Journal of Parallel and Distributed Computing 167 (2022) 31–49

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Using skip graphs for increased NUMA locality

Samuel Thomas, Roxana Hayne, Jonad Pulaj, Hammurabi Mendes [*]

Davidson College, Mathematics and Computer Science, 209 Ridge Rd., Davidson, NC, USA

A R T I C L E I N F O

Article history:
Received 24 February 2021
Received in revised form 17 February 2022
Accepted 9 April 2022
Available online 21 April 2022

Keywords:
NUMA
Concurrent data structures
Skip graphs
Locality

A B S T R A C T

We present a NUMA-aware concurrent data structure design based on a data-partitioned, concurrent skip graph indexed by thread-local sequential maps. Our design brings significant quantitative and qualitative improvements on NUMA locality, as well as reduced contention for synchronized memory accesses. Maps show up to 6x higher compare-and-swap (CAS) locality, up to a 68.6% reduction on the number of remote CAS operations, and an increase from 88.3% to 99% on the CAS success rate compared to a control implementation. Remote memory accesses are not only reduced in number, but the larger the NUMA distance between threads, the larger the reduction is. Relaxed priority queues implemented using our technique show similar scalability improvements, with provable reduction in contention and decrease in relaxation in one of our proposed implementations.

1. Introduction

The increasing availability of computing cores on shared memory machines makes concurrent data structure design a critical factor for the design of high-performance applications or parallel systems. Non-blocking [28], linearizable [29] data structures are particularly appealing, since they can effectively replace sequential or blocking (lock-based) structures without compromising the semantics expected by users (systems designers). However, the design landscape for concurrent structures is changing: NUMA architectures emerge as a set of computing/memory "nodes" linked by an interconnect, making memory accesses within the same NUMA node cheaper than those made across different ones.

Under the usual assumption that threads are pinned to cores, we adopt the definition of *local memory* accesses as those operating in memory initially allocated by the current thread (under first-touch NUMA policy), and *remote* accesses as those accesses that are non-local. Note that our definitions are conservative, as data initially allocated by two different threads could indeed be located within the same NUMA node. Our goal is to increase NUMA *locality* – the ratio of local over total memory accesses, and research is very active in this area. Some approaches [7,11,38] focus on redesigning data structures with NUMA awareness, which is effective as we have full ability to exploit the structure's internal features for the task. Unfortunately, complete redesigns can pose significant development and research efforts, unsuitable for non-specialists. On the other hand, approaches such as [9] allow sequential structures to be "plugged-in" and benefit from NUMA-aware concurrency, based on replicating the dataset among nodes, batching local operations, and coordinating batches as to minimize inter-node traffic.

A critical goal in concurrent data structure design is the reduction of *contention* for synchronized memory accesses, characterized when two or more threads operate concurrently on nearby locations in memory (e.g., same cache line). Synchronized operations introduce memory fences in the cache-coherence protocol, and optionally with enriched semantics, such as `get_and_increment()` (atomic increment) or `compare_and_swap()` (CAS) (conditional atomic exchange), so they are critical for lock-free data structure design. However, they also introduce high invalidation traffic in the cache-coherence system, particularly under contention. With NUMA, it is even more critical that contention is reduced, as such traffic happens *across* different memory domains, resulting in expensive access costs. Reducing contention can be attained by promoting *internal data parallelism* for synchronized memory accesses, and our technique simultaneously promotes a reduction on remote memory accesses on NUMA.

1.1. Our contributions

We present a technique to promote NUMA-aware data parallelism inside the concurrent data structure, bringing significant quantitative and qualitative improvements on NUMA locality, as well as reduced contention for synchronized memory accesses. Our design is based on integrating thread-local sequential maps with skip graphs ([4,23], described also in Sec. 3), while per-

[*] Corresponding author.
E-mail addresses: sathomas@davidson.edu (S. Thomas), anhayne@davidson.edu (R. Hayne), jopulaj@davidson.edu (J. Pulaj), hamendes@davidson.edu (H. Mendes).

https://doi.org/10.1016/j.jpdc.2022.04.006
0743-7315/© 2022 Elsevier Inc. All rights reserved.

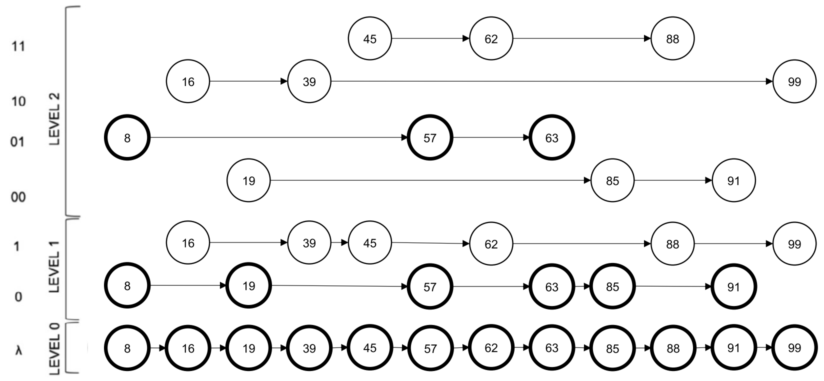

Fig. 1. A skip graph can be seen as a set of skip lists sharing their levels (one of them is highlighted). It contains 2^i linked lists at each level i. With T threads, we have $T/2^i$ of them working in each level-i linked list, separated by NUMA proximity, which increases data access parallelism, reduces contention, and increases NUMA locality.

forming a data partitioning scheme over the skip graphs for increased NUMA locality. By "qualitative" increase in NUMA locality, we mean that remote memory accesses are not only reduced in number, but the larger the distance between threads in the system, the larger the reduction is (Sec. 11). At a high level, skip graphs can be viewed as multiple skip lists [40] that overlap, so we partition sub-components of skip graphs among threads as to promote higher NUMA locality and reduced contention (increased data parallelism). We design internal algorithms to take full advantage of the partitioned dataset in order to promote this goal. Note that, as originally defined, skip graphs are expensive data structures, so our technique is made viable in practice by incorporating existing thread-local indexing and well-documented laziness principles [10,11] into our design. As a proof-of-concept, we implemented maps and relaxed priority queues ([2,26,48,49]). Maps have been implemented with and without using laziness techniques, but always using thread-local indexing similarly to [11]. We are competitive with state-of-the-art maps [10,11,14]: in some cases, we see 80% increased performance, while in others, we see similar performance to the faster running implementation. As part of our NUMA locality assessment, we observe a 6x higher CAS locality, a 68.6% reduction on the number of remote CAS operations, and an increase from 88.3% to 99% of CAS success rate when using a lazy skip graph map implementation, as compared to our baseline data structure – a skip list subject to the same codebase, optimizations, and implementation practices. Memory access patterns are visualized in Sec. 11, showing evident qualitative improvement.

