithms.

Lifetime of a Node

The discussion of concurrent skip list algorithms will be framed by portraying the lifetime of a node. Nodes are created by a thread making a call to an `insert` method with a particular key in the event that such a key does not exist in the structure. In the event that the key is not in the structure the node is linked into the structure. This is determined by making a call to `searchRelink`, which is a search function that tracks lists of predecessors and successors. A node is discoverable after the node has been inserted. It will continue to be discoverable until it a thread calls `remove` on it. This method will set a flag that ensures that the node is logically removed from the structure, though physically linked into the structure. `remove` makes a single attempt at unlinking the nodes, but it will be later unlinked by `searchRelink` in the event that this first unlinking fails.

Search Relink

`searchRelink` performs the physical unlinking of nodes in the skip list. This operation keeps the structure clean and fills a list of predecessors and successors for insertion and removal operations to utilize. Finally, it makes a determination as to whether or not a node is present in the skip list.

Notice that unlinking a node in Alg. 3 happens in the `while`-loop block starting in line 6. In unlinking a node, the `searchRelink` function detects that the `current` node is marked at that level and will link the `previous` to the `next` node. However,

Algorithm 3 Skip List Search Relink

```
1:  procedure searchRelink(key, predecessors, successors)
2:      current = head
3:      for each level from greatest to least do
4:          while true do
5:              next = current.next
6:              while next node is marked do
7:                  next_next = first unmarked node after next
8:                  next = next_next
9:                  if previous.next is not current then
10:                     current = previous.next
11:                     if previous is marked then
12:                         goto line 2
13:                     next = current.next
14:                     continue
15:             if CAS on previous.next from current to next is false then
16:                 current = previous.next
17:                 if previous is marked then
18:                     goto line 2
19:                 next = current.next
20:                 continue
21:             if current.key >= key then
22:                 break
23:             previous = current
24:             current = next
25:         predecessors[level] = previous
26:         successors[level] = current
27:     return successors[0].key is key and successors[0] is unmarked
```

in the event that the `next` node is also marked, the previous node's new successor will still be marked and the search finds itself in the same position as before. Thus, in order to avoid performing multiple unnecessary CAS operations to relink, we advance the `next` pointer through the `next_next` pointer to find the first unmarked successor. This optimization shows strong impact in high contention testing environments. From here, we can link the `previous` to an appropriate `next` node.

In the event that the previous node is marked when unlinking a node, we restart the search from the head element of the structure because the skip list is only singly-linked. Such an operation is an expensive fallback and should be minimized if at all possible.

Insert

Inserting a node in a skip list requires obtaining a list of predecessors and a list of successors for a particular key from `searchRelink`. Nodes are constructed by the `insert` function in the event that the node doesn't exist in the structure. In line 2, of Alg. 4, the random algorithm is implemented to output a level such that the probability of an output of a node having a maximum height i is $\frac{1}{2^i}$. From there,

the node is linked to all of the predecessors and all of the successors to the node.

Lines 8 and 9 of Alg. 4 calls `setNext` and `casNext`, which are embedded in skip list nodes. Skip list nodes are implemented with each of these functions as built-ins. Skip list nodes have a `next` attribute, which is an array of maximum height `AtomicPointers`. We can use the `setNext` method to the successor in that the node is not yet discoverable. In this sense, the synchronization of `casNext` is not yet necessary. However, seeing as `casNext` changes the reference from the discoverable `previous` node, this operation must be done with `casNext` to maintain thread-safe synchronization. At upper-levels, linking to the successor must be done with `casNext` because the current node is discoverable.

Algorithm 4 Skip List Insert

```
1:  procedure INSERT(key)
2:      topLevel = random height with weighted probability
3:      while true do
4:          if searchRelink(key, predecessors, successors) then
5:              return false
6:          if toInsert == nullptr then
7:              toInsert = Node(key, topLevel)
8:          toInsert.next[0] = successors[0]
9:          if CAS on predecessors[0].next from successors[0] to toInsert is false
    then
10:             continue
11:         for each level from 1 to topLevel do
12:             while true do
13:                 repeat
14:                     oldSuccessor = toInsert.next[level]
15:                     if toInsert is marked then
16:                         return true
17:                 until CAS on toInsert.next[level] from oldSuccessor to succes-
    sors[level] is true
18:                 if CAS on predecessors[level] from successors[level] to toInsert is
    false then
19:                     toInsert.next[level] = null
20:                     return true
21:                 else
22:                     break
23:     return true
```

A node is considered to be inserted into the structure at the point at which it is successfully linked in at the bottom level. The rest of the operation is done to preserve tree semantics, which refers to logarithmic search. As such, in Line 16, there is a check to see if `toInsert` has been marked by another thread while it is trying to link itself to the successor at the upper-levels of the structure. If this is the case, the current operation is successfully terminated as the node had been inserted successfully at the bottom level because the node has been found and in the process of being removed.

When the predecessor at a level has been removed when trying to link the pre-

decessor to the current node, which is demonstrated on Line 20, the operation must restart following the same reasoning as `searchRelink`. However, we must first unlink the current node from the its next at that level.

Contains

We do not use the `searchRelink` call described in Section 1.4 to implement the call to `contains`. Seeing as the unlinking operation is expensive and that contains calls are the most frequent function calls in practice, we use a fast search algorithm called `searchNoRelink` in order to determine whether or not a node is in the skip list. This operation has $O(log(n))$ runtime and does not have the overhead of relinking nodes as does `searchRelink`. This function has an *exit-fast* property. That is, it does not need to reach the bottom level in order to make a determination if a node with a matching key is found at an upper level. Note that only unmarked nodes are traversed in line 9 of Alg. 5.

Algorithm 5 searchNoRelink

```
 1: procedure CONTAINS(key)
 2:     return searchNoRelink(key)
 3: procedure SEARCHNORELINK(key)
 4:     current = head
 5:     for each level from top to bottom do
 6:         while current.key < key do
 7:             previous = current
 8:             current = previous.next
 9:             while current is marked do
10:                 current = current.next
11:         if current.key is key and current is unmarked then
12:             return true
13:     return false
```

Remove

Removing a node in skip list can refer to *physically removing* a node, which is performed in `searchRelink` and optimistically in `remove`, and *logically removing* a node, which is the process undertaken by the `remove` function. A node is logically removed if its bottom-level reference is marked. Upper-level references are marked to notify `searchRelink` calls to unlink the node at that level.

`remove` calls `searchRelink` for the given key and marks the node's references top down. The thread that removes the node is the one that successfully places the mark on the node at the bottom level, as can be seen in line 9 of Alg. 6. Note that the `while`-loop in lines 6 and 8 are only expected to run one time each. The `while`-loops ensure that no spurious CAS failures occur and that the semantics are preserved.

Another thing to note is that an optimistic attempt is made at unlinking the node upon a successful marking of the node in the `for`-loop in Line 10. That is, an attempt at a physical removal is made immediately following a logical removal

Algorithm 6 Skip List Remove

```
1:  procedure REMOVE(key)
2:      if !searchRelink(key, predecessors, successors) then
3:          return false
4:      toRemove = successors[0]
5:      for each level from topLevel to 1 do
6:          while toRemove is unmarked do
7:              CAS on toRemove.mark[level] from false to true
8:      while toRemove is unmarked at the bottom level do
9:          if CAS on toRemove.mark[0] from false to true then
10:             for each level from 0 to topLevel do
11:                 next = toRemove.next[level]
12:                 if CAS on predecessors[level] from toRemove to next is false then
13:                     break
14:             return true
15:     return false
```

to keep the structure clean. It is more likely that this operation is successful under low-contention. Furthermore, this would be desirable in that the necessary variables to unlink the node are still accessible.

Linearizability

In inserting a node, the linearization point of having successfully inserted a node is the point at which a node is successfully linked into the bottom level. In all insertions, the function returns `true` if the script successfully performs the CAS operation in Line 9 of Alg. 4. A node is linearizable in the structure at this point. That is, all searches should be able to find this node after this point thereby making it markable and unlinkable. Otherwise, the search returns `false`.

In removing a node, the linearization point of having successfully removed a node is setting the bottom-level mark. Otherwise, the node is still in the structure and is discoverable. That is, the node can be considered a partially inserted node in the skip list, which is equivalent to a fully inserted node with a smaller `topLevel` than it was initially provided. This operation is successful if line 9 in Alg. 6 is successful and from this point forwards the function will return `true`, otherwise `false`.

If a `contains` call returns `true`, it must have been the case that the node was discoverable. That is, it was found and it was unmarked at the bottom-level in the skip list. If it returns `false`, it must have been the case that it either did not find the node at the bottom level or that we did and its bottom-level reference was marked.

Detecting `next` as being marked means that the each of its next references will be immutable because changing it would require a CAS operation to make the node is discoverable. Note that the mark is built into the reference itself, so both an expected mark and reference are required when passing an expected value into the CAS operation. However, when inserting a node, it is assumed that the predecessor is unmarked. This can be seen in line 20 of Alg. 4.

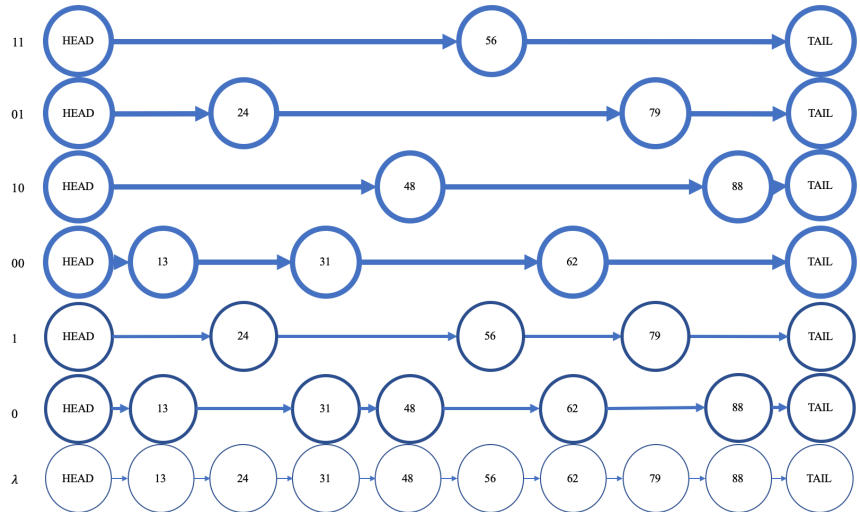

Figure 1.5: Each varying line thickness represents a different level of the skip graph with level 0 being the thinnest and level 2 being the thickest. Each list there is a partition the lower-level linked lists with corresponding lower-bit membership vector. The bottom level has one list, level one has two lists, level two has four lists.

1.5 Skip Graph

Another data structure that will be heavily relied upon for the purposes of this thesis is a skip list variant called a *skip graph*[4]. A skip graph is a collection of skip lists in which each node has a fixed height, and is randomly assigned to a particular skip list. Its structure can be viewed in Fig. 1.5. Random assignment to a skip list means that the nodes themselves are assigned to a particular *membership vector*. This vector is a series of bits defines in which linked list each node will exist in each level.

The skip graph in Fig. 1.5 is made up of four skip lists that each have three levels. They are (1) linked lists λ, 0, and 00, (2) λ, 0, and 10, (3) λ, 1, and 01, and (4) λ, 1, and 11. Each skip list is made up of a single linked list per level and nodes from each linked list are split into one of two potential linked lists at the next level up. Each level i will have 2^i linked lists. The number of lists in the upper-most level signifies how many skip lists are in the skip graph.

Each node exists in the skip graph up to `MAX_LEVEL`, but each skip list in the skip graph preserves skip list semantics. The average node height is two in any skip list within the skip graph. However, each node will be inserted up to the top level of one particular skip list of the skip graph. For instance, in a skip graph with `MAX_LEVEL` = 5, each node will have to be inserted to level 5. Note that this means that all nodes are in all skip lists at the bottom level.

The random placement into a particular skip list in the structure is determined by a node's membership vector. A membership vector is a sequence of bits particular to each node. From here, the final `MAX_LEVEL` bits are used to determine in which of the two lists at the next level up a node should exist. For example, consider skip list (2) from Fig. 1.5. Nodes in this skip list will have a membership vector of 10. All nodes in the structure are in the λ level of the skip graph. At level 1, nodes in skip list (2) will be in linked list 0 as that is the first bit of the membership vector. At level 2, nodes will be placed in linked list 10 as the next bit in the membership

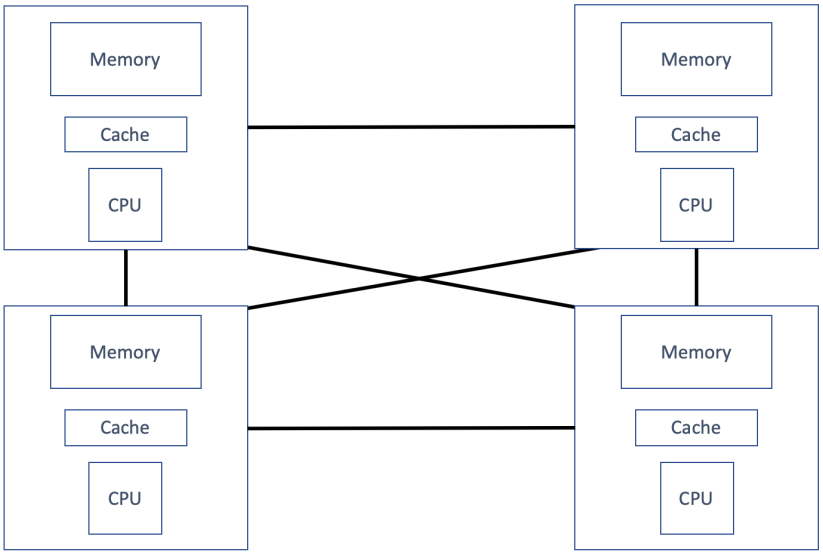

Figure 1.6: A four socket NUMA hardware architecture.

vector is 1, so it will be placed in the 1 partition in level 2 of linked list 0 at level 1. Note that the bits required to be in such a list are displayed next to the list itself in Fig. 1.5.

1.6 Non-Uniform Memory Access (NUMA)

Modern multicore computers are organized differently than the traditional memory model. Rather than using one central location of shared memory, the term "main memory" often refers to several individual nodes of memory associated with centralized clusters of CPUs. This is referred to as *Non-Uniform Memory Access* or *NUMA*. In this sense, memory is distributed.

The benefit of using a NUMA system is that it is very fast to access memory in the same NUMA node that the thread is operating in. That is, memory that is physically in the same NUMA node as the CPU should be prioritized in order to make a system *NUMA-aware*. As such, developers who look to build code that is optimized for a NUMA system look to optimize their structures so as to make more accesses to local, "closer" data than data in remote NUMA sockets. In particular, that means that threads operating from a particular CPU should most frequently access memory from a NUMA socket that corresponds to that CPU.

From a software perspective, memory in NUMA systems can be accessed in the exact same way in NUMA and uniform memory systems. CPUs operating in separate sockets traverse across inter-socket channels in order to access memory in other sockets. Accessing memory in remote sockets looks consistent in software to accessing memory in local sockets, but it is critical to update a concurrent data structure for NUMA environments to primarily access local data.