We also contribute by implementing *relaxed priority queues* [2,26,48,49], which return an element among the k smallest elements in a set, rather than the absolute smallest. We not only use our data partitioning technique, which brings increased NUMA locality and reduced contention, but we consider a couple of algorithmic variations that further harness key structural features of the skip graph. Additionally, we provide a *formal argument* indicating that one of our two priority queue protocols is subject to smaller contention and it is also slightly less relaxed (that is, the removed elements are closer to the minimum element as defined in a strict priority queue).

2. Background: skip graphs

Fig. 1 shows a skip graph, which performs the role of our shared structure. A skip graph is composed of many singly-linked lists across multiple levels 0...*MaxLevel*. Each level i has exactly 2^i linked lists, and partitions the nodes in level $i-1$ (for $i>0$) in two sublists. In the level-0 list (called λ), all nodes are present. The two level-1 lists, 0 and 1, partition the level-0 list, and so on. In the original definition of skip graphs, aimed at peer-to-peer distributed applications, the specific partitioning is probabilistic. In this paper, we have a *partitioning scheme* that will allocate nodes as to maximize data locality in the operations in our overall data structure (we describe our partitioning scheme below).

Skip lists [40] are similar to skip graphs, but the former contain one linked list per level, and the later 2^i lists at level i. In a skip list, all elements belong to level 0, 1/2 of the elements at level 1, 1/4 of the elements at level 2, and so on. Hence, a skip graph is a collection of overlapping skip lists: in Fig. 1, if we select exactly one linked list per level, we obtain a skip list. In Fig. 1, one skip list is highlighted, which we denote $(\lambda, 0, 01)$ after the linked lists that must be chosen to define it, from the bottom level to the highest level. This way, 39 is in the skip list denoted by $(\lambda, 1, 10)$, and 85 is in the skip list denoted by $(\lambda, 0, 00)$. Skip graph searches are skip list searches: start from a node's top level, and follow high-level pointers as far as possible before moving down levels. For example, from 16 we reach 63 by following the path $16 \rightarrow 39$, $\downarrow$, $39 \rightarrow 45$, $45 \rightarrow 62$, $\downarrow$, $62 \rightarrow 63$. Note we only follow pointers in the skip list that 16 (our starting point) belongs to in its top level: $(\lambda, 1, 10)$. We refer to any skip list within the skip graph as *shared skip list*, and any of the individual linked lists within the skip graph as a *shared linked list*. All elements belong to one shared skip list in all of its levels, so we can *always* perform a skip list search starting from that node's top level, traversing only the skip list that the node belongs to in its top level. Hence, despite a richer structure, it is important to notice that skip graph searches are skip list searches.

3. Architecture

In this section, we start with a high-level view of our NUMA-aware optimizations (our data partitioning scheme), following with implementation details and considerations about linearizability and correctness.

3.1. General architecture

Our overall architecture consists of an underlying skip graph [4], called a *shared structure*, and multiple thread-local, sequential, navigable maps called *local structures*, one per thread. The local structures allow insertions, removals, and contains operations to "jump" to positions in the shared structure near to where they will complete. The "jump" is done on thread-local memory, which contributes to reducing remote memory accesses since we avoid traversing long paths in the shared structure (distributed across multiple NUMA memory banks). Once in the shared structure, our data partitioning scheme over the skip graph promotes a further reduction in remote memory accesses due to our partitioning mechanism. We call our overall structure *layered structure* given this architecture.

We say *nodes* store *elements*, although we use these terms interchangeably. We further use the terms *local nodes* or *shared nodes* to refer to those nodes belonging to a local structure or to the shared structure, respectively. We now describe how the local and shared structures interact. When a thread inserts an element e, it first creates a shared node `s` in the skip graph, and then the thread's local structure will map $e \rightarrow$ `s`. When a thread removes an element e, it first (i) *logically deletes* the shared node `s` in the skip graph; which causes the next two events, in arbitrary order: (ii-a) a "lazy" physical removal by traversing threads in the shared structure and (ii-b) a "lazy" physical removal of the local structure entry $e \rightarrow$ `s` by the thread that originally inserted the element.

3.2. Data partitioning

Even though the skip graph is shared by all threads, we limit where each thread operates on it, configuring a *partition scheme*. First, we establish that the maximum level of the skip graph is $\text{MaxLevel} = \lceil \log(T) \rceil - 1$, where T denotes the number of threads in the system (named $\{T_0 \dots T_{T-1}\}$). While such a "low" MaxLevel does not guarantee logarithmic searches by itself, our local structures perform the role of the missing higher levels as they "jump" to positions in the shared structure nearby where we expect operations to complete. We then divide the skip graph lists in the following way: (i) each thread T_i has a sequence of MaxLevel bits called a *membership vector*, denoted by M_i; (ii) T_i can only operate on the skip list characterized by the suffixes of M_i. That skip list is called the *associated skip list* of T_i, denoted L_i. For example, consider Fig. 1. Because $\text{MaxLevel} = 2$, T_i's membership vector M_i is a 2-bit string, say 01. Hence, T_i will always insert, remove, and search by following paths of the skip list $L_i = (\lambda, 0, 01)$. Note that since $\text{MaxLevel} = \lceil \log(T) \rceil - 1$, each top-level linked list is shared by 2 threads, and any arbitrary level-i list is operated by at most $T/2^i$ threads.

A data partitioning is induced as we distribute threads over the skip graph as above, as we assume threads operate in different NUMA domains, and allocate memory within their domain. Furthermore, we have the opportunity to increase NUMA locality by having threads pinned to "closer" hardware to share more lists in the skip graph. We do that by generating membership vectors according to *physical NUMA features* of the machine. For example, consider a system with $T = 16$ threads ($\text{MaxLevel} = 3$) and 2 NUMA nodes, each with 2 CPUs, each of those with 2 cores, each of those with 2 hyperthreads. Now, for any two threads T_i and T_j: (i) if T_i and T_j run on different NUMA nodes, they get membership vectors with no common 3-bit suffix; (ii) if T_i and T_j run on the same NUMA node, but on different cores, they get membership vectors with a common 1-bit suffix only; (iii) if T_i and T_j run on the same NUMA node and core, but on different hyperthreads, they get membership vectors with a common 2-bit suffix only; and (iv) if T_i and T_j share the same hyperthread, they get membership vectors with a common 3-bit suffix. Now, on the level-3 linked lists, any contention relates to core-local data (and hopefully located on the core's closest cache); on the level-2 linked lists, any contention relates to CPU-local data; on the level-1 lists, any contention relates to NUMA-local data. Not only we expect less contention in upper-level lists, because they are shared among less threads, but we expect this *contention to relate to more local data*. Further, any search that traverses the skip graph, as we discussed, is a skip list search. Hence, we *first* traverse core-local data, and *if we go down a level, the target data cannot be located in the same core*. Similarly, we then traverse CPU-local data, then NUMA-local data, and if we ever go down a level, the target data must be found remotely, where local/remote depend on the level in question. If we ever leave, say, a NUMA node, we never come back to it. The same applies to CPU-local data, or core-local data, and this happens **as a direct consequence of our data partitioning mechanism, and our choice of data structure**.