1.7 Thesis Statement

This thesis proposes a new concurrent data structure called a *layered skip graph*. The data structure is made up of a series of per-thread, thread-local sequential data structures which are *layered* on top of a novel concurrent skip list variant called a *concurrent skip graph*. This data structure satisfies three primary needs within the subfield of concurrent data structures. (1) The data structure ensures that the shared reference is small. The per-thread, thread-local data structures act as the upper index levels of the structure and, as such, the maximum level of the shared reference can be kept low. (2) The data structure must limit search sizes. By layering the thread-local data structures on top of a shared concurrent reference, threads can jump to the middle of the structure to begin their searches as opposed to always starting from the head element. (3) In the shared structure, preference should be given to nodes local to the current NUMA node. That is, if a thread is making a search in the concurrent structure it should try all possible nodes within the NUMA node that it is currently operating in before moving to another NUMA node. This is addressed in our implementation of a novel *partitioning scheme* that is integrated into the skip graph structure. The partitioning scheme partitions the data set across NUMA nodes in such a manner so that threads will first access NUMA-local data.

Chapter 2 will discuss the contributions to the research area. In particular, the chapter will discuss the contributions of layering and data partitioning, which are the contributions that I have added that benefit performance. Chapter 3 will discuss related work in the subfield of concurrent and distributed data structures research to distinguish the contributions. Chapter 4 will go into the evaluation of the experimentation which compares the performance of the layered skip graph to other state-of-the art data structures. Chapter 5 will discuss potential future work that will come out of the research that has and can be started.

Chapter 2

Contributions

While non-blocking concurrent skip lists are advantageous in that they provide a fast, linearizable concurrent algorithm with limited size and a relatively small search size, the structure fails to make any guarantees as it relates to NUMA locality. This chapter describes the layered skip graph, a new concurrent data structure made up of a series of per-thread, thread-local sequential data structures and a novel non-blocking concurrent skip list variant called a skip graph as a shared structure. It will describe how these structures look and work. The chapter will also address how the structure is capable of (1) ensuring that the concurrent data structure, the skip graph, is limited in the average maximum level of each node; (2) searching through the shared structure, the skip graph, minimizes the number of nodes traversed; and (3) how preference is given to NUMA-local nodes. Furthermore, the chapter discusses variants of the layered skip graph that further add to its advantages.

2.1 Fundamental Contributions

Lifetime of a Node

To insert a node into the layered skip graph, there must first be an attempt to insert the node into the shared skip graph structure. Inserting a node is done first by searching the structure. Searches, which are demonstrated in Fig. 2.1, start from the local structure before "jumping" to the corresponding node in the shared structure. After searching for and not finding the node, the node can be inserted into the skip graph. If this can be done successfully, the local structure will emplace the node into the local structure. As such, the local structure will only contain nodes that exist in the shared structure.

Searches in the local structure also check the "mark" of the node in the shared structure, which denotes that the shared node has been logically removed, while traversing. As such, the local structure is kept clean of removed nodes and jumps will only be made to nodes that are in the structure and smaller than or equal to the desired value.

After the point at which the node has been inserted into the layered skip graph, it is then "discoverable" until it is removed. When a node is discoverable, threads that make a contains or search call for that node should return `true`. Contains calls are the equivalent to a call to search that return `true` if the key of the node returned by the search matches the key being searched for. Removing a node from the layered

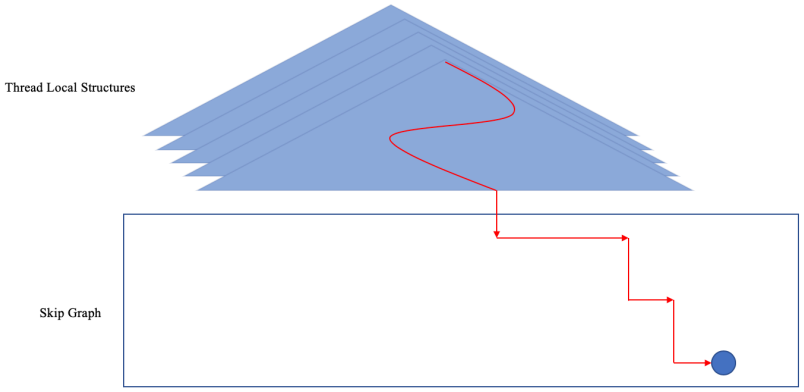

Figure 2.1: Demonstration of a search operation through the layered structure. Searches start in a thread's thread-local structure before the thread starts searching the shared structure from the closest node in its local structure.

skip graph simply entails making a call to the skip graph's remove method. If a node has been removed from the skip graph, it will be marked and will be removed from the local structure lazily through the next search call that traverses its key.

Layering

Layering refers to a data structure design technique in which a concurrent "reference" is maintained under a series of lazily maintained, thread-local upper levels to a concurrent data structures. The layered structure is made up of per-thread, thread-local sequential data structures. C++ maps combined with a fast implementation of a hash table[3] have worked well as sequential data structures in the layered skip graph. That is, by inserting into both a C++ map and a fast hash table, the benefits from very fast existence checks, from the hash table, outweighs the overhead of inserting into both the hash table and the C++ map. The thread-local maps map the actual data, which is stored as the key, to their corresponding node in the skip graph, which is stored in the value. As such, "jumps" to shared nodes requires simply following the reference of the value in the key-value pair.

Using a layering technique in concurrent data structure design helps to address the first concern; concurrent data structures should be limited in size. The thread-local structures emulate the upper-levels of the skip list in that they act as "fast-lanes" for threads to traverse through. The fact that the structure is thread-local means that it can use its most highly optimized CPU local cache to perform operations. Recall that in a concurrent environment variables must be atomic which makes them slower to update as caches must be acquired in exclusive mode. Furthermore, using thread-local data structures allows for the user to plug which ever data structure they find most optimized for their task into the structure which makes the structure highly adaptable for the user's purposes. The specifics as to the bounds of just how big the skip graph is will be discussed in greater detail in the discussion of the skip graph contribution.

Layering also addresses the second concern of concurrent data structure design; ensuring that searches are limited in the number of shared nodes they traverse. The local structures are utilized to give a good starting point to begin searches and

allow approximations of the previous node in the shared structure. Recall that, in the skip list search algorithm described in Sec. 1.4, threads traverse as far as possible in the upper-most level before moving down a level in the search. Also note that the number of nodes at the upper-most level will be inversely related to the height of the upper-most level. To demonstrate this point, consider a skip graph with maximum level n. It is true that the first $n-1$ levels of this skip graph is also, itself, a skip graph with half as many lists in the upper-most level, but that each of these lists will be twice as dense as they are at level n of the original skip graph. As such, seeing as the upper-most level must be kept low to satisfy the first requirement, limiting traversals at this level is crucial to limit the overall number of nodes traversed in the skip graph.

Additional benefits of this phenomenon are that layering disperses threads throughout the skip graph rather than forcing them to begin all searches from the head element and that threads are able quickly to approximate a previous element. Dispersion in the skip graph is advantageous in that it will limit the amount of contention for nodes along the path for a particular element. This is particularly beneficial when two threads are searching for the same node in that they will not perform CAS operations on the same data. Making approximations of the previous node is beneficial when a very specific case of CAS operation failure. Namely, when changing the reference of a `previous` node and determining that the `previous` was marked as was discussed in Sec. 1.4, approximating the preceeding node significantly minimizes the cost of this expensive case.

Concurrent Skip Graph

Developing a concurrent skip graph also required building a concurrent skip graph. The skip graph is built on-demand at compile time given the testing environment. This is done to preserve the invariant that the maximum level of a skip graph being operated on by t running threads will have a maximum level of $\log_2(t) - 1$ (there are $\log_2(t)$ total levels when counting the bottom-most level as level 0). This means that, for an environment with 8 threads, the upper-most level of the structure will be level 2. Recall that, in such a structure, there would be four linked lists in the second level of the structure. By having $\log_2(t) - 1$ levels, there will be half as many lists at the upper-most level of the skip graph as there are threads.

Recall that, in a skip graph, all nodes exist at the upper-most level of the structure. That is, if 8 threads are operating, then all nodes will exist at three levels of the structure or $\log_2(t)$ levels for t threads. In a skip list, however, nodes will exist in a constant 2 levels on average. While it is true that for most cases, the average maximum level of nodes in the skip graph will cause unwanted overhead, the other benefits from the design of the structure still proves advantageous. The sparse skip graph was developed to further combat this discrepency, and it is described in Sec. 2.2.

In a skip graph, nodes have a similar lifetime as they do in the overall layered skip graph structure. Nodes are inserted by first searching the structure to see if the desired node is in the structure. Recall from Sec. 1.4 that it is advantageous to have two distinct searching methods, one `searchRelink` that finds a node while lazily cleaning the structure of logically removed nodes and one `searchNoRelink` that leaves logically removed nodes in the structure while only searching for the desired

key. Note that these two methods can take advantage of good starting locations provided for by the thread-local data structures when the skip graph is applied to the layered skip graph. In an insertion, `searchRelink` is used. After the search, nodes are linked to their successor and predecessor nodes similar to the insertion in Sec. 1.4.

After a node is linked into the structure at the bottom-most level, that node is discoverable. That is, it can be found by calls to contains or searches until it is removed[1]. Calls to contains are made with the faster searching method, `searchNoRelink`. To remove a node, the node is to be found by a `searchRelink` call. Afterwards, the node is logically removed at the point at which the mark is set at the bottom-most level. At that point, an optimistic attempt at unlinking the node is made and the node can be lazily unlinked by search methods.

Full algorithmic details can be found in Appendix B.

This thesis does not provide a formal proof that the non-blocking concurrent skip graph is linearizable. With that said, here is an intuition that it is. Seeing as a skip graph is a collection of embedded skip lists, pick the skip list that is currently being operated on by a set of operations and a set of threads. Now, apply any and all arguments of linearizablity from the non-blocking concurrent skip list to this skip list. This same argumentation can be applied to all skip lists in the skip graph, and as such the skip graph must be linearizably correct.

Partitioning Scheme

The partitioning scheme applied to the concurrent skip graph allows for the dataset being added to the data structure to be partitioned stratgically to produce NUMA-locality. The broad idea is to monitor which threads can operate on which skip lists within the overall skip graph. By doing so, each skip list within the skip graph will only be shared by 2 threads, while lower levels are shared by increasingly more threads. That is, for each level i from 0 to the maximum level of the skip graph, there will be $t/2^i$ threads operating in each list within that level.

Each node in the skip graph will have a membership vector that is defined to be a `uint64_t` with the lower-order bits being filled the reversed `threadId` and the higher-order bits filled randomly. That is, a node that is inserted by `threadId` = 1 will have a membership vector of 61 random bits followed by `100` if maximum level is 2. Note that this would place nodes into skip list (1) from Fig. 1.5 along with nodes inserted by `threadId` = 0. The membership vector for nodes inserted by `threadId` = 0 would have a membership vector of 61 random bits followed by `000`.

Note that this design ensures that only two threads will operate in the upper-most level of any particular skip list within the skip graph. That is, the data will be partitioned across the skip lists within the skip graph by the thread that inserted that data. At the next level down, the linked list will be shared by four threads and so on. As such, more contention is introduced by traversing in lower levels of the skip graph because the linked list that is traversed will be shared by more skip lists and, as a result, more threads.

[1]Notice that this means that nodes could be removed before the node is fully linked into the structure. A node is only inserted into a thread's thread-local data structure if the node is fully linked into the shared structure in the layered skip graph.

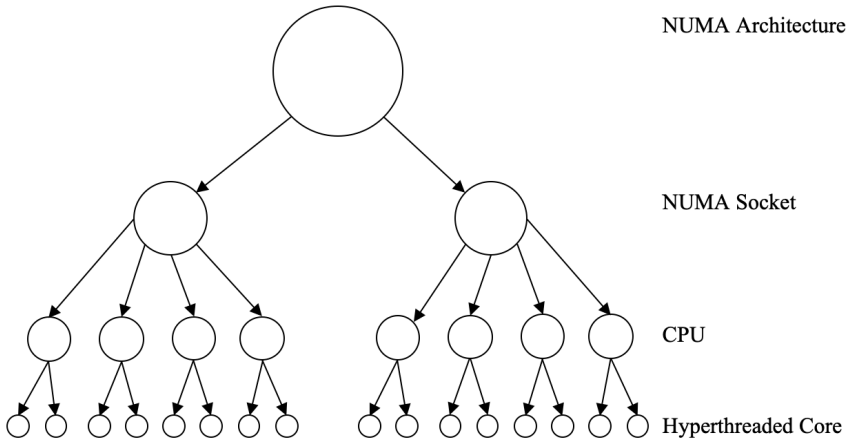

Figure 2.2: The following is a representation of a hierarchical NUMA architecture with 2 NUMA nodes, 8 CPUs and 16 hyperthreaded cores.

This organization of a skip graph allows for strategic data *partitioning*. That is, storing the data from the data structure in a strategic location in hardware. Suppose a NUMA machine supports *hyperthreading*; hyperthreading is a type of CPU that has two instruction pointers and can, thus, support two threads. Recall that NUMA means that each socket has its own associated memory. By ensuring that all data in the uppermost level of the structure is in the same location in memory, the structure is *NUMA-aware*. That is, threads allocating memory from a particular CPU will allocate memory within that CPU's associated memory.

The current state of NUMA technology is commonly separating distinctive NUMA sockets, each with their own CPUs that have hyperthreaded cores. Creating a hierarchical scheme for partitioning data and where threads will operate on data is currently optimal for this architecture. It is also highly adaptable to other potential hierarchical architectures that may be designed in the future, which is key in successful systems design. For example, the structure would still have an optimal memory-access in the event that hyperthreading technology were to progress to the point at which it is more common to have more than two hyperthreaded cores per CPU if the maximum level of the structure is changed. Alternatively, consider the fact that memory were to be further distributed beyond the original splitting of NUMA nodes. The structure can be configured to still access the closest memory most frequently.

The thread pinning strategy which obtains optimal performance comes from pinning threads that share the upper-most level of the skip graph to hyperthreaded cores in the same CPU, *Partition 1*. Partition 1 pins ensures that the two threads that share the uppermost level in the skip graph will be pinned to CPUs that are physically closest to one another. That is, seeing as Thread 0 and Thread 1 share the upper-most level of the skip graph, they will be placed in adjacent CPUs. This is the default configuration. This thesis demonstrates that there are several other partitioning schemes to demonstrate the benefit of Partition 1. They are as follows:

1. Partition 1: Threads that share their upper-most level in the skip graph are placed in adjacent CPUs.

2. Partition 2: Threads that share their upper-most level in the skip graph are placed in the same NUMA node, but in not adjacent CPUs.

3. Partition 3: Threads are pinned to random CPUs.

4. Partition 4: The opposite of Partition 1. Threads are pinned to corresponding CPUs in the furthest NUMA node. This partition is intentionally bad to demonstrate the effects of cross NUMA node traversals.

5. Partition 5: Threads are not pinned to CPUs.

The partitioning scheme on which a skip graph runs is determined at compile time. As such, the data structure is easily adaptable to any particular machine to optimize its pinning strategy.

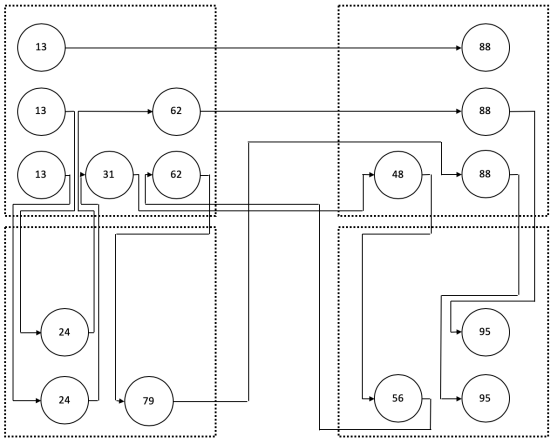

Figure 2.3: A skip list stored in memory in a NUMA machine.