We have an automated mechanism that generates membership vectors based on inspecting the system's CPU/socket/domain structure, and we indeed verify less contention (substantial improvement on CAS success ratio) and more locality (visualized graphically) in Sec. 11. As indicated in the introduction, the above partitioning scheme not only provides a *quantitative* improvement on CAS and synchronized read locality, but also a *qualitative* improvement on those metrics: the larger the distance between two NUMA nodes, the bigger the reduction in remote accesses between threads pinned to those nodes. In fact, we also expect more locality and less contention even when threads are not strictly pinned to cores, as discussed in Sec. 11.

3.3. Alternative shared structure

In order to further explore benefits and tradeoffs of skip graphs, we also created and tested a second shared structure, called a *sparse skip graph*. This structure is a skip graph where elements are made present in level i *of any shared skip list* with probability $1/2^i$, just like in a regular skip list. The combination of skip graph partitioning and skip list refinement makes elements be present in level i *of a particular linked list* with probability $1/4^i$. As seen in Fig. 2, each of the level-1 lists "0" and "1" partition only 50% of the elements of "λ", which would be selected at 50% chance independently. So, "0" and "1" each have about 25% of the elements in "λ". Similarly, the level-2 lists "00" and "01" partition only 50% of the elements of "0", each selected at 50% chance, independently. So, lists "00" and "01" each have about 6.25% of the elements in "λ". With sparse skip graphs, only elements that reach the top level are added to the local structures. This is crucial because the local structures, besides pointing to shared nodes nearby the target destinations, should also point to maximum-level nodes from which we can start an efficient search. Hence, using sparse skip graphs gives two immediate advantages: (i) the local structures are smaller; and (ii) the insertion and removal in the shared structure requires changes in less than MaxLevel levels. The tradeoff is that the starting point given by the local structures is not as close to the requested element compared to regular skip graphs.

3.4. Notation and data representation

The skip graph has an array that points to the first node of every list in the structure, called `head`. For each node `s` in the skip graph, we denote by `s.next[i]` its successor on the level-i list that it participates (we also call it the *level-i reference* of `s`). Each reference has a `marked` and a `valid` bit used for lazy removal, and the functions `s.getMark(i)`, `getValid(i)`, `s.casMark(i, exp, new)`, and `s.casValid(i, exp, new)` operate on `s.next[i]`. The functions `s.getMarkValid(i)` and `s.casMarkValid(i, (mExp, vExp), (mNew, vNew))` operate on `marked` and `valid` bits of `s.next[i]` simultaneously. As the address space on the x64 architecture is currently restricted to 48B, we have 16 bits available for flags in each word, so we can use one bit to implement marking while still using single-word CAS operations.

The local structures have their local nodes store an element identifier (a "key") and a *reference* to a shared node (a "value"). The local structures use these to map elements to shared nodes. A local node *l*'s key (resp. value) is obtained by calling `l.getKey()` (resp. `l.getValue()`). Note that the "value" (the shared node) always itself contains the same key as the local node *l*. We assume that the local structures are navigable, and we can traverse them by calling `l.getPrev()` and `l.getNext()` on a local

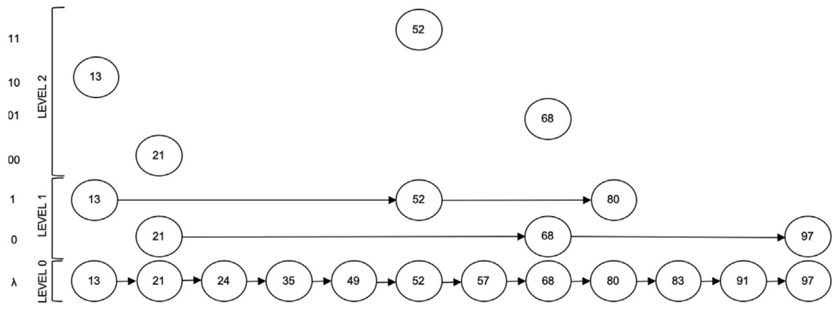

Fig. 2. A sparse skip graph: elements are in level i *of any skip list* with probability $1/2^i$.

node. The local structure for thread i, a C++ `std::map`, is denoted by `localstructures[i]`. Each has an auxiliary hash table `hashtable[i]`, allowing threads to consult a fast hashtable [3] before consulting a slower map. Therefore, our local structures, in practice, are implemented with two complementary, sequential data structures.

3.5. Laziness

State-of-the-art concurrent data structures rely heavily on postponing internal work until it is absolutely needed, in the hope it becomes unnecessary. We implemented a *lazy* variant of our layered structure, employing this principle as: (i) The insertion of shared nodes in the skip graph is done in the level 0 first, and the inserting thread only completes the insertion at upper levels when the node in question is requested to start a search operation. (ii) Removals are performed logically by "invalidating" a shared node, and nodes are marked for physical removal only when the threads that originally inserted those nodes find them invalidated after a minimal *commission period* for which they exist in the structure. As the physical removal of a node is expensive, the commission period is intended to have this operation done only when necessary. Experimentally (Sec. 11), we found that a commission period proportional to the number of threads, say $350000 \cdot T$ cycles, for instance, performs *very well* under high-contention without introducing too much overhead in low-contention (in the latter, a longer commission period could leave the data structure much larger at times). (iii) We implemented an optimization that removes *chains* of marked shared nodes with a single CAS operation, and, related to laziness, we do that only when *substituting* a chain of marked shared nodes with an inserting node. Although this protocol has the potential to leave too many marked nodes in the data structure, we verified experimentally that the number of traversed shared nodes per operation is less than in a skip list up to 96 threads.

3.6. Physical removal

We now expand our discussion of valid and invalid nodes to a more formal definition. Each node has a *valid* bit and a *marked* bit in each successor reference in the skip graph. When a node's level-0 reference `marked` (resp. `valid`) bit is set, we say the shared node is *marked* (resp. *invalid*). The concepts of *unmarked* and *valid* are obvious. In the non-lazy version, we do not use the `valid` bit, and in that case an unmarked shared node indicates presence in the abstract key set, while a marked shared node indicates a *logically deleted* node.