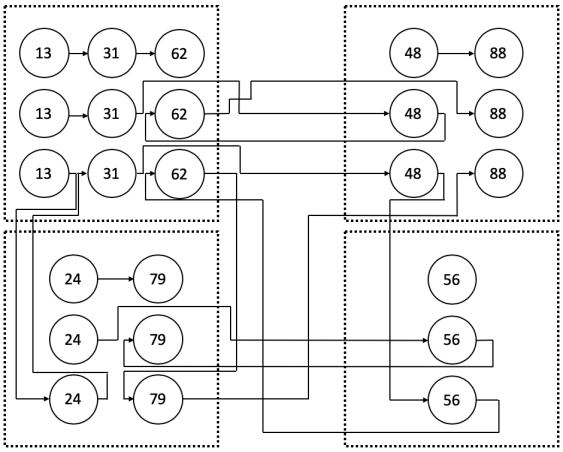

Figure 2.4: A skip graph with the Partition 1 partitioning scheme stored in memory in a NUMA machine.

As a result of the partitioning scheme, it is trivial to know how a skip list will be stored in memory as compared to a skip graph. This can be viewed in Fig. 2.3 and 2.4. In particular, the lack of locality in a skip list can be noticed in Fig. 2.3. Notice how, in the first level of the skip list, the nodes are 13, 24, 62, 88, and 95. To traverse across this level, a thread must travel from the top left socket to the bottom left socket, the bottom left socket to the top left socket, the top left socket

to the top right socket, and from the top right socket to the bottom right socket. There is no ordering nor restrictions to where a thread might have to travel within the system in order to find a node.

In the skip graph from Fig. 2.4, however, notice that there is an ordering to where these traversals can be made. At the upper-most level, where nodes are shared by two threads, threads can only ever traverse within that particular NUMA socket. At the next level down, threads can only traverse with the socket to which they are horizontally aligned. It is only at the bottom-most level where threads can traverse to any point in memory in the system.

Note that threads do the fewest traversals at the bottom-most level, thereby making such traversals the most rare. Also notice that there is no reason why, at the middle level, threads necessarily traverse horizontally. As such, the partitioning scheme can be configured to pin threads to NUMA sockets with the closest corresponding socket to maximize the hardware efficiency of the NUMA system. As such, the partitioning scheme effectively accomplishes the third goal of concurrent data structure design - that local nodes are given preference - is satisfied.

2.2 Variant Contributions

Sparse Skip Graph

The *sparse skip graph* was developed to mitigate the negative performance effects of the skip graph. This structure is built the same way as a skip graph, but each node is given a random node maximum height as is done in a skip list. As such, the skip graph is significantly more spare than a normal skip graph. More importantly, The same guarantee can be made about the average maximum height of a node in the sparse skip graph that we can make in the skip list. The sparse skip graph can be viewed in Fig. 2.5.

Designing this skip list and skip graph variant is motivated by the overhead of performing insertions and removals in skip graphs as opposed to skip lists. Inserting or unlinking a node has significantly more overhead in the skip graph than it does in the skip list. Furthermore, skip graphs are much denser structures, so creating a skip graph with a maximum level of 5 is much memory intensive than a skip list with a maximum level of 5. However, skip lists do not provide the opportunity to use a membership vector.

Lazy Layered Skip Graph

Throughout the research process, it became clear that most state-of-the-art non-blocking concurrent skip lists took advantage of a similar lazy pattern. That is, nodes would be inserted at the bottom level of the structure only and some external lazy mechanism would later "raise the towers" of each node. Many times, this has been done through the use of a background thread (see Chapter 3). Time and time again, it has been demonstrated that such techniques are incredibly effective under high contention, but tend not to scale with the same effect for low contention tests. This was the inspiration for designing a lazy version of the layered skip graph.

Nodes have nearly same definition in the lazy layered skip graph as they do in the layered skip graph. However, nodes are given an additional flag that is used

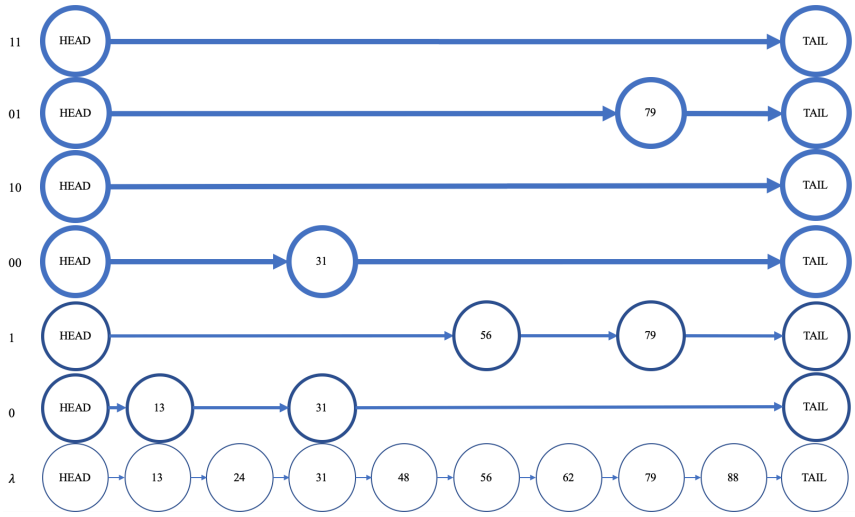

Figure 2.5: The following is a representation of a sparse skip graph. Note that it is both partitioned and that nodes have an average maximum height of two.

for *flip operations* in the lazy layered skip graph. Flip operations are insertions and removals on a node without setting the mark. Instead, insertions or removals modify the additional flag on nodes with the same key that are in the structure so that the node will never be physically removed as a result of a flip operation.

As such, the lifetime of a node in the lazy layered skip graph looks different than it does in the layered skip graph. To insert a node into the lazy layered skip graph, a thread must search to see if the node is in the structure. This operation is identical to the layered skip graph. However, upon determining that the node does not exist in the structure, the node will then only be inserted into the bottom level of the skip graph and the thread's thread-local structure. At this point, the node is discoverable. Contains calls are done the same in both the lazy layered skip graph and the layered skip graph.

When the node is discoverable, it can be removed. Removals in the lazy layered skip graph work by setting the additional flag on the node from `false` to `true`. At this point, the node is removed from the structure, but the same physical node can be once again inserted without allocating any new memory. Other insertions can come through and find the node in the structure and "flagged," at which point they flip the flag back from `true` to `false` and the node is once again inserted in the structure. This process can happen several times.

The mark on the node can still be set. This happens when a node has been discovered by a thread searching for a node from a skip graph remove method call after the node has been in the structure for a pre-specified period of aging. After a the node's mark has been set, it can no longer be flagged and it can be unlinked. Any further insertions to this key will once again have to allocate memory.

A node's towers can also be lazily raised. This is done when the node is discovered to be unmarked in the skip graph by a search from the local structure, even if the node is flagged in the skip graph. In the event that this happens, the thread will attempt to link the node to its predecessors and successors in the skip graph as has been described in Sec. 2.1. Notice that only the thread that initially inserted the node will be able to raise its towers as only that thread will have added that node to its thread-local data structures.

As such, this thesis will be working with the following data structures:

	Skip Graph	Sparse Skip Graph
Traditional	Layered Skip Graph	Layered Sparse Skip Graph
Lazy	Lazy Layered Skip Graph	

Notice that there is no Lazy Layered Sparse Skip Graph implementation. As it turns out, making such an algorithm has several small design decisions that were crucial for its correctness. Given more time, this is an area of future study.

2.3 Applications

The ideas discussed in Sec. 2.1 and Sec. 2.2 each can be applied into different data structures in practice. In particular, this thesis will look at how each can implement maps and priority queues.

Maps

Notice that the algorithms described in the table at the end of Sec. 2.2 all implement a set or map. That is, nodes in the skip graph or variant structure contain a key and a value and can store data as such. To implement a set, a user would simply not pass a value.

Priority Queues

The question of building concurrent, scalable priority queues has been considered by many to be an irreconcilable problem in concurrent computing. A priority queue is a data structure that removes the absolute minimum element in the structure. In a concurrent priority queue, there can only be one linearizably minimum element in a particular key space at any particular moment in time. This means that only one thread out of t can correctly remove the absolute minimum node at a time if t threads attempt to remove the absolute minimum while $t - 1$ threads would fail.

Several approaches have been proposed to mitigate this problem, such as relaxing the demands of linearizability or the semantics of removing the linearizably absolute minimum node. This thesis considers the approach of relaxing priority queue algorithms utilizing skip lists as the primary backbone of their architecture. In particular, it examines how the layered skip graph model can provide improved performance. In particular, the discussion will be focused on the Spray List [1].

In order to best counteract the scalability paradox in concurrent priority queues, the Spray List introduces a random walk at each level in order to "spray" threads across the data structure to a bounded "minimum." By doing so, the Spray List approximates the absolute minimum rather than ensuring that the linearizable absolute minimum is removed. There is a trade-off to be had with proximity of a node to the absolute minimum and the contention for an element. That is, the proximity of the largest "minimum" node that threads are allowed to remove to the true absolute minimum is inversely related to the amount of contention. If threads must remove the true absolute minimum, there is the most contention for the "minimum" element and if there is no constraint on the "minimum" element (i.e. threads can

remove any node in the structure) there is the least contention. The goal for creating a scalable implementation of a concurrent priority queue is to both minimize the potential element space in which the node can be found as well as creating an algorithm that scalably increases in performance as thread counts increase.

The Spray List's contribution comes from the bound for the largest possible absolute minimum element that could possibly be removed by their spraying technique, which they define to be $\Theta(t \log t)$. The goal of my research is two-fold when considering concurrent relaxed priority queues. It would be an improvement from the Spray List by (1) making the same guarantees of maximum possible removed element with a scalably better overall performance or by (2) creating a bound closer to the linearizable absolute minimum element while still scaling as in performance at higher thread counts.

Algorithm 7 Spray List RemoveMin - Skip List

```
1:  procedure RemoveMin
2:      Pick random number [1, t] where t is the number of threads
3:      if Random number is threadId then
4:          ▷ Cleaning policy described in [1]
5:          Clean
6:      while true do
7:          level = MAX_LEVEL
8:          while level >= 0 do
9:              r = random number [0, log t]
10:             Walk r unmarked nodes
11:             level -= max(1, ⌊log t⌋)
12:         if node is tail then
13:             return false
14:         if node.mark() then
15:             return true
```

First, this thesis will consider the goal of attempting to improve the overall performance of the Spray List while preserving the same guaranteed maximum bound. In order to do this, it is necessary to demonstrate how the analysis can hold. First, it is important to note that such a bound can hold for a skip list. This point has been proven in the Technical Report associated with the Spray List [16]. Seeing as the skip graph is a series of interwoven skip lists, any analysis made on the Spray List holds for the structure because threads operate on a single skip list of the skip graph during its execution. A formal theorem for this claim is made in [19]. The description in this thesis will provide an intuition of said theorems and associated proofs.

Recall that the skip lists in the skip graph are both NUMA-aware and minimize contention between threads at upper-levels of the structure as a result of the partitioning scheme employed in the skip graph. Furthermore, note that threads are operating on $t/2$ different skip lists at the upper-most level. As a result, the expectation is that the Spray List algorithm applied to this structure will achieve even greater dispersion throughout the structure with the same bounds in place because threads are operating on entirely different skip lists. The anticipation would be that the structure would significantly outperform the Spray List algorithm applied to a

skip list.

Algorithm 8 Spray List - Skip Graph

```
1:  procedure RemoveMin
2:      Pick random number [0, t − 1] where t is the number of threads
3:      if Random number is threadId then
4:          ▷ Cleaning policy described in [1]
5:          Clean
6:      while true do
7:          level = MAX_LEVEL
8:          while level >= 0 do
9:              r = randomly pick zero or one
10:             find r-th unmarked node
11:             level -= max(1, ⌊log t⌋)
12:         if node is tail then
13:             return false
14:         if node.mark() then
15:             return true
16:         else
17:             traverse forwards one unmarked node
18:             if node.mark() then
19:                 return true
```

By making a few key modifications to the algorithm proposed by the Spray List, which can be seen in Alg. 8, this thesis claims that threads will not need to spray as far to obtain a node to remove while still maintaining scalable performance. The formal proof, again, can be found in [19].

Instead of spraying for a random number of nodes `r` as is done in line 9 of Alg. 7, the algorithm will instead merely find the first or second unmarked node, which is decided randomly, and try to mark nodes "along the way." This means that a thread optimistically attempt to mark each node that it finds. Only upon failing to mark that node will a thread go down a level and try again. At the bottom-most level, threads traverse forwards an additional node if they fail so as to avoid having to restart the entire spray.

Note that threads will only contend with one other thread at each level of a perfect skip graph structure. That is, when a thread makes an optimistic attempt at removing a node at the uppermost level, it will contend with one other thread at most. At the next level down, the thread will share the list with three other threads, but seeing as one thread has already won a node at the uppermost level of each partition, the two remaining threads will only compete with one another for the node at the next level down. This is demonstrated in Fig. 2.6.

A perfect skip graph is defined as a skip graph in which every $t/2$ nodes are put in different skip lists of the skip graph. In a perfect skip graph with the NUMA-aware partitioning scheme, each thread will contend with one other thread for the minimum node in the upper-most level of their skip list. Seeing as one thread will succeed at this level, only half the threads will move down a level to contend for the smallest unmarked element at that level. As such, this level will once again be contended for by no more than two threads. This process will repeat itself until the

bottom-most level where, upon failing, the single remaining thread will once again move forwards to the next unmarked node and it will have its solution.

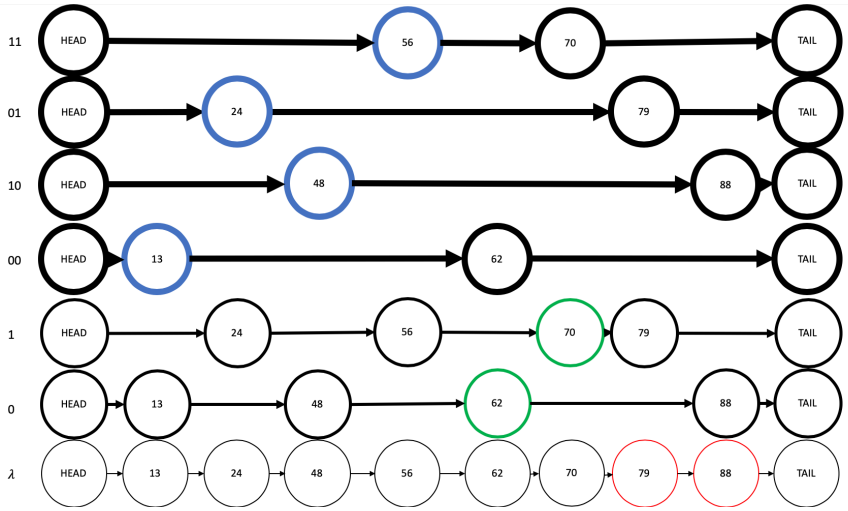

Figure 2.6: The pattern by which threads would remove nodes if all 8 threads were removing. Note that one thread will win and one thread will lose contended nodes.

Such a policy will only pay off in the event that the optimistic marking of nodes does not devolve into the initial priority queue paradox of having $t-1$ out of t threads fail. In a concurrent data structure that does not have the NUMA-aware partitioning scheme, like a skip list, these guarantees about reduced contention cannot be made. As such, such a policy is unique to the NUMA-aware partitioning scheme and fully takes advantage of the structure of the skip graph.

Recall once again that using the partitioned skip graph structure already gives threads much of the dispersion that they obtain from the skip list-based spraying. This means that making a traversal of size `L` to spray through the structure is not as necessary as it might have been in a traditional skip list-based structure. Instead, threads can simply traverse to the first unmarked node of whatever level they are currently opearting and attempt to mark that node while contending with at most one other thread in a perfect skip graph environment while failing $\log(t)$ times at most and twice in expectation at all levels above the bottom-most level and traverse one more time upon failing at the bottom-most level. This means that, in analyzing the structure under perfect conditions, the structure will remove one of the first t nodes with t threads executing.

Furthermore, the skip graph variant of the algorithm minimizes the number of CAS operations needed per successful spray operations. The reason for this, again, is explained in [19]. Essentially, the CAS operations performed in the Spray List algorithm reduces to a "balls-in-bins" problem, making the expected number of CAS operations per removal logarithmic. However, by attempting to mark nodes along the way, the variant algorithm performs a constant two CAS operations on expectation per operation.

Chapter 3

Related Work

At this point, this thesis will discuss related work in the field of concurrent skip lists. The goal of this section is to highlight major contributions in the areas of concurrent data structure design, concurrent priority queues, and developments in NUMA-awareness. This section will also highlight some of the distinctions between the work of other groups from the work proposed in this thesis.

3.1 Skip Lists

Skip lists were first introduced by [17]. Modern state-of-the-art lock-based skip lists are variants of [15]. The first lock-free skip lists were introduced and patented by [18]. The algorithms of this skip list are those that are discussed in Sec. 1.4. Skip graphs were introduced in [4]. Somem of the best performing state-of-the-art lazy skip list algorithms, which tend to perform very well under high contention, are described in greater detail below.

No Hotspot

No Hotspot [7] is a state-of-the-art skip list implementation that proposes the decoupling of tasks in insertion and removal operations. In particular, this means that inserting a node means simply inserting a node into bottom level of a skip list before returning. From here, a background running thread is tasked with "raising the towers," or inserting nodes at levels above the bottom-most level of the structure, of all nodes and fills in the upper levels of the structure entirely contention-free. That is, the threads running in the structure do not "raise the towers" of the skip list to simplify their task. In removing a node, running threads are pegged with the task of setting the mark of the node and then, again, relying on a background thread to physically remove marked nodes.

No Hotspot demonstrates particularly effective performance in a high-contention environment. That is, the structure performs well in write-intensive environments with a high-collision key space. In such an environment, the task of raising towers and physically removing nodes from the structure is manageable as a result of the key space. However, it is anticipated that such a structure will perform poorly in low-contention environments for two reason. Namely, (1) the task of raising towers is spread across a larger space making the task a lot larger and (2) the fact that the

task of the background thread is larger means that it will be harder to preserve skip list semantics.

Inspired by this implementation of a skip list, a variant of the layered skip graph called the *lazy layered skip graph* was built in the research leading to this thesis. The lazy layered skip graph is particularly well-suited for high-contention environments and takes advantage of the idea of lazily raising node towers while still maintaining NUMA-aware skip graph data partitioning and thread-local layered data structures. However, the implementation described in this thesis is different than that of the No Hotspot skip list in that a background thread is not used. Instead, the lazy layered skip graph's towers are raised in the layered interface when searching the local data structure. In searching for a node in the local structure, a node's towers are raised if it's found to be not fully inserted in the shared structure. Full implementation details are provided in Appendix B.

Rotating Skip List

The Rotating Skip List [11] is a state-of-the-art skip list made up of "wheels" as opposed to nodes to minimize contention. Wheels are another way to consider a series of next pointers, but the bottom level of the structure is fluid. There is a counter from 0 to the maximum level of the structure that reset when overflowing the maximum level. If node i has a maximum level of n_i, then for any counter value c, the current bottom level of the node is $c\%n_i$ and all pointers will follow that pattern. From time to time, a background thread will increment this counter, which will mean that threads will have different positions in the structure. This gives a means of dispersing threads to reduce contention.

Wheels are cache efficient and give the effect of rotating trees in a sequential environment that self balance. The utilization of wheels means that entire linked lists of the structure can be temporarily removed from the structure in constant time. This allows for synchronization within that linked list by freezing its state and allowing for a single thread to update it contention-free.

A background thread exhibits similar behavior to that of the background thread in No Hotspot. In both structures, the background thread is tasked with raising the towers of a node. In the context of the Rotating Skip List, this refers to the cache-efficient wheels. A key difference between the background thread in the Rotating Skip List is that it operates lazily to minimize contention. That is, it will operate maintenance operations on all data after sleeping for a fixed unit of time to minimize contention with active threads in the structure.

The strategy of rotating levels in and out of a structure will largely benefit a structure in which a background thread maintains nodes within a structure. As such, the idea of rotating does not directly pertain to the structure described in this thesis as background threads for data maintenance are not employed. However, the application of a grace period before nodes have their towers raised in the lazy layered skip graph implementation is considered.

The layered skip graph and related structures do not adopt the practice of rotating pointers. For one, this design adds lots of complexity to the overall structure and is not typically adopted. Furthermore, the skip graph already disperse threads throughout the structure without this complexity. However, the practices of building cache efficient skip list structures was highly influential in how data is aligned

within nodes and across nodes in the layered skip graph structure.

3.2 Priority Queue

This thesis has given extensive discussion of [1] in Sec. 2.3. The Spray List provides one means of developing concurrent priority queues by relaxing the definition of absolute minimum. In so doing, contention is minimized by allowing multiple threads to successfully remove different "minimum" nodes. This thesis proposes a means of relaxing the definition of absolute minimum that (1) out-performs the Spray List and (2) removes elements that are closer to the absolute minimum.

Another approach that effectively works to minimize the contention in a priority queue with a relaxed absolute minimum is through an elimination array [5]. The technique works by having a producer-consumer array where an inserting thread adds nodes that are "close" to the absolute minimum for removing threads to consume. This works well, as the task of performing CAS operations to link the node into the structure are eliminated, and the node will never need to be unlinked. Such a situation would linearize in the following way: (1) the inserting thread inserts an element into the structure; (2) the removing thread instantaneously finds the node and removes it. The issue with this technique is that determining the cutoff between "close" and "far" nodes is computationally expensive to ensure that the protocol is utilized enough to scale. This approach was adopted in the research leading to this thesis with promising results, as the layered skip graph can be used to approximate this barrier, though the contribution could not be stated clearly.

There are other ways of minimizing contention for the absolute minimum element in building a concurrent priority queue. In fact, several techniques were considered in the research for this thesis. The technique of flat-combining was proposed by [13]. The idea is to minimize the contention for the head element by filtering threads through a tournament so that only on thread would ever remove nodes at a time. This is beneficial in that it means the true absolute minimum will be removed by a single thread. However, two glaring issues made this technique difficult to implement in practice. For one, the protocol is blocking, so threads that are not actively removing nodes have nothing to do, which wastes CPU resources on those threads and hurts overall performance. It is possible that threads could "clean" the structure while waiting, or unlink logically marked nodes, but it turns out that this pushes the issue of contention onto the task of cleaning up the structure. There are no other obvious tasks to be done while waiting. The other issue comes from the scalability of a single thread needing to altrusitically serve $n - 1$ threads while making the structure inaccessible to any other thread while the serving thread is operating. Resolving this issue adds complexity and does not necessarily guarantee scalability[1].

Another approach to mitigate the issue of building a concurrent priority queue with minimal contention for the absolute minimum element is to relax the definitions of correctness. In [14], Chapter 3 discusses alternative concurrent correctness properties with more relaxed definitions that linearizability. It may be possible to develop a concurrent priority queue that is quiescently consistent or sequentially consistent, as is described in the chapter, and correctly scale while removing the

[1]With that said, this is an area of potential future work as early as Summer 2020.

absolute minimum element. However, such an implementation is not trivial in how it would even look, and was not seriously pursued in the research leading to this thesis.

3.3 NUMA

NUMA-aware concurrency has spanned across several areas of development. NUMA-aware synchronization mechanisms, like the locks proposed in [6] and [10], ensure that priority is given to waiting threads based on their NUMA distance to the lock itself. The development of NUMA-aware skip list (NUMASK) was first done in [8] and is discussed in greater detail below.

NUMASK

NUMASK is a state-of-the-art NUMA-aware skip list. It was produced around the same time as the layered skip graph. NUMASK builds a skip list in which the bottom level is separated across NUMA nodes. However, each NUMA node also has an "intermediate layer," which a complete replica of the bottom level of the skip list. Similar to the No Hotspot skip list, a background thread observes insertions to the bottom level of the skip list and physically unlinks nodes. It notifies each intermediate layer of insertions by pushing instructions onto a per-intermediate-layer queue. Each intermediate-layer-queue is maintained by a per-NUMA-node background thread which updates the intermediate layer for its respective NUMA-node. After doing so, the per-NUMA-node background threads lazily raise the towers within its NUMA node in a manner inspired by the No Hotspot skip list.

As a result of background threads lazily raising node towers, NUMASK exhibits similar performance to No Hotspot. However, the architecture is NUMA-aware and exhibits certain advantages for high-performance computing in NUMA. The similarities suggest that NUMASK will perform particularly well in high-contention environments, but its advantages will diminish in lower contention testing environments.

Though developed at similar times, the idea of a data structure local to a particular NUMA node was not necessarily inspired by NUMA. This is evident in the differences between the two architectures. In order to maintain its semantics, NUMASK replicates data from the upper-levels of the skip list and the intermediate layer throughout each of the NUMA-nodes. As such, it is much more space intensive. By using a skip graph as opposed to a skip list, nodes are only inserted once in the overall structure and data replication across NUMA-nodes is not necessary in order to preserve NUMA-aware semantics.

Furthermore, the local data structure in NUMASK is the upper levels of the overall skip list, which is itself a skip list. In this sense, NUMASK would the equivalent of layering the same thread-local skip list in the structure proposed by this thesis on top of a concurrent linked list with intermediate layers for each NUMA-node. The structure proposed by this thesis is more adaptable in that it allows for the upper levels of the skip list to be represented by faster data structures, which allows for faster searches in ultimately accessing the true representation of data in the shared bottom-level.

Chapter 4

Evaluation

4.1 Testing Procedures

Tests are run on two different hardware architectures. One, which will be referred to from here forwards as the "local cluster," has 2 NUMA sockets, 16 Intel Xeon E5-2620 cores, which have hyperthreading capabilities thereby giving us access to 32 hardware threads, each running at 2.0-2.5GHz, which varies due to TurboBoosting, and 128GB of memory. The other, which will refer to as the "remote cluster," is a system with 2 Intel Xeon Platinum 8275CL CPUs, each with 24 cores running at 3.0GHz (96 hardware threads total). The system has 192GB of memory and two NUMA nodes. The tool numactl –hardware reports intra-node distances of 10 and internode distances of 21. The system runs Ubuntu Linux 18.04 LTS with kernel 4.15.0.

The methods described in this thesis use thread pinning to the advantage of the structure. *Thread pinning* refers to the practice of ensuring that a single thread operates on only one CPU at a time. Thread pinning can produce stronger performance results in that it ensures a level of predictability of a thread's memory and makes better use of local CPU caches as threads do not migrate across the system. It makes sense to utilize thread pinning in concurrent programming for NUMA because the policy ensures that threads do not migrate across NUMA sockets and make slower accesses to memory inserted by that thread, which would be advantageous for the overall application as a whole.

4.2 Testing Environment

The code must be compiled with `clang++-std=c++14-O3`, which is the highest level of optimization that can be passed to the Linux compiler. From here, the output executable is run with several arguments. Namely and in order they are the following: `number_threads`, `protocol`, `percent_insert`, `percent_remove`, `percent_contains`, `preload`, `element_space`, `number_of_operations`, `time`, and the "architecture string" (optional if `__linux__` is not defined).

This thesis proposes five potential options for the `protocol` token. It can be `operations`, `clustered_operations`, `time`, `clustered_time`, or `synchrobench`. The options `clustered_{operations,time}` mean that threads will be tasked with a particular operation and run on that operation only. That is, if a thread is tasked with `insert`, then it will only call the `insert` method, etc…. The options

`operations` and `time` mean that threads will run for a preselected number of operations or for a preset amount of time. After considering the advice of conference reviewers, it was determined that most results will be run through the `synchrobench` [12] protocol seeing as it is the uniform industry standard.

The "architecture string" is made up of the number of NUMA sockets; a comma; the number of cores in each of the sockets delimited by commas; a dollar sign as a delimiter; and the core id to which a thread i will belong if i is the index of the cores, which are delimited by pipes. For instance, on our local cluster, the architecture string for *Partition 1* is the following: `2,16,16$0|16|1|17|2|18|3|19|4|20|5|21|6|22|7|23|8|24|9|25|10|26|11|27|12|28|13|29|14|30|15|31|`. In this case, thread with id 0 will run on CPU id 0, thread id 1 will run on CPU id 16, etc…. This string is created on demand using the Linux built-in file `/proc/cpu_info` by a Python 3 script called `cat.py`. It is through this string that a hardware-aware architecture can be built to help with the overall performance of the structure.

The tests are run through a main file called `Tester.hpp` in the repository. This script reads through the arguments to assign global testing variables, creates threads and their corresponding thread function, parses through the "CPU Info" argument to set thread affinity to a particular CPU, runs and joins threads.

Thread functions track several counting variables that are used for the analysis. In particular, each thread counts the success or failure of each function and the totals are accumulated at the end. This is done so as to avoid paying to write to a synchronized variable while the thread is running so as to ensure the most parallelism between threads as possible[1]. These values are aggregated into a global atomic counting variable after threads are joined to ensure thread safety. These variables allow for assessments about overall running performance.

The infrastructure in place is designed to keep a much more in-depth analysis about the performance of the structure if the compiler directive `STATS` is defined. However, it is important to note that only make per operation assessments can be made when running and counting these metrics. The additional counting metrics add a significant amount of overhead to the overall running time of any particular operation. Such metrics include nodes per search and locality metrics.

4.3 Performance

Overall throughput is calculated by measuring the number of operations per millisecond. An operation is considered to be the completion of a call to any of the `insert`, `remove`, or `contains` functions. The results for tests in high contention, medium contention, and low contention testing environments are shown in Fig. 4.4, Fig. 4.5, and Fig. 4.6 respectively.

Fig. 4.4 and 4.5 shows the benefits of structures that lazily raise their reference towers with background threads. Such structures perform particularly well under high contention environments. In particular, the lazy layered skip graph performs well in high and medium contention environments as it replicates the effect of lazily raising reference towers. As was anticipated in Sec. 2.1, using a NUMA-aware data

[1]Note that only count the time it takes for threads to perform operations is counted so as not to confound background operations with true performance, but as much parallelism as possible is desirable so as to emulate real contention scenarios as frequently as possible.

partitioning scheme gives the structure a massive boost in performance over its competitors in NUMA. Furthermore, it is anticipated that shared nodes are found particularly fast in smaller key space testing environments in that more nodes will be found immediately in the thread-local hash table used to index shared nodes from the layered interface.

Notice that Fig. 4.1 just shows the layering technique on top of the skip graph, sparse skip graph, and the skip list. Note that the skip graph and the sparse skip graph both apply the partitioning scheme whereas the skip list does not. Otherwise, the structures use the exact same layering technique, the exact same thread-local data structures, and are run through the exact same testing protocols on the exact same machine. The only difference is the application of the partitioning scheme. Also note that our thread pinning strategy will pin threads to CPUs within the same node in NUMA as long as possible. The effects of threads being pinned to separate NUMA nodes can first be seen at 64 threads. The partitioning scheme applied to the skip graph and sparse skip graph allow for the performance to continue to scale where the skip list does not. Herein lies demonstable proof of the benefits of the partitioning scheme.