In the lazy version, an unmarked, valid node indicates presence in the abstract set; an unmarked, invalid node indicates absence from the abstract set (logically deleted), but also that the process of physically unlinking the node has not started; a marked node can only be invalid, and in that case the process of physically unlinking is ready to start. Each shared node `s` has a field `s.allocTimestamp`, set at shared node construction, used to calculate when a node's commission period has elapsed, thus making it a candidate for physical removal (see the discussion on laziness, above). The removal process happens in distinct phases: (i) A thread that traverses the data structure will mark a shared node with an expired commission period only if such node has been inserted by the thread in question (recall that "marking" nodes means marking its successor reference at level 0). (ii) When threads remove marked nodes from their local structures, they also mark upper-level successor references in order to promote physical removal of the shared node in the upper levels as well. As in many cases logically deleted nodes have not been inserted in upper levels, this additional reference marking is not always necessary or complete. (iii) Finally, traversing threads perform the physical cleanup using the relink optimization discussed below. Note that (i) and (ii) are synchronization operations done in *local memory*, and we consider this protocol for maximal locality as one of our implementation-related contributions.

As in textbook skip lists [27], we indicate willingness to physically remove a node `s` by marking its `s.next[i]` references for all levels $i \dots 0$ the node belongs to. Within a level i, searches performed on behalf of insertions and removals physically remove nodes with marked `next[i]` references by employing a *single* CAS per node. In contrast, both our skip lists or skip graphs remove *sequences* of marked references with a single CAS per sequence, a trivial optimization that we denote by *relink optimization*. The correctness of this protocol is trivial when we consider that marked references are immutable.

4. Related work

Applications, particularly DBMSs, are not only concerned with *data placement* (somewhat related to our goals), but also with *task scheduling* (a non-goal here). Some approaches consider NUMA locality in the query evaluation and planning [19,33,34], and others do at the OS level [47]. We have a simple approach to data placement: threads index their inserted elements locally and the dataset is effectively partitioned; our focus is on the *data access pattern*. Systems such as [44] are particularly concerned with data access pattern issues, as their operations are mostly uniform among threads. We do not discuss task scheduling, as we *do not* rely on approaches such as delegation [6,32] or flat-combining [15,25], which *do* bring concerns related to this issue because of the nonuniform work division among threads.

Efficient data shuffling policies have been examined in [18,36], and in [5,12], the latter not only considering remote memory access *latency*, but also *contention*. This kind of work is more relevant for us, as improving the data access pattern in order to minimize contention is a *primary goal* in NUMA settings; minimizing latency is an *additional goal* for our work in particular. Techniques such as replication, interleaving, etc., mentioned in these papers, particularly in [12,36], are *not* used here, but look appealing for future work.

4.1. Skip lists and skip graphs

Skip lists first appeared in [41], although [17,30,35] were most widely discussed in the literature [27]. Skip lists have also been used to implement priority queues, either exact [8,42,43] or with a relaxed definition [2,26,48,49]. SkipNets [23] are similar (if not identical) to skip graphs, proposed relatively at the same time. We consider those equivalent, and equally applicable. Skip graph variations, such as in [21], typically address issues related to distributed systems, such as node size; we are aware of a single concurrent implementation in shared memory in [37], although it is lock-based, in contrast with both of our lock-free variants. Our implementation relies heavily on laziness, as we postpone much of the internal work until they are absolutely needed. The “No Hotspot” skip list [10] uses similar lazy principles, albeit with a different protocol. The “Rotating” skip list [14] has a novel construction (“wheels”) meant to improve cache efficiency and locality, and also constitutes a modern, state-of-the-art implementation.

4.2. NUMA awareness and layered design

The work presented in [13] gives a systematic approach to provide NUMA-awareness to locks. Tailor-made data structures for NUMA systems, such as [7,20,38] have also been developed, using (now) standard techniques such as elimination [24] and delegation [6]. We think that “blackbox” approaches, such as in [9], are interesting as they relieve systems programmers from “customizing” their data structures for NUMA, a notoriously complicated task for non-specialists (and specialists alike [27]). NUMASK [11] is an interesting skip list that uses its higher levels as a hierarchical “index” to the bottom-level list, which stores the dataset. In our case, the dataset is located in a structure of its own, a multi-level skip graph. This allows for our data partitioning mechanism, designed to (i) reduce non-local NUMA traffic, and particularly avoid traversals that navigate back and forth across NUMA nodes; and (ii) reduce contention by creating areas within the shared structure where only subsets of threads operate. Our thread-local indexing, similarly, is more detached from the dataset, and could be implemented with any sequential, navigable map. In our implementation, for example, our map is actually a combination of a search tree and a hash table. Finally, our indexes are not replicated, but partitioned. Even with our optional load balancing mechanism in place, threads donate nodes but do not replicate their indexing. Apart from differences of granularity and function, the idea of separating thread-local views and shared views appeared in [1], although their approach is more akin to combining, as they eventually merge thread-local views into the shared structure from time to time.

A brief announcement (i.e. not a full paper) related to this work was originally published in [45], and substantial new material has been incorporated in [46]: lazy skip graphs (Sec. 3.5), load balancing (Sec. 9), the commission period policy (Sec. 3.5), relaxed priority queue algorithms (Sec. 10) and their analysis (Sec. 10.1). In addition to [46], this journal version includes (i) a complete, formal analysis of our priority queue algorithms; (ii) a detailed discussion of all the algorithms in our protocol (insertion, removal, search, PQ removal); (iii) further details on the load balancing protocol; (iv) more experimental results, now including locality experiments for layered skip graphs, lazy-layered skip graphs, and sparse skip graphs. We also provide detailed measurements on cache misses, and performance/locality results for read-heavy workloads.

Algorithm 1 bool Layered::insert(K key, V value).

```
1: SharedNode result = hashtables[threadId].find(key)
2: if result ≠ null then
3:     bool returnValue = false
4:     if SG::insertHelper(result, returnValue) then
5:         return returnValue
6: SharedNode insertedSkipNode = null
7: if SG::lazyInsert(key, insertedSkipNode) then
8:     localstructures[threadId].insert(key, insertedSkipNode)
9:     hashtables[threadId].insert(insertedSkipNode)
10:    return true
11: return false
```

Algorithm 2 bool SG::insertHelper(SharedNode toInsert, **ref** bool returnValue).

```
1: while true do
2:     if not toInsert.getMark(0) then
3:         if toInsert.getMarkValid(0) == (false, valid) then        ▷ Duplicate
4:             bool returnValue = false
5:             return true
6:         if toInsert.casMarkValid((false, invalid), (false, valid)) then   ▷ Flipped valid
7:             returnValue = true
8:             return true
9:     else
10:        localstructures[threadId].erase(toInsert.getKey())
11:        hashtables[threadId].erase(toInsert.getKey())
12:        break
13: return false
```

5. Insertion

The main operations supported by our layered skip graphs are insertion, removal, and containment. In this section, we describe the insertion algorithms, pointing out relevant details in the code, and providing linearization arguments as the various subroutines are described. The linearization details are given below, marking the linearization point descriptions with I-i, I-ii, etc.