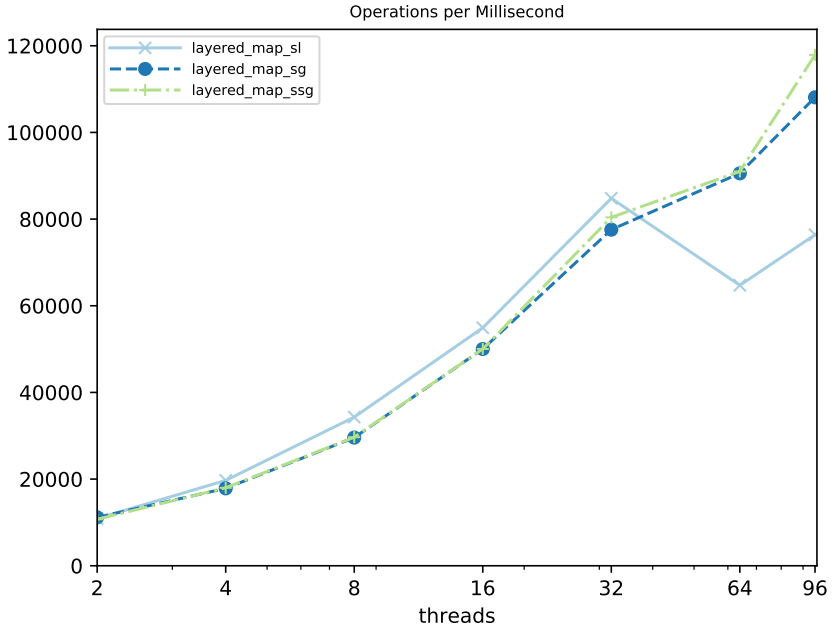

Figure 4.1: Medium-Contention, Write-Heavy Workload. 32% effective updates.

Fig. 4.2 shows two structures, the skip graph and the layered skip graph, to show the effect of layering. As was demonstrated above, these two tests are identical except for the variable of layering thread local-data structures on top of the skip graph or not. This technique demonstrates demonstrable benefits irrespective of NUMA. That is, the layered skip graph significantly out performs the skip graph as early as 2 threads. However, also note that the impact of introducing NUMA does not impact the layered skip graph as much as it does the skip graph. That is because the thread-local data structures are also stored in the memory associated with each thread's CPU in NUMA in the layered structure, whereas the skip graph does not benefit from such locality.

Note that, in Fig. 4.3, the layering technique on top of a linked list equals

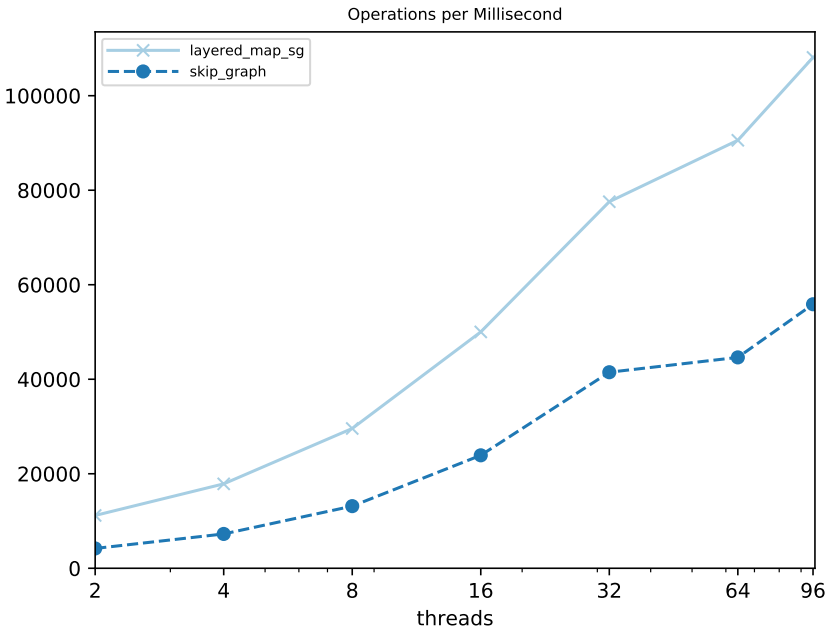

Figure 4.2: Medium-Contention, Write-Heavy Workload. 32% effective updates.

the performance or outperforms layering on a skip graph under high contention. This further reinforces the well-established idea that the overhead of maintaining the skip graph semantics through upper-level index layers outweighs the benefits of NUMA-awareness and overall throughput. Yet, such a technique does not scale well into lower contention environments. This shows that concurrent structures that are optimized for one environment are antithetical to structures optimized for other environments. Furthermore, by being aware of the benefits from this technique in the implementation proposed by this thesis, the larger ideas of laziness can used without the components of design that will cause unnecessary overhead or slow-down. It is this practice that allows us to build such a structure that is still NUMA-aware. It is also why the lazy application of the technique scales better than other lazy concurrent structures in low-contention.

The Rotating Skip List is the best performing competing data structure in the high contention testing environment. NUMASK and the No Hotspot skip list both scale particularly well. This is attributed to the effective utilization of the caching architecture and contention-free maintenance of the structure upon rotation. Other structures that utilize background threading to maintain the structure, like NUMASK and No Hotspot, also scale well under the high contention testing environment.

In low-contention environments, the layered skip graph list is the best performing non-blocking structure. By providing skip list nodal semantics, the overhead of a NUMA-aware data partitioning scheme can be significantly decreased. Furthermore, the benefit obtained by using the layered skip graph over a traditional skip list structure demonstrates the throughput performance benefit of using a NUMA-aware structure over an unaware structure.

The lock-based skip list performs well in low-contention environment since there will be a very low probability that there will be contention for a lock, and so the performing of many operations within a lock reduces the number of synchronized

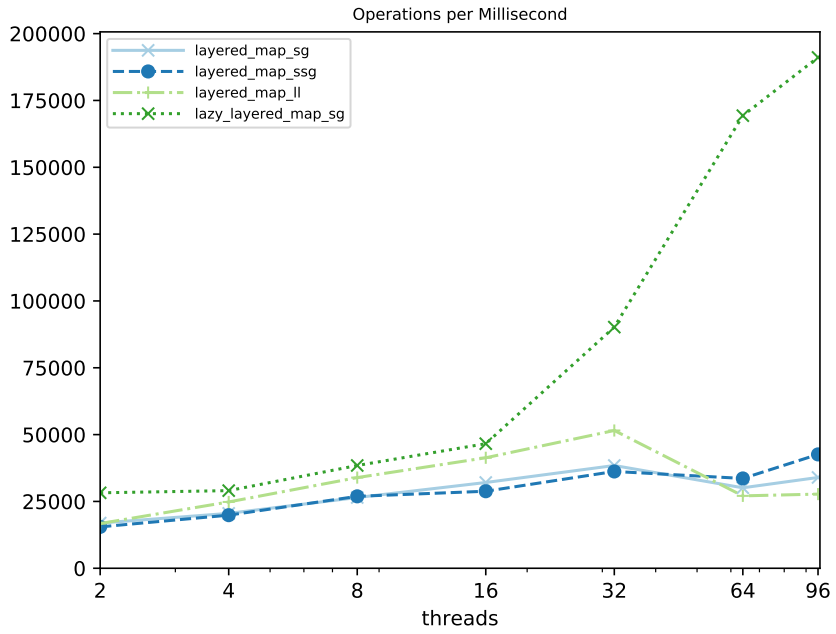

Figure 4.3: High-Contention, Write-Heavy Workload. 32% effective updates.

variables to which writing must be performed. This is seen in practice in Fig. 4.6. These results are accepted, but they are not particularly interesting from a research perspective due to the incredibly poor performance lock-based structures demonstrate under high-contention.

4.4 Locality

In measuring thread locality, an additional attribute is placed on each node called "owning thread." From here, a counting metric is included for each node operation and map each operation from the thread currently operating on that node to the thread to which it belongs. This is done because inserted nodes will be in the memory associated with that thread's particular NUMA node.

The aim of measuring where threads access memory is to show that the partitioning scheme implemented in the skip graph and skip graph variants demonstrates strong NUMA locality. The locality heavily contributes to the performance benefits. In the research leading to this thesis, it was hypothesized that the NUMA-aware partitioning scheme will introduce a highly predictable memory-access pattern. For the machinery that was used in the research leading to this thesis, such a pattern is optimal. However, seeing as most of the skip graph configuration is done at compile time, the controls that implement NUMA-awareness can be easily tuned for any hardware NUMA layout, and the general patterns will remain in tact.

It is important to note that the raw numbers do not reflect true outcomes in running the tests without the `STATS` compiler instruction flags, as additional overhead is introduced. However, they do provide an intuitive grasp as to the scale at which the operations are performed.

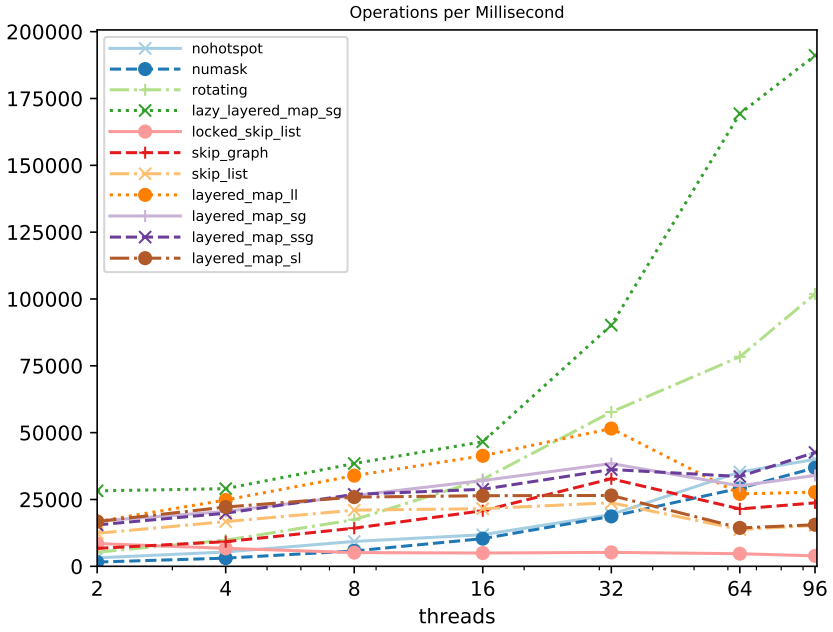

Figure 4.4: High-Contention, Write-Heavy Workload. 32% effective updates.

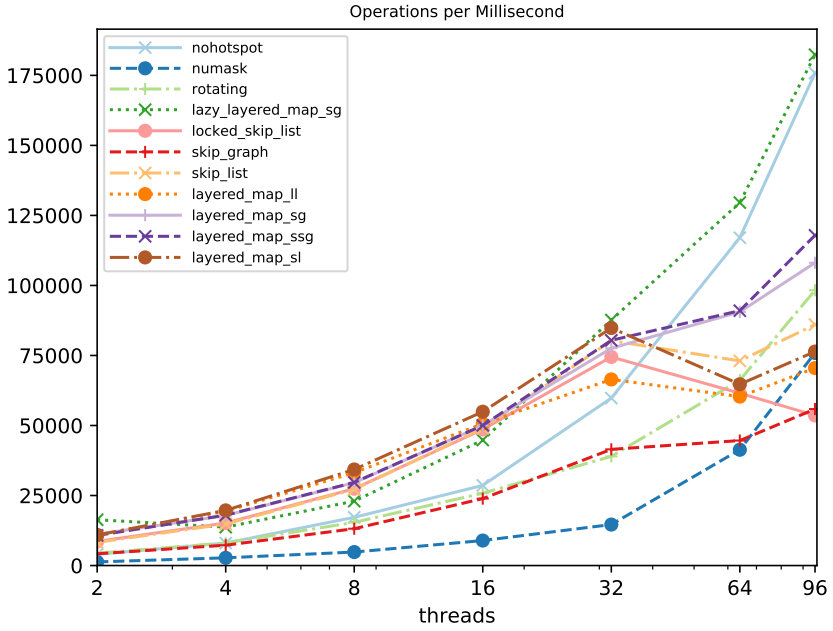

Figure 4.5: Medium-Contention, Write-Heavy Workload. 32% effective updates.

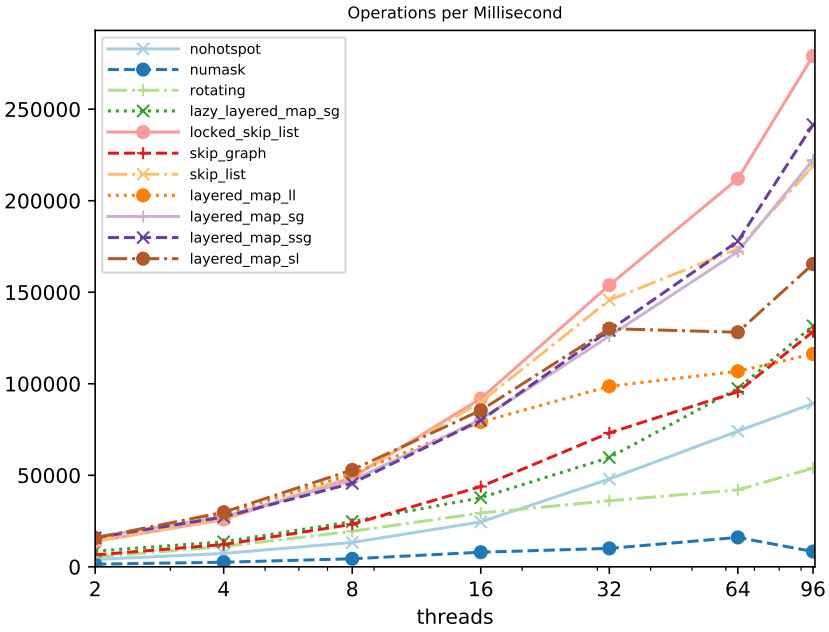

Figure 4.6: Low-Contention, Write-Heavy Workload. 32% effective updates.

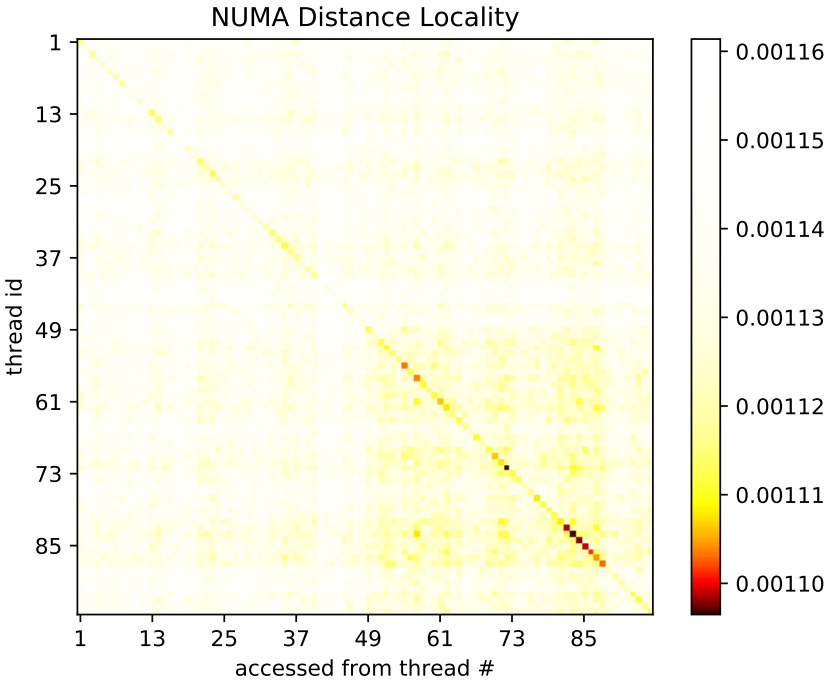

Figure 4.7: Skip List CAS memory access heatmap.

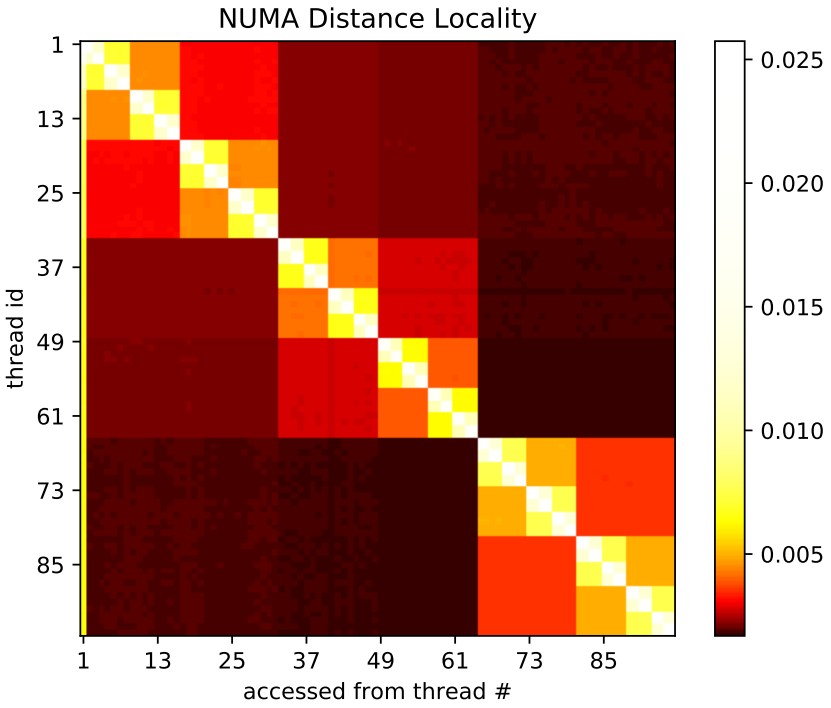

Figure 4.8: Skip Graph CAS memory access heatmap.

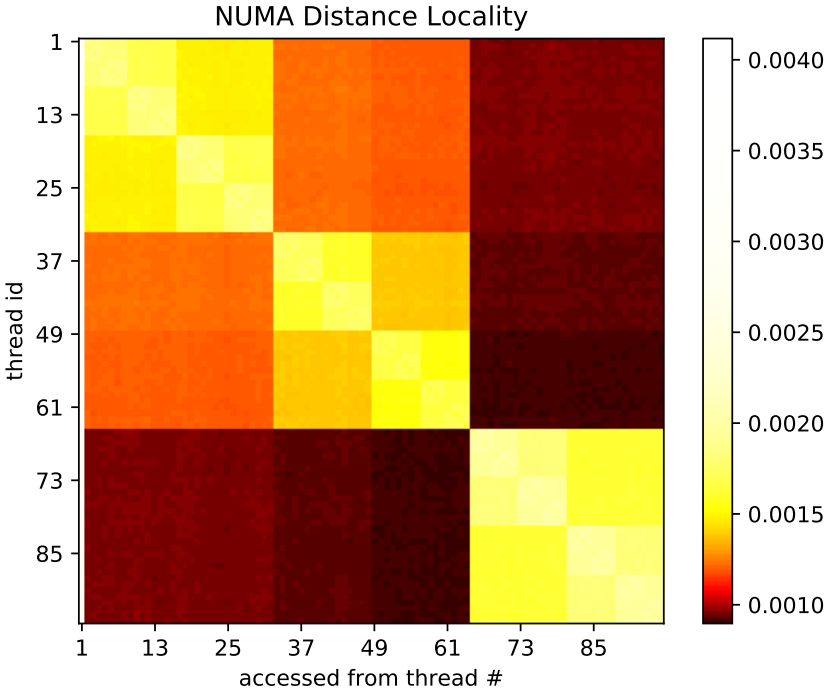

Figure 4.9: Skip Graph List CAS memory access heatmap.

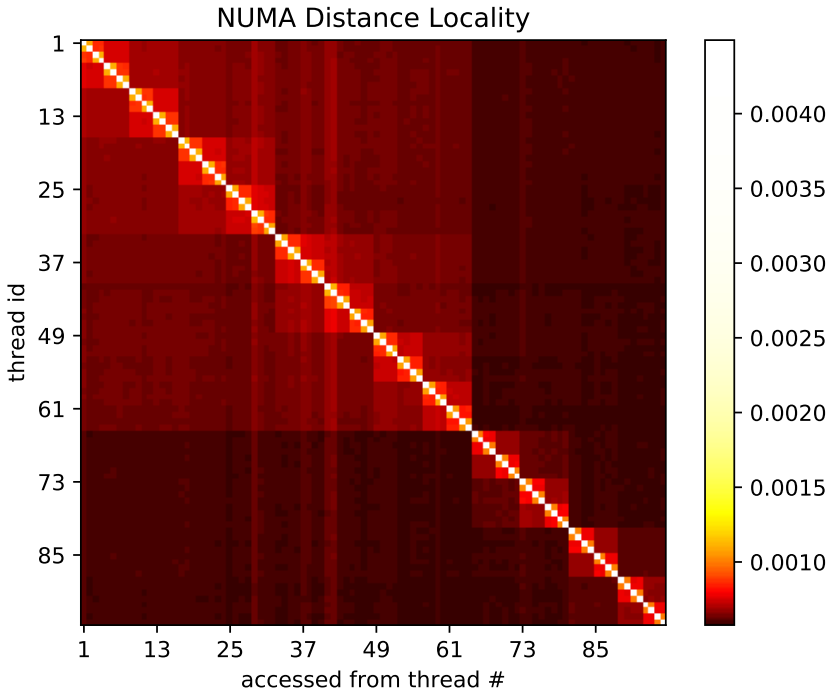

Figure 4.10: Lazy Skip Graph CAS memory access heatmap.

Skip List Locality

Seeing as most competing data structures are made of a skip list variant without the NUMA-aware partitioning scheme, the research leading to this thesis began by examining the locality features of a skip list. It can be easily seen that, in Fig. 4.7, a skip list has a totally random access pattern.

Un-stacked skip lists searches start from the head node in all situations. Furthermore, the search makes a traversal to the minimum node at the top level, regardless of where that node is in memory. As was discussed in Section 4.3, this works really well in a large element space (low contention environment) as this node will likely help make for a very fast search, as significantly more nodes are traversed at the upper level which allows for the skipping of more nodes.

However, skip lists perform particularly poorly in a small element space (high contention environment). When writing to a node has a high probability of failure, traversing nodes will become increasingly costly. The number of nodes searched will remain low in each call to the search operation, but more nodes will be traversed per overall operation. Furthermore, writing to these nodes will become increasingly more difficult, and failing at expensive operations is incredibly costly to the overall performance of the structure.

Skip Graph Locality

In examining the locality trends of the skip graph, contrary to the skip list, the access pattern is highly predictable. Threads operate in a logarithmic pattern of accessing certain regions in the skip graph structure. In each of the three skip graph variants tested, this pattern of accessing nodes takes place. That is, the majority of node accesses exist along the diagonal of the graph, demonstrating that most nodes

are accessed by the thread itself.

These accesses to nodes that each particular thread has inserted largely comes from the properties of the layered the structure, so the first node traversed in the skip graph will belong to that thread because it was inserted by that thread and accessed by its local structure. This will inflate the value along the diagonal. However, note also that there will only be one other thread contending for nodes in the upper-most level of the structure. As such, the most accessed nodes belong to the thread itself and the other thread with which share the upper-most level. At the next level down, threads share the list with two more threads and those are the next most intensive accesses. This pattern continues throughout the structure and exists entirely because of the NUMA-aware partitioning scheme.

This also reinforces the idea that, regardless of the introduction of more bottom-level contention, upper-level contention will remain constant in a skip graph no matter how many threads are operating in the overall environment. Skip list insertions will face the same contention when inserting a node into the upper-most level as it will in the bottom level. In a high element space, the consequences of this effect will seldom be seen. However, by partitioning the data strategically, contention is lower at the upper levels, which means that those CAS operations should be successful more often than in the skip list. This phenomenon, coupled with the increased NUMA locality gained from the partitioning scheme, are all structural reasons for the performance benefit of skip graph-based structures over skip list-based structures.

It is important to note that Fig. 4.8 shows that the layered skip graph has the largest scale of CAS operations. This comes from the fact that skip graphs are very dense structures. As was discussed in Section 1.5, all skip graph nodes exist at the upper-most level thereby demanding that all nodes have CAS operations at each level upon insertion. This problem is mitigated by deploying the skip graph list, as the scales in Fig. 4.9 and Fig. 4.7 are approximately the same.

4.5 Thread Pinning

Recall that running experimentation with thread pinning makes sense for NUMA. It can be argued that this will artificially inflate the results, seeing as system calls will be made in the application of such a data structure to a programmers application. However, the random assignment of threads to different regions of a NUMA architecture would merely cause thread memory to exist in several different NUMA regions[2]. As such, experiments are run with the understanding that such results may not occur in an application of the data structure rather than in one specifically for testing high-performance computing. Note that the artificial inflation of performance resulting from the lack of system calls will not impact any one structure more or less than any other structure in the experimentation.

Several different thread pinning strategies are tested. In doing so, the most effective way to access memory from a hardware perspective can be determined. There are five different pinning strategies employed from the `cat.py` file that are described in Section 2.1. They are shown in Fig. 4.11.

[2]This problem is further mitigated by pre-allocating per-thread memory that corresponds to their running position in the NUMA architecture.

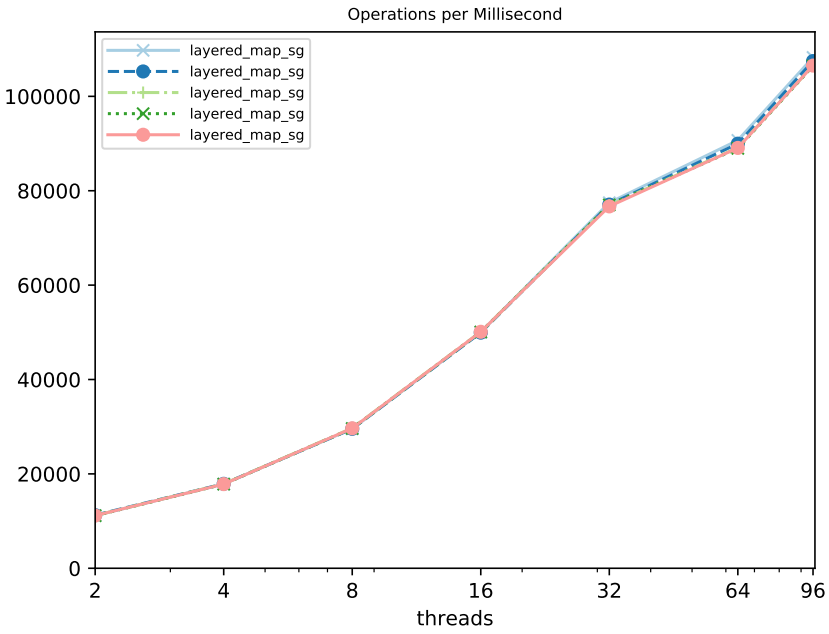

Figure 4.11: Performance of different thread pinning strategies.

Each pinning strategy is run on the stacked map skip graph list structure to demonstrate their effect. Pinning threads into the same hyperthreaded core has the best effects on the performance of the stacked map skip graph list. The performance benefits are marginal but will make an impact on the overall performance of the structure.

4.6 Nodes Per Search

The research leading to this thesis looked to measure the number of shared nodes that each structure traverses. In so doing, the effects of additional levels of a skip list or skip graph in addition to the runtime effect of layering are demonstrated. As has been shown in Section 1.3, it is anticipated that the number of nodes traversed in a perfect skip list to be $O(log(n))$. Seeing as the maximum level of a skip list or skip graph is fixed at the beginning of the runtime, this effect is closely approximated with an arbitrary element space.

Seeing as the maximum level of the skip graph must be kept low so as to avoid large overhead and segmentation faults due to memory overruns because of the structure's density, the structure is layered with local structures to approximate the effect of the upper levels of the skip list. In so doing, the number of shared nodes traversed in the skip graph and skip list follow the same overall curve without raw values. This is true as a result of more nodes needing to be traversed at the uppermost level in the skip graph because its uppermost level is lower than the skip list. The effect of layering the structure decreases the overall number of shared nodes traversed. This comes from the fact that the local structure gives each thread a unique starting position close to the node for which it is looking.

The benefit of this effect is two-fold. For one, this minimizes the number of shared node-traversals that are to be made. Following pointers to random locations

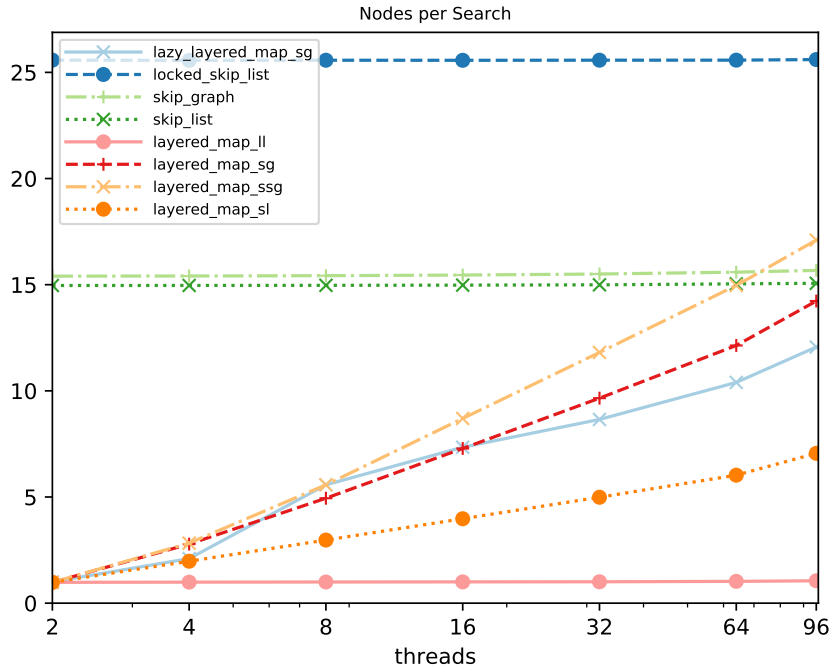

Figure 4.12: Nodes per search for several structures.

in memory can be particularly expensive in cross-NUMA socket settings. Furthermore, this demonstrates that the contention for the head element will be minimized as many threads will start their operations from a dispersed location within the structure itself.

Notice that the number of shared nodes traversed increases in layered structures. As more threads are introduced, fewer nodes will be in each thread's local structure. This means that the starting locations in the shared structure will not necessarily be as close to the final location in a search, especially if local structures are not balanced[3]. However, the limit of such pattern as the number of theads approaches infinity will only ever be the non-layered version of that structure. Never will a thread need to traverse more nodes than it would from starting from the beginning.