Algorithms 1, 2, and 3 show the insert operation for `key`. In `insert()` (Algorithm 1), if the thread finds a reference to a shared node with the goal key (call it `result`), it invokes `insertHelper()` (Algorithm 2), which returns `true` to indicate to `insert()` that the insertion was completed. The `returnValue` parameter, which is populated inside `insertHelper()`, is set to `false` to indicate that we found a duplicate node, and set to `true` to indicate that a logical insertion occurred by flipping the `valid` bit of the pre-existing node found.

Specifically, `insertHelper()` atomically checks `result`’s `marked` and `valid` bits and finishes the operation if: (I-i) `result` was seen as an unmarked valid node, so we linearize a failed insertion right before our check (Algorithm 2, line 3). (I-ii) `result` was atomically changed from invalid to valid, so we linearize the successful insertion right at that time (Algorithm 2, line 6). If `insertHelper()` cannot finish the operation (by returning false), `result` got marked, so (i) we clean the thread’s local structure (Algorithm 2, lines 10 and 11); and (ii) we call `lazyInsert()` (Algorithm 3, line 7), which will complete the insertion.

The procedure `lazyInsert()` (Algorithm 3) adds the node at the bottom level of the skip graph. At line 5, it starts by calling `getStart()` (Algorithm 4, described just below), which finds the closest preceding, unmarked shared node pointed by the local structure, but one that also has been fully inserted, lazily, at all levels beforehand. We call this node `currentStart`. We then perform a search in line 7, starting from `currentStart`, by calling `lazyRelinkSearch()` (Algorithm 5, Sec. 6). This procedure searches for the bottom-level predecessor for the new element, so we can eventually perform physical insertion at line 14 with a CAS operation.

Algorithm 3 bool SG::lazyInsert(T key, **ref** SharedNode insertedSkipNode).

```
1: Array predecessors[MaxLevel], middle[MaxLevel], successors[MaxLevel]
2: ▷ Shared node being inserted in the bottom level only
3: SharedNode toInsert
4: ▷ Search the local structure and get the closest starting point
5: SharedNode currentStart = getStart(key)
6: while true do
7:     if SG::lazyRelinkSearch(key, predecessors, middle, successors, currentStart) then
8:         returnValue = false
9:         if SG::insertHelper(successors[0], returnValue) then
10:            return returnValue
11:        else
12:            continue
13:    toInsert.setNext(0, successors[0])
14:    if not predecessors[0].casNext(middle[0], toInsert) then
15:        updateStart(currentStart)
16:        continue
17:    insertedSkipNode = toInsert
18:    return true
```

In case the invocation of `lazyRelinkSearch()` finds an unmarked shared node with the goal key, this node will be pointed by `successors[0]`. In that case (I-iii) a call to `insertHelper()` (line 9) will be our linearization point, just like it happened in our previous discussion of Algorithm 1. If that call returns false, `successors[0]` became marked, so we retry the search at line 12. (I-iv).

In case the invocation of `lazyRelinkSearch()` does not find an unmarked shared node with the goal key, we try to insert the node on line 14. In that case (I-iv), if the attempt fails, we retry the search until either (I-iv-a) we link the new shared node in line 14, so we linearize at the point the CAS succeeded, noting that shared nodes are allocated as unmarked and valid; or (I-iv-b) find another unmarked shared node with the goal key, reverting to case (I-i). The lazy insertion only happens at level 0. In line 15, `updateStart` (Alg. 13, Appendix A) makes sure that `currentStart` is unmarked, otherwise it traverses the local structure backwards and updates `currentStart` to the closest unmarked shared node seen.

As briefly mentioned before, the procedure `getStart()` (Algorithm 4) finds an unmarked shared node pointed by the local structure that closest precede `key`, but one node that has been fully inserted at all levels of the skip graph lazily. It performs an initial search at line 1, then traverses the tree backwards (line 18) for as long as the shared nodes pointed to by the local structure are found marked. Along this traversal, if we find a suitable candidate, but one whose insertion in the upper levels of its shared skip list has not been completed, we call `finishInsert()` (Alg. 14, Appendix A) to *try* to complete the insertion of such potential predecessor right away. This call is done at line 6 (still talking about Algorithm 4), and it can fail if the node gets marked before all levels are linked. In that case we continue our backward traversal on `getStart()`. Otherwise, we found our predecessor node, and `getStart()` can return it.

One problem with completing the insertion of our potential returned predecessor, in line 6 of Algorithm 4, is that we need *another* predecessor in order to complete an insertion. The call to `updateStart()` (a subroutine described in Alg. 13, Appendix A) finds such "predecessor's predecessor," but does so without trying to finish insertions itself, or the whole `getStart()` procedure would be ill-defined. In other words, `updateStart()` is a simplified version of `getStart()` that does not finish insertions lazily, and ignores nodes that are not fully inserted. With the "predecessor's predecessor," we can complete the insertion of the predecessor node returned by `getStart()`.

Note that `insertHelper()` can continuously loop only when (i) `getMark(0)` continuously changes to `true` inside the block of

Algorithm 4 LocalStructureIterator LocalStructure::getStart(K key).

```
1: iterator = localstructures[threadId].getMaxLowerEqual(key)
2: while iterator $\neq$ null do
3:     SharedNode sharedNode = iterator.sharedNode
4:     if not sharedNode.getMark(0) or not sharedNode.getMark(MaxLevel) then
5:         if not sharedNode.inserted then
6:             if SG::finishInsert(sharedNode, updateStart(iterator)) then
7:                 return iterator          ▷ Node has just been fully inserted
8:             else
9:                 ▷ Erase below does not invalidate the iterator
10:                localstructures[threadId].erase(key)
11:                hashtables[threadId].erase(key)
12:        else
13:            return iterator              ▷ Node already found fully inserted
14:    else
15:        ▷ Erase below does not invalidate the iterator
16:        localstructures[threadId].erase(key)
17:        hashtables[threadId].erase(key)
18:    iterator = iterator.getPrev()
19: return iterator
```

line 2, indicating continuously successful insertions and removals; or (ii) we continuously fail to execute line 6, indicating continuously successful insertions. In addition, Algorithm 3 can continuously loop only when other insertions and removals are succeeding, thus changing the state of the `predecessors`, `middle`, and `successor` arrays inside `lazyRelinkSearch()` (called in line 7 in `lazyInsert()`). The main insertion procedure relies on Algorithms 2 and 3, and otherwise contains no loops. In addition, the procedure `getStart()`, also used in Algorithm 3, cannot run continuously, as it navigates through a non-concurrent data structure that only decreases in size when marked nodes are identified and removed. Hence, the whole insertion procedure is lock-free.

6. Search

All of the supported operations by the layered skip graph, not only containment, rely on two internal search procedures: `lazyRelinkSearch()` in Algorithm 5, and `retireSearch()` in Algorithm 6. While insertions use `lazyRelinkSearch()`, remove and contains operations will use `retireSearch()`, a slightly quicker variant that does not keep track of `predecessors`, `successors`, and `middle`. Both algorithms are presented below. The algorithms in this subsection are slight variants from their skip list counterparts, and are provided for completeness; linearizability and progress (lock-freedom) arguments are very similar to those discussed in [17,30,31].

Our first search procedure, `lazyRelinkSearch()`, is presented below in Algorithm 5. It identifies, at all levels: (A) which nodes should precede `toInsert` (referenced in `predecessors`); (B) which nodes should succeed `toInsert` (referenced in `successors`); and (C) which nodes, referenced in the `middle` array, had `predecessors[i].getNext(i)` right at the moment each predecessor `predecessor[i]` was identified. Between `predecessors[i]` and `successors[i]` we have a sequence of nodes with their level-*i* references marked. We hope these nodes get replaced, *through a single CAS operation*, with `toInsert` (hence implementing our relink optimization protocol, Sec. 3.6). Importantly, during this process, line 4 uses `checkRetire()` (Alg. 15, Appendix A.1) to check if nodes are invalid and their commission period has expired, in which case these nodes get marked.

Our second search procedure, `retireSearch()`, is presented below in Algorithm 6. This algorithm is employed by contains and remove operations in order to search for unmarked shared nodes with a goal `key`, starting from `currentStart`. The algorithm is a simplification of `lazyRelinkSearch()`, presented above, as it does not keep track of predecessors or successors as the structure is traversed.

Algorithm 5 bool SG::lazyRelinkSearch(K key, SharedNode[] predecessors, SharedNode[] middle, SharedNode[] successors, LocalStructureIterator currentStart).

```
1: current = null
2: for level = MaxLevel → 0 do
3:     current = originalCurrent = previous.getNext(level)
4:     while current.getMark(0) or current.checkRetire() do
5:         current = current.getNext(level)
6:     while current.getKey() < key do
7:         previous = current
8:         current = originalCurrent = previous.getNext(level)
9:         while current.getMark(0) or current.checkRetire() do
10:            current = current.getNext(level)
11:    predecessors[level] = previous
12:    middle[level] = originalCurrent
13:    successors[level] = current
14: return (successors[0].getKey() == key and not successors[0].getMark(0))
```

Algorithm 6 bool SG::retireSearch(K key, SharedNode found, LocalStructureIterator currentStart).

```
1: previous = currentStart.sharedNode
2: current = null
3: for level = MaxLevel → 0 do
4:     current = previous.getNext(level)
5:     while current.getMark(0) or current.checkRetire() do
6:         current = current.getNext(level)
7:     while current.getKey() < key do
8:         previous = current
9:         current = originalCurrent = previous.getNext(level)
10:        while current.getMark(0) or current.checkRetire() do
11:            current = current.getNext(level)
12:    if current.getKey() == key and not current.getMark(0) then
13:        found = current
14:        return true
15: return (successors[0].getKey() == key and not successors[0].getMark(0))
```

Algorithm 7 bool Layered::contains(K key).

```
1: SharedNode result = hashtables[threadId].find(key)
2: if result ≠ null then
3:     if not result.getMark(0) then
4:         return (result.getMarkValid(0) == (false, valid))
5:     else
6:         localstructures[threadId].erase(key)
7:         hashtables[threadId].erase(key)
8: return SG::contains(key)
```

The search algorithms given above are very similar to their skip list counterparts discussed in [17,30,31]. Their linearization points follow exactly the arguments in the references just provided, noting that skip graph searches are identical to skip list searches.

7. Containment

The containment operation supported by the layered skip graph is described in this section. The linearization points are noted in locations marked with indicators C-i, C-ii, etc. Our main procedure, `contains()` (Algorithm 7), starts trying to locate an element through the local hashtable. If an unmarked, valid node is found (line 4), we linearize a successful contains at this point (C-i). Otherwise, we call `SG::contains()` (Algorithm 8), which will complete the procedure.

The algorithm `SG::contains()` (Algorithm 8) starts by calling `getStart()` (Algorithm 4), which finds the closest preceding, unmarked shared node pointed by the local structure (line 2). We call this node `currentStart`. We then perform a search in line 4, starting from `currentStart`, using our procedure `retireSearch()` (Algorithm 6). (C-ii) If an unmarked shared node with the goal key is not found, we linearize a failed contains operation at the time we find the element succeeding that key in the bottom level. (C-iii) If an unmarked shared node with the goal key is found: (C-

Algorithm 8 bool SG::contains(K key).

```
1: ▷ Search the local structure and get the closest starting point
2: currentStart = getStart(key) alg:sgContains:getStart
3: ▷ If an unmarked shared node is not found, the element cannot exist
4: if not retireSearch(key, found, currentStart) then
5:     return false
6: return (result.getMarkValid(0) == (false, valid))
```

Algorithm 9 bool Layered::remove(T key, U value).

```
1: SharedNode result = hashtables[threadId].find(key)
2: if result ≠ null then
3:     bool returnValue = false
4:     if SG::removeHelper(result, returnValue) then
5:         return returnValue
6: return SG::lazyRemove(key)
```

Algorithm 10 bool SG::removeHelper(SharedNode toRemove, **ref** bool returnValue).

```
1: while true do
2:     if not toRemove.getMark(0) then
3:         if toRemove.getMarkValid(0) == (false, invalid) then        ▷ Non-existent
4:             bool returnValue = false
5:             return true
6:         if toRemove.casMarkValid((false, valid), (false, invalid)) then    ▷ Flipped valid
7:             returnValue = true
8:             return true
9:     else
10:        localstructures[threadId].erase(toRemove.getKey())
11:        hashtables[threadId].erase(toRemove.getKey())
12:        break
13: return false
```

iii-a) if we verify the node is unmarked and valid (test of line 6), we linearize a successful contains at that time. (C-iii-b) Otherwise, we linearize the failed contains at the earliest moment among the time we failed to verify the node was unmarked and valid (line 6), or right after the time the node became unmarked.

These algorithms are also lock-free: the only loop involved in the contains procedure is located in the lock-free `retireSearch()`, discussed in the previous section.

8. Removals

Removals are the final major operation supported by our layered skip graphs. Removals are similar to insertions, and are performed with three analogous algorithms. Our discussion follows below, with linearization points marked with indicators R-i, R-ii, etc.

Algorithms `remove()` (Algorithm 9), `SG::removeHelper()` (Algorithm 10), and `SG::lazyRemove()` (Algorithm 11) implement the remove operation for `key`. In `remove()` (Algorithm 9), if the thread finds a reference to a shared node with the goal key, it calls `removeHelper()` (Algorithm 10). This call returns true only if: (R-i) We found an unmarked, invalid node, so we linearize a failed removal right at that time. There cannot exist another unmarked, valid node in this case, which justifies our linearizability. (R-ii) We successfully unset the `valid` bit of an unmarked, valid node, so we linearize the successful removal right at that time as well. If `removeHelper()` returned false, the shared node previously found is marked, so (i) we clean the thread's local structure (Algorithm 10, lines 10 and 11); and (ii) we call `lazyRemove()` (Algorithm 11), which will complete the removal.

The algorithm `lazyRemove()` (Algorithm 11) starts by calling `getStart()` (Algorithm 4), which finds the closest preceding, unmarked shared node pointed by the local structure (line 2). We call this node `currentStart`. We then perform a search in line 5, starting from `currentStart` and: (R-iii) If an unmarked shared node with the goal key is found (call it `found`), then a call to

Algorithm 11 bool SG::lazyRemove(SharedNode current).

```
1: ▷ Search the local structure and get the closest starting point
2: currentStart = getStart(key)
3: while true do
4:     ▷ If an unmarked shared node is not found, the element cannot exist
5:     if not retireSearch(key, toRemove, currentStart) then
6:         return false
7:     returnValue = false
8:     if SG::removeHelper(successors[0], returnValue) then
9:         return returnValue
```

`removeHelper()` (line 8) will define our linearization, just like our previous discussion of Algorithm 9. If that call returns false, `found` became marked, so we retry the search. (R-iv) No unmarked node with the goal key has been found, so in that case we linearize a failed removal at the time we find the element succeeding that key in the bottom level.

Note that `removeHelper()` can continuously loop only when (i) `getMark(0)` continuously changes to `true` inside the block of line 2, indicating continuously successful insertions and removals; or (ii) we continuously fail to execute line 6, indicating continuously successful insertions. In addition, `lazyRemove()` can continuously loop only when other insertions and removals are succeeding, thus changing the state of the `predecessors`, `middle`, and `successor` arrays inside `retireSearch()` (called in line 5 in `lazyRemove()`). The main removal procedure relies on Algorithms 10 and 11, and otherwise contains no loops. In addition, the procedure `getStart()`, also used in Algorithm 6, cannot run continuously, as it navigates through a non-concurrent data structure that only decreases in size when marked nodes are identified and removed. Hence, the whole removal procedure is lock-free.

9. Load balancing

We have an optional mechanism for handling unbalanced workloads, which addresses the following scenarios: (i) Some threads may only insert, while others only perform removals/contains. (ii) Distinct groups of threads may insert in distinct partitions of the element space. Both scenarios are problematic because threads do not necessarily find good starting points for their search operations if their local structures are empty or skewed towards a partition of the element space. Our load-balancing mechanism is based on having threads donate a fraction of their nodes inserted in the shared structure so they are added to local structures of other threads. A background thread takes into account the number of inserted elements announced by every worker thread, and, based on those numbers, continuously indicates to each worker thread the fraction of inserted nodes that are requested for donation. The worker threads place such fraction of inserted nodes into donation queues (one per worker thread), which are collected by the background thread and distributed uniformly among all other worker threads. Threads inspect receiving queues for incoming nodes, and add them into their local structures. Donated nodes are deleted in the local structures just like non-donated nodes, when they are seen as marked while a thread traverses its local structure.

9.1. Implementation details

In order for the background thread to determine how many pointers each worker thread should donate to other threads, it must consider the number of times each worker thread has successfully inserted into the shared structure. The more successful insertions a worker thread completes, the more pointers it has in its local structure, and the more pointers it should donate to other threads. Each worker thread maintains atomic counters indicating the number of successful insertions (updated every 100 operations to amortize the cost of atomic updates). The *background thread* will

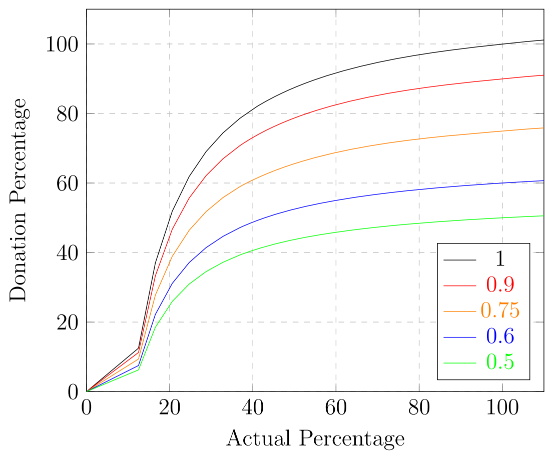

Fig. 3. Donation % (y) vs. insertion % (x), assuming 8 threads.

update global fractions indicating the percentage of inserted nodes that should be donated by each worker thread. The actual donation happens through two concurrent, wait-free, single-producer, single-consumer queues. We optimized the design of those queues to take advantage of the fact that we only have a single producer and a single consumer operating over them. For each thread, one queue receives donations from the background thread (*incoming queue*), and the other queue sends donated pointers from the worker thread to the background thread (*outgoing queue*). These latter pointers will then be redistributed to other worker threads by the background thread.

The donation percentage is calculated as follows: if a worker thread T_i announces the insertion of I_i elements out of a total of $I_T = \sum_{\{0\le i<T\}} I_i$ elements inserted, define $q_i = I_i/I_T$. If $q_i \le 1/T$, then T_i donates the fraction q_i of its elements; otherwise, it donates $p \cdot [((T \cdot q_i) + q_i - 1)/(T \cdot q_i)]$ of them, where p is an "aggressiveness factor" varying between 0.5 and 1. The value $1/T$ is the "expected fraction", or the fraction of total insertions each thread would have inserted if all threads were inserting under similar uniformly-distributed loads. Note that if the only goal of this protocol was to ensure that all threads have equally sized local structures, it would not make sense for a thread with an actual fraction less than or equal to the expected fraction to donate at all. However, our protocol also seeks to ensure an even distribution of keys within each thread's local structure. Then, it is possible that all threads could be inserting equally but only within distinct ranges of the key space. Therefore, all threads inserting less than or equal to the expected fraction still donate a percentage of their pointers. This ensures that there will always be some amount of pointer redistribution across threads to help expand the key range of each local structure.

If a thread's actual fraction is greater than the expected fraction, that thread is inserting more than its fair share of nodes into the shared structure. Fig. 3 illustrates the donation percent associated with a thread considering its actual inserting percentage (q_i for worker thread i in our formula), under different aggressiveness factors p, varying between 0.5 and 1, in a system with 8 threads.

Donated nodes have been inserted in their bottom-level by a thread T_i, while they might be inserted in upper levels by a thread T_j, $j \ne i$. We are currently working on delegating the creation of upper levels to the original thread T_i, avoiding to cross NUMA nodes when building up those levels. On lazy implementations, with levels built only when needed, the impact of this problem is reduced.

10. Priority queues

Our layered structure can implement priority queue ADTs (and relaxed versions of it) in addition to sets/maps. Similarly to [8, 42,43], we rely on marking elements in the bottom level of the (now) skip graph in order to logically delete elements. We consider two *relaxed priority queue* approaches [26,48,49]: (i) the use of the spraying technique of [2] over skip graphs; and (ii) a custom protocol that traverses the skip graph deterministically, marking elements along this traversal. We consider (ii) as one of the contributions of this paper.

Regarding option (i), the main idea of [2] is to disperse threads over the skip list bottom level through a random walk called a *spray*. If we apply the technique to a skip graph, each thread T_i would navigate only through its associated skip list L_i. We see several advantages of performing spray operations over skip graphs rather than skip lists. Our partitioning scheme will still incur in better memory locality and reduced contention, as discussed before. On the other hand, we show that spraying over skip graphs has a slightly bigger removal range (same asymptotically, but higher nevertheless, proved in Sec. 10.1). Related to option (ii), we implement a deterministic traversal in the skip graph, marking elements along the way. Informally, a thread T_i starts at the highest level of its associated skip list, traverses marked nodes, and attempts to mark at the current position. If the attempt succeeds, a node has been logically deleted, otherwise the thread moves down a level, traverses marked nodes, and proceeds similarly. At level 0, two mark attempts are tried, and upon failing the second one, the process is restarted (Algorithm 12). We prove in Sec. 10.1 that the number of CAS operations required to logically delete a sequence of T nodes in our deterministic protocol is $2T$ (so each node is subject to contention by 2 threads in expectation). In either approach, our layered structure gives the opportunity to perform physical cleanup of a *whole prefix* of the local structure containing logically deleted nodes, so many nodes could be removed at once at cost comparable to a single removal.

In Sec. 10.1, we give formal arguments that quantify (A) the removal range of spraying operations in skip lists and skip graphs; and (B) the contention anticipated for approaches (i) and (ii) discussed above, over skip graphs.

10.1. Analysis of spray operations on perfect skip graphs

In this section, we provide formal arguments related to the performance of spray operations as in [2] as applied over skip graphs. Part of the analysis presented here closely follows the aforementioned paper, while others are particular to skip graphs. Recall that T is the number of threads in the system. We will assume that $\log T$ is integral, and that for the number of nodes N in our skip graphs or lists, $N \geq T$ holds. When convenient we will explicitly assume $T = 2^n$, where $n \geq 1$. The particular variant of skip graphs we refer to, throughout our analysis, is the one where in each level i, where $0 \leq i \leq \log T - 1$, there are 2^i lists as described in Sec. 3.

10.1.1. Reach of skip graph spraying

In this section, we discuss the removal range of spray operations over skip graphs. Lets start with a few general definitions used throughout our proofs:

Definition 1. A skip graph is **perfect** if the absolute value of the difference in the positions of any two consecutive nodes on level i is 2^i, for all $0 \leq i \leq \log T - 1$.

The operation $\mathtt{SPRAY}(\mathtt{H}, \mathtt{L}, \mathtt{D})_{\mathtt{j}}$ defines a random walk starting at height H, i.e., at the head of a list j at level H of the skip graph. The walk continues forward to the right a uniformly random number of consecutive nodes chosen from $[0, \mathtt{L}]$ then moves down the skip graph D levels. This is repeated until we reach the bottom list and (possibly) walk forward to the right one last time. Thus the walk ends at position x in the bottom list. We are interested in characterizing the probability that any SPRAY operation starting from any j of the $2^{\log T - 1}$ lists of the maximum level of the skip graph lands on position x in the bottom list.

Definition 2. Let j be any of the $2^{\log T - 1}$ lists of the maximum level of a perfect skip graph, and let x be the position of any node in the perfect skip graph. Then $\mathtt{F}(\mathtt{x})_{\mathtt{j}}$ denotes the probability that for fixed H, L, and D, $\mathtt{SPRAY}(\mathtt{H}, \mathtt{L}, \mathtt{D})_{\mathtt{j}}$ lands on position x in the bottom list.

In the proof of the theorem below, we closely follow [2] with appropriate changes and modifications for the perfect skip graph. We assume w.l.o.g. that $\log T$ is odd.

Theorem 1. *Fix* $\mathtt{H} = \log \mathtt{T} - 1$, $\mathtt{L} = \log \mathtt{T}$, *and* $\mathtt{D} = 1$. *Then for any position x,* $\mathtt{F}(\mathtt{x})_{\mathtt{j}} \leq \frac{1}{\mathtt{T}}$.

Proof. Let j be any of the $2^{\log T - 1}$ lists of the maximum level of a given skip graph, and let x be the position of any node in the perfect skip graph. Let pos_j denote the position of the head of list j. If $x < pos_j$, then trivially $\mathtt{F}(\mathtt{x})_{\mathtt{j}} = 0$. Hence, we assume w.l.o.g. $x \geq pos_j$. Furthermore let $\bar{x} := x - pos_j$. We simplify the exposition and show that $\mathtt{F}(\mathtt{x})_{\mathtt{j}} \leq \frac{1}{\mathtt{T}}$ by making our arguments on $\bar{x}$. Note that in order for $\mathtt{SPRAY}(\log \mathtt{T} - 1, \log \mathtt{T}, 1)_{\mathtt{j}}$ to land on position x, the following must hold:

$$\sum_{i=0}^{\log T - 1} a_i 2^i = \bar{x}, \tag{1}$$

where for each i, a_i is chosen independently from $[0, \log T]$. Thus, we are interested in the probability that the sum of the different a_i, each chosen independently from $[0, \log T]$ yield Equation (1). Instead of calculating this probability directly we will give an upper bound for it based on a simple argument with respect to the parity of the bits in the binary expansion of $\bar{x}$.

The idea is to give the probability that each randomly chosen a_i, starting with $i = 0$, contributes to the correct parity of the binary expansion of $\bar{x}$ moving from left to right. This is possible because of the shifting factor of 2^i for each a_i. Certainly, all randomly chosen a_i that satisfy Equation (1) must satisfy this property, but not the other way around. Therefore, this probability yields an upper bound for $\mathtt{F}(\mathtt{x})_{\mathtt{j}}$. We show this more precisely as follows.

Let