4.7 Memory Reclamation

In several concurrent data structures, reesearchers often make the decision between writing in Java or C++. By choosing Java, memory is freed automatically by the language processor through built-in garbage collection. In choosing C++ on the other hand, researchers can more directly control the memory in which threads operate. As has been discussed, the partitioning scheme employed in skip graph-based structures benefit from NUMA-awareness when memory is strategically partitioned. This can be most effectively done in C++, but it means that a system of memory reclamation must be implemented.

Memory management in concurrent environments can be extraordinarily difficult to maintain. In fact, it took almost an entire calendar year (from Summer 2019 up until our final tests being run in April 2020) to build a bug-free system of node

[3]This problem is addressed in Ana Hayne's '20 Honors Thesis

reclamation. Data structures built using the techniques proposed in this thesis perfectly free all memory that they allocates through their own `Allocator.hpp` library. The library works by maintaining a system of reference counting. It implements a means of lock-free reference counting, similar to the ideas first introduced in [9]. Reference counting mechanism, like that of [9], mean including a counter representing the number of pointers pointing to an object. In the event that such counter is greater than zero, the object in question cannot be freed as it is still *accessible*. As such, a piece of memory is deemed freeable when its reference count is zero.

This required the building of a concurrent skip graph-specific form of memory reclamation as opposed to applying a skip list-variant adaptable memory reclaimer. However, it also means that memory will be freed at a less frequent rate as reference counts are increased without any thread pointing directly to that reference. References are increased as soon as they are discoverable and only decreased when they are unlinked. As such, a much stricter protocol of reference counting than `Threadscan` is employed.

The protocol works by setting the number of possible "references," or means by which a node can be accessed, to the maximum number of levels by which a node can be accessed. This includes an additional reference for the local structure. When a node is unlinked from the structure at a particular level, that node's reference count is decreased as it is no longer accessible at that level. When the reference count reaches 0, the thread puts the node in a predefined queue to free several nodes in batch. This is done to ammortize the cost of freeing nodes in a particular operation and to reduce the contention for the resources of the operating system, but nodes are logically freed at the point at which they are placed in the queue.

Not all nodes will be freed from this process alone. Nodes may still be left in these queues upon terminating their thread function in the event that the reclamation threshold is not hit upon termination and nodes may still be in the structure without being physically removed. As such, two further operations are performed to perfectly free all memory that is allocated. In particular, the launching thread frees all memory remaining in the queues and makes a call to `cleanup` the data structure. This operation entails a physical removal of all nodes remaining in the data structure.

Existing memory management libraries, such as `Threadscan`[2], make assumptions about concurrent data structures that do not necessarily hold in all testing environments. In particular, `Threadscan` assumes that nodes can only be found at the node's upper-most level. Such an assumption is safe when operating in a skip list, as nodes are found through a search at the highest possible level by traversing as far forwards as possible before moving down a level, as was discussed in Sec. 1.3. However, in a skip graph, nodes are found at the highest possible skip list in which a thread is operating. Recall, however, that nodes exist in all skip lists in the bottom-most level of the skip graph. As such, it is possible that the node exists at a higher level in a different skip list than the one being operated on by the current thread.

This problem is demonstrated using Fig. 4.13 as a reference. Suppose a thread is operating in the skip list made up of "00", "0", "λ." When making a `searchRelink` call for the node `56`, the predecessors array would be `[31,31,31]`. Notice that `48` would be unlinked at levels 1 and λ. The successors array would be `[tail,88,79]`, with `62` being unlinked at all levels. After this call is made, `48` has been fully unlinked *only from the skip list "00", "0", "λ."* It has not been unlinked from

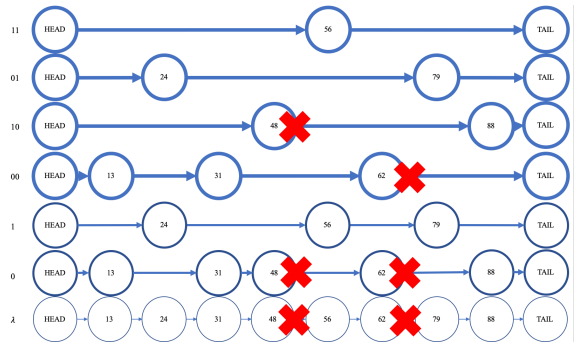

Figure 4.13: In this skip graph image, suppose that `48` has been logically removed and that `62` has been fully unlinked and freed from the structure.

the structure as a whole. If a thread operating on skip list "10", "0", "λ" were to traverse the upper-most level for `88`, then it still accesses `48`. As such, this breaks the invariant that a node is discoverable at its upper-most level, but instead at the upper-most level of the skip list in which a thread is currently operating.

The reference counting mechanism in `Threadscan` relies on the invariant that unlinking a node occurs from the node's uppermost level downwards. However, skip graph's preserve the invariant that threads do not have access to every node at its maximum height, which breaks the tacit assumption made by `Threadscan`. As such, the reference count of each node must increase at each place where it can be *potentially* found. In particular, this refers to the local structure of the node as well as the shared levels of the node, so the reference count of a node that has been inserted to 3 shared levels and the thread's local structure should have a reference count of 4.

It is also worth noting that this system of allocation perfectly reclaims all memory that it allocates over the course of its runtime. That is, no nodes are lost during the operations in either the lazy or non-lazy versions of the structure. This is demonstrated by counting exactly how many nodes are allocated which is done by counting the number of successful insertions and having a "delete" call to match each insertion. For lazy operations, however, only the insertions that truly allocated a node as opposed to insertions that merely flipped the flag from true to false should be counted. All other insertions are removed from the count.

This method has been tested in two ways. The first was by running the test that all inserted nodes had a matching free call in 30 iterations of tests in the lazy layered skip graph, the layered sparse skip graph, and the layered skip graph. The log of this test showed that no node was ever left un-freed throughout each of the 30 iterations.

4.8 Priority Queue

Recall that this thesis proposes an alternative means of developing the Spray List algorithm [1] particularly made for skip graphs in Sec. 2.3. In particular, this thesis looks to show that utilizing skip graphs for the development of priority queues ensures that threads (1) remove nodes that are closer to the absolute minimum element of the structure than in skip lists and (2) perform scalable remove operations. Experimentally, utilizing such an implementation gives scalable performance benefits

by seven times as is demonstrated in Fig. 4.14. Experimentally, this is demonstrated by showing that this solution is scalable for 96 threads.

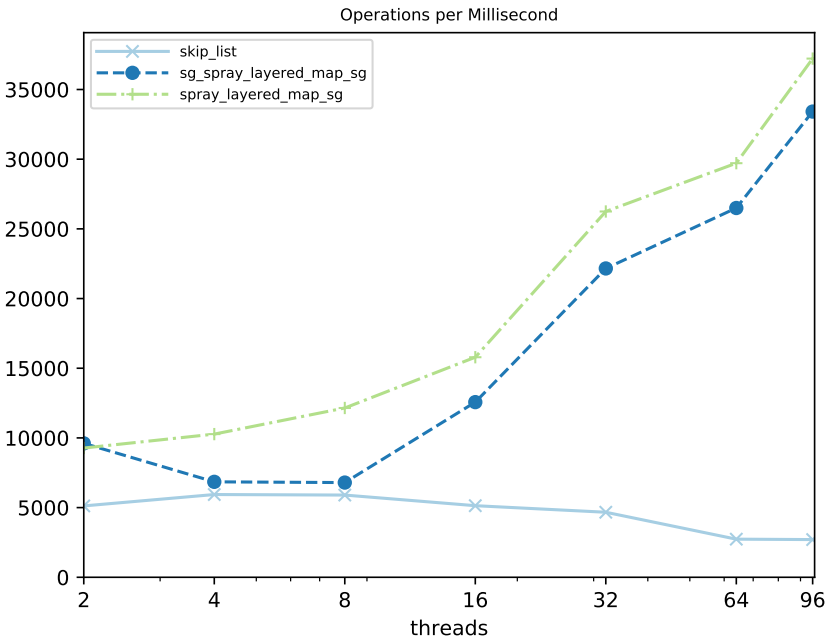

Figure 4.14: Performance in operations per millisecond. Medium Contention, 50% insertion, 50% removal.

Notice that there are two variants of the skip graph implementation of the Spray List - SG Spray or traditional Spray. SG Spray refers to the proposed algorithm that attempts to mark nodes along the way, as opposed to being sprayed to the final destination. As anticipated, this approach is slower than the process of a thrad being sprayed to a node and trying to remove the node it was assigned. Namely, contention for each node increases because threads are performing more CAS operations (which are expensive to do) and removing nodes that are much closer to the true absolute minimum element.

With that said, the SG Spray algorithms still show scalable performance at least as good or, in the normal case, better than the Spray List algorithm applied to skip lists. This while removing nodes closer to the true absolute minimum element of the structure.

Experimentally, nodes are removed much earlier in the structure, which is visible by looking at the key distribution in Fig. 4.15. This test was performed by preloading the entire element space with a node at each key and "turning off" true removals of the node. That is, instead of threads attempting to mark nodes, they will merely report the index to which they were sprayed. This method of testing, introduced by [1], gives an intuition of where threads are being sprayed to support the bounds proof provided in Sec. 2.3. As can be seen, the key distribution from the SG Spray is frequently much closer to the true absolute minimum than the traditional Spray List algorithm.

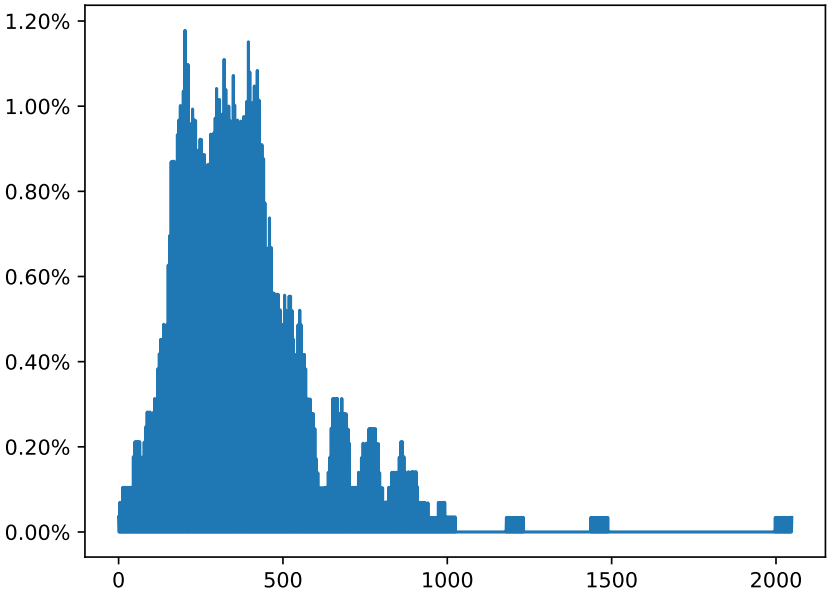

Figure 4.15: Key distribution for SG Spray PQ algorithm on a skip graph.

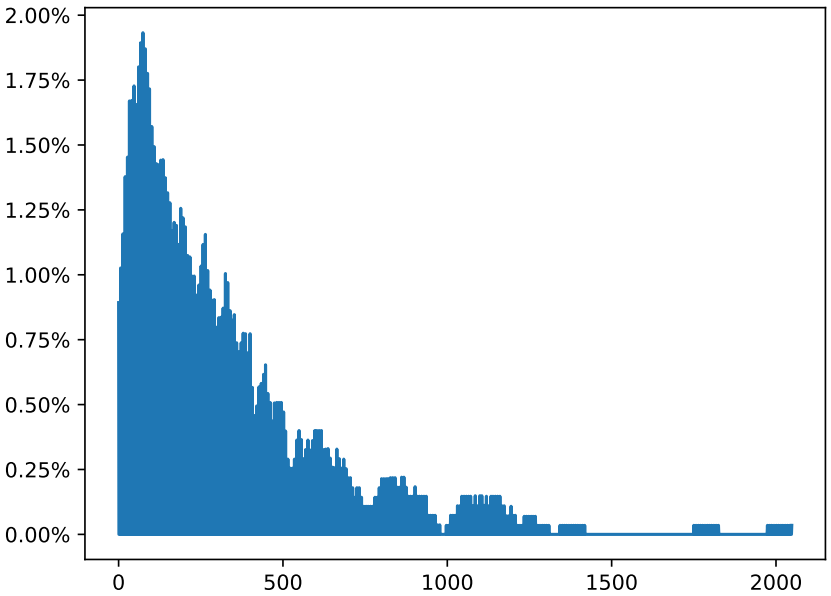

Figure 4.16: Key distribution for Spray PQ algorithm on a skip graph.

Chapter 5

Potential Future Work

Throughout this process, I have been amazed at all of the opportunities these techniques have provided for future research. There are many components that elicit further study and there is a lot of untapped potential to provide increased performance in all testing configurations.

5.1 Background Threading

Recall several of the other state-of-the-art concurrent data structures task a "background thread" with performing several update operations in the structure to minimize contention in hotspot areas. This was never explored the possibility of such a consideration in the structure, which could be particularly useful in high-contention testing environments where laziness is not necessarily present.

Work relating to background threading is being pursued by Ana Hayne (Davidson College '20) for her Honors Thesis, which builds upon the framework of the Layered Data Structures. In so doing, a system of load balancing the thread-local data structures will be performed by a background thread. This will imply a more even distribution of data across local structures and, consequently, local structure search time will be more uniform and shared structure starting positions will be consistently close. Load balancing is a commonly implemented technique in distributed systems and will be beneficial to the structure as a whole.

Background threading poses the opportunity to create a structure that performs with both minimal hotspot contention in high-contention environments and can continue to output high performance rates under low-contention environments. In particular, it would be interesting to see to what degree a key space can continue to grow before the task of a background thread becomes too large and the task will bottleneck. As such, the prospects of pursuing background threading research within the framework and heuristics of a stacked skip graph or variant seem promising.

A difficulty of pursuing such a solution is that it will artificially inflate the results of low-thread count performances. In the event that all hardware CPUs are not actively being used by threads running on the testing environment, additionally parallelism is being introduced that will inflate the overall performance of the structure. In this sense, considerations must be made to see if it makes more sense to consider the "background threads" as an active thread delegated with a singular task and to consider the work-rate as such. This changes the meaning of the metric and perhaps suggests that certain threads be constantly tasked with different roles,

none of which would create contention hotspots with the other.

5.2 Transactional Memory

On several occasions, this research considered the possibility of providing a speed-up to the structure by employing TM into the structure. As has been stated in Sec. 1.1, Transactional Memory (TM) ensures that several operations can be done within a singular synchronized block and provides an all-or-nothing result. This is both optimistic in nature and, given the results of the lock-based skip list under low contention environments, should provide some form of speed up in similar environments.

In the research leading to this thesis, a structure similar to that of the skip graph and its variants was built that is TM-based as opposed to CAS-based. However, instead of using conventional lock-based fallback code, the TM structure falls back on the CAS architecture designed in the traditional implementation of the layered skip graph and its variants. As such, TM allows for several operations to be achieved at once, like the linking together of a node into the structure or the marking of several levels of a node at once.

This research has most recently been pursued by Altan Tutar (Davidson College ‘20) as part of a research grant during the summer of 2019. His work found that TM was more beneficial in removal operations than insertions. The overall perfomance of a TM-based layered skip graph matched that of a CAS-based approach. However, TM had a very high success rate, so similar results can be achieved in two very different ways. This suggested that the well established algorithms written for skip lists are highly optimized and that changing the synchronization mechanism alone will not give increased performance.

One area of future work that is intriguing is using TM to detect contention. The benefit of using TM is that it can return a particular flag as part of its aborting upon failure. As such, this flag could be used to indicate that a particular area of the structure is hot with contention. Unfortunately, implementing such a structure with such clues is difficult in practice. Furthermore, the skip list algorithms are well enough established that it is non-trivial to see exactly how such knowledge will be positively influential on pragmatic aspects of the algorithm itself. However, once again, the potential for structural speed-up in this area of research seems promising.

Chapter 6

Conclusion

The NUMA-aware, non-blocking layered concurrent skip graph and its variants provide several opportunities for algorithmic speed-up as compared to other state-of-the-art architectures and algorithms. The techniques from several optimized skip graphs for differing levels of contention are used and they are applied to the infrastructure to partition data for NUMA so as to reduce the number of cross-socket traversals made. This will be increasingly important as modern multi-core machines are increasingly designed with this memory hierarchy. Additionally, the structure is highly adaptable so as to be easily converted to any potential future developments in the hierarchical organization of memory.

Appendix A

General Commentary

Testing Procedures

Reviewers from conference submissions before `synchrobench` was included as a standard consistently asked that the paper run tests with `synchrobench`. It makes sense, from a reviewers perpsective, to have a consistent benchmark for which all state-of-the-art concurrent maps can be tested to eliminate testing bias. However, there are certain parts of `synchrobench` that will inherently help certain data structures over another.

In particular, the fact that the `synchrobench` protocol will *alternate* by default in the construction of the local data structure is utilized. Alternate means that an update call to remove a node will try and remove the value of the most previously inserted node by that thread. As a result, there is a fast hash table implemented as part of the local data structure in the lazy layered data structure. This is done to demonstrate that the techniques are highly adaptable to the particular testing environment. However, this also demonstrates some of the faults of the `synchrobench` testing standards.

As has been suggested, the original submission was not submitted using the `synchrobench` testing standard. Instead, the testing script would pick random keys for all insertion and removal calls, which effectively emulated the effect of a non-altering testing configuration. However, the paper had originally lacked a metric to demonstrate *effective* update operations. That is, insertions or removals that return **`true`**. Doing so measures the effect of the update operations themselves as opposed to those operations that return **`false`**, which are effectively just a search operation.

All of this to say, `synchrobench` provides several important contributions; namely, it is a consistent testing framework, a well-thought out protocol, and a tool for researches to avoid necessarily implementing their own testing script. However, it is also important to recognize that merely utilizing `synchrobench` does not perfectly eliminate all testing bias. Instead, I believe an ideal testing standard would be one in which several different fair testing protocols are tested to demonstrate the effect of different protocols under different requirements. Ideally, a high-performance data structure is one that is malleably scalable in all testing environments.

Memory Reclamation

It is no secret that memory reclamation is an important part of data structure design. Furthermore, it is generally understood that calls to **`delete`** or **`free`** are expensive system calls that will slow down testing in a concurrent data structure. It is for this reason that the reclamation protocol and other protocols like `Threadscan` operate lazily on undiscoverable batches of nodes.

I wanted to use this section of this thesis to reiterate the importance of ensuring that all memory can be freed if a structure advertises that memory is perfectly reclaimable. A structure's inability to perform such memory reclamation is not in any way productive for the academic field, even at the expense of performing optimally.

In this research experience, the work has involved several state-of-the-art concurrent data structures. It has not always been plainly apparent as to what protocol is being used so as to reclaim memory in all data structures, even when it is expressly advertised in a journal publication. Such procedures do not always fully appear as advertised in the implementation of the structure, and the use of background threads render tools like `valgrind` effectively useless with its additional overhead.

The consequences of implementing, publishing, and advertising a data structure are not severe. The paper will not be retroactively removed from a conference proceeding, nor will other any other consequences land. As such, it is up to researchers to self-impose and uphold the standards for which are set for implementing a high-performance concurrent data structure. The research area as a whole will lose its integrity and developing new techniques or further developing existing techniques will render meaningless.

In my opinion, advertising full memory-reclamation capabilities should be an element of correctness in the structure. Furthermore, the extent to which memory-reclamation is used in the structure should be fully transparent. In the event that it is not, fair comparisons between structures are impossible.

For full disclosure, all metrics from all tests are without memory reclamation enabled. These decisions were made so as to ensure that the structures are tested fairly as compared to all other structures to which the tests compare.

Appendix B

Additional Algorithms

Algorithm 9 Skip Graph - searchRelink

```
1:  procedure searchRelink(key, predecessors, successors)
2:      current = head of current thread's skip list
3:      ▷ Note that this is a call to getStart when layered
4:      for each level from greatest to least do
5:          while true do
6:              next = current.next
7:              while next node is marked do
8:                  next_next = first unmarked node after next
9:                  next = next_next
10:                 if previous.next is not current then
11:                     current = previous.next
12:                     if previous is marked then
13:                         goto line 2
14:                     next = current.next
15:                     continue
16:             if CAS on previous.next from current to next is false then
17:                 current = previous.next
18:                 if previous is marked then
19:                     goto line 2
20:                 next = current.next
21:                 continue
22:             if current.key >= key then
23:                 break
24:             previous = current
25:             current = next
26:         predecessors[level] = previous
27:         successors[level] = current
28:     return successors[0].key is key and successors[0] is unmarked
```

Algorithm 10 Skip Graph - Insert

```
1:  procedure INSERT(key)
2:      while true do
3:          if searchRelink(key, predecessors, successors) then
4:              return false
5:          if toInsert == nullptr then
6:              toInsert = Node(key)
7:          toInsert.next[0] = successors[0]
8:          if CAS on predecessors[0].next from successors[0] to toInsert is false
    then
9:              continue
10:         for each level from 1 to topLevel do
11:             while true do
12:                 repeat
13:                     oldSuccessor = toInsert.next[level]
14:                     if toInsert is marked then
15:                         return true
16:                 until CAS on toInsert.next[level] from oldSuccessor to succes-
    sors[level] is true
17:                 if CAS on predecessors[level] from successors[level] to toInsert is
    false then
18:                     toInsert.next[level] = null
19:                     return true
20:                 else
21:                     break
22:     return true
```

Algorithm 11 Skip Graph - Remove

```
1:  procedure REMOVE(key)
2:      if !searchRelink(key, predecessors, successors) then
3:          return false
4:      toRemove = successors[0]
5:      for each level from topLevel to 1 do
6:          while toRemove is unmarked do
7:              CAS on toRemove.mark[level] from false to true
8:      while toRemove is unmarked at the bottom level do
9:          if CAS on toRemove.mark[0] from false to true then
10:             for each level from 0 to topLevel do
11:                 next = toRemove.next[level]
12:                 if CAS on predecessors[level] from toRemove to next is false then
13:                     break
14:             return true
15:     return false
```

Algorithm 13 searchNoRelink

```
1:  procedure Contains(key)
2:      return searchNoRelink(key)
3:  procedure searchNoRelink(key)
4:      current = head of current thread's skip list
5:      ▷ Note that this is a call to getStart when layered
6:      for each level from top to bottom do
7:          while current.key < key do
8:              previous = current
9:              current = previous.next
10:             while current is marked do
11:                 current = current.next
12:         if current.key is key and current is unmarked then
13:             return true
14:     return false
```

Algorithm 14 Non-Lazy Layered Interface - Insert

```
1: procedure Insert(key)
2:     if shared_structure.insert(key) is true then
3:         if shared node is unmarked then
4:             local_structure.emplace(key)
5:         return true
6:     return false
```

Algorithm 15 Non-Lazy Layered Interface - Remove

```
1: procedure Remove(key)
2:     return shared_structure.remove(key)
```

Algorithm 16 Non-Lazy Layered Interface - Contains

```
1: procedure Contains(key)
2:     return shared_structure.contains(key)
```

Algorithm 17 Non-Lazy Layered Interface - GetStart

```
1: procedure GetStart(key)
2:     result = successor to key in local_structure
3:     while result is not null do
4:         if result is not marked in shared structure then
5:             return result
6:         else
7:             local_structure.erase(result)
8:         result = previous node in local_structure
9:     return result
```

Algorithm 18 Non-Lazy Layered Interface - UpdateStart

```
1: procedure UpdateStart(current)
2:     ▷ Current is a node in the local structure
3:     while current is not null do
4:         if current is not marked in shared structure then
5:             return current
6:         else
7:             local_structure.erase(current)
8:         current = previous node in local_structure
9:     return current
```

Algorithm 19 Lazy Layered Interface - Insert

```
1: procedure Insert(key)
2:     if key is in local_hashtable then
3:         while true do
4:             if shared node is unmarked and shared node is unflagged then
5:                 return false
6:             if shared node flag is flipped to inserted is true then
7:                 return true
8:             else
9:                 local_structure.erase(key)
10:                local_hashtable.erase(key)
11:                break
12:    else
13:        if shared_structure.lazy_insert(key) is true then
14:            if shared node is unmarked then
15:                local_structure.emplace(key)
16:            return true
17:    return false
```

Algorithm 20 Lazy Layered Interface - Remove

```
1: procedure Remove(key)
2:     if key is in local_hashtable then
3:         while true do
4:             if shared node is unmarked then
5:                 if shared node is flagged then
6:                     return false
7:             if shared node flag is flipped to removed is true then
8:                 return true
9:             else
10:                local_structure.erase(key)
11:                local_hashtable.erase(key)
12:                break
13:    return shared_structure.lazy_remove(key)
```

Algorithm 21 Lazy Layered Interface - Contains

```
1: if key is in local_hashtable then
2:     return shared_node is unmarked and shared_node is unflagged
   return shared_structure.contains(key)
```

Algorithm 22 Lazy Layered Interface - getStart

```
1: procedure GetStart(key)
2:     result = successor to key in local_structure
3:     while result is not null do
4:         if result is not marked in shared structure then
5:             shared_structure.finish_insert(key)
6:             return result
7:         else
8:             local_structure.erase(result)
9:             local_hashtable.erase(result)
10:        result = previous node in local_structure
11:    return result
```

Algorithm 23 Lazy Layered Interface - updateStart

```
1: procedure UpdateStart(current)
2:     ▷ Current is a node in the local structure
3:     while current is not null do
4:         if current is fully linked in the shared structure and current is not
   marked in shared structure then
5:             return current
6:         else
7:             local_structure.erase(current)
8:             local_hashtable.erase(current)
9:         current = previous node in local_structure
10:    return current
```

Algorithm 24 Lazy Skip Graph - searchNoRelink2

```
1: procedure searchNoRelink(key)
2:     current = head of current thread's skip list
3:     ▷ Note that this is a call to getStart when layered
4:     for each level from top to bottom do
5:         while current.key < key do
6:             previous = current
7:             current = previous.next
8:             while current is marked or (current is flagged and enough time has
   elapsed since current was inserted and CAS on current's mark from false to
   true) do
9:                 current = current.next
10:        if current.key is key and current is unmarked then
11:            return true
12:    return false
```

Algorithm 25 Lazy Skip Graph - Lazy Insert

```
1: procedure LazyInsert(key)
2:     while true do
3:         if searchNoRelink2(key, predecessors, successors) then
4:             return false
5:         if toInsert == nullptr then
6:             toInsert = Node(key)
7:         toInsert.next[0] = successors[0]
8:         if CAS on predecessors[0].next from successors[0] to toInsert is false
   then
9:             continue
10:        return true
```

Algorithm 26 Lazy Skip Graph - Finish Insert

```
1: procedure FinishInsert(key)
2:     if searchNoRelink2(key, predecessors, successors) then
3:         for each level from 1 to topLevel do
4:             while true do
5:                 repeat
6:                     oldSuccessor = toInsert.next[level]
7:                     if toInsert is marked then
8:                         return flase
9:                 until CAS on toInsert.next[level] from oldSuccessor to succes-
   sors[level] is true
10:                if CAS on predecessors[level] from successors[level] to toInsert is
   false then
11:                    toInsert.next[level] = null
12:                    return false
13:                else
14:                    break
15:        return true
16:    return false
```

Algorithm 27 Lazy Skip Graph - Remove

```
1:  procedure REMOVE(key)
2:      current = head of current thread's skip list
3:      ▷ Note that this is a call to getStart when layered
4:      while true do
5:          if searchNoRelink(current) is false then
6:              return false
7:          while true do
8:              if current is unmarked then
9:                  if additional flag is set to removed then
10:                     return false
11:                 else
12:                     if CAS on additional flag from inserted to removed is true
    then
13:                         return true
14:             else
15:                 break
```

Bibliography

[1] Dan Alistarh et al. "The spraylist: A scalable relaxed priority queue". In: *ACM SIGPLAN Notices*. Vol. 50. 8. ACM. 2015, pp. 11–20.

[2] Dan Alistarh et al. "Threadscan: Automatic and scalable memory reclamation". In: *ACM Transactions on Parallel Computing (TOPC)* 4.4 (2018), p. 18.

[3] Martin Ankerl. *Robin Hood Hashing*. `https://github.com/martinus/robin-hood-hashing`. 2019.

[4] James Aspnes and Gauri Shah. "Skip graphs". In: *Acm transactions on algorithms (talg)* 3.4 (2007), p. 37.

[5] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. "The adaptive priority queue with elimination and combining". In: *International Symposium on Distributed Computing*. Springer. 2014, pp. 406–420.

[6] Irina Calciu et al. "NUMA-aware reader-writer locks". In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 157–166.

[7] Tyler Crain, Vincent Gramoli, and Michel Raynal. "No hot spot non-blocking skip list". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE. 2013, pp. 196–205.

[8] Henry Daly et al. "NUMASK: high performance scalable skip list for NUMA". In: *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[9] David L. Detlefs et al. "Lock-Free Reference Counting". In: *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*. PODC '01. Newport, Rhode Island, USA: Association for Computing Machinery, 2001, pp. 190–199. ISBN: 1581133839. DOI: `10.1145/383962.384016`. URL: `https://doi.org/10.1145/383962.384016`.

[10] David Dice, Virendra J Marathe, and Nir Shavit. "Lock cohorting: a general technique for designing NUMA locks". In: *ACM SIGPLAN Notices* 47.8 (2012), pp. 247–256.

[11] Ian Dick, Alan Fekete, and Vincent Gramoli. "A skip list for multicore". In: *Concurrency and Computation: Practice and Experience* 29.4 (2017), e3876.

[12] Vincent Gramoli. "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms". In: *ACM SIGPLAN Notices*. Vol. 50. 8. ACM. 2015, pp. 1–10.

[13] Danny Hendler et al. "Flat combining and the synchronization-parallelism tradeoff". In: *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. 2010, pp. 355–364.

[14] Maurice Herlihy and Nir Shavit. "The art of multiprocessor programming". In: *PODC*. Vol. 6. 2006, pp. 1–2.

[15] Maurice Herlihy et al. "A provably correct scalable concurrent skip list". In: *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer. 2006.

[16] Jerry Zheng Li. "The SprayList: a scalable relaxed priority queue". PhD thesis. Massachusetts Institute of Technology, 2015.

[17] William Pugh. "Skip lists: a probabilistic alternative to balanced trees". In: *Communications of the ACM* 33.6 (1990).

[18] Nir N Shavit, Yosef Lev, and Maurice P Herlihy. *Concurrent lock-free skiplist with wait-free contains operator*. US Patent 7,937,378. May 2011.

[19] Samuel Thomas et al. "Using Skip Graphs for Increased NUMA Locality". In: *CoRR* abs/1902.06891 (2020). arXiv: 1902.06891. URL: http://arxiv.org/abs/1902.06891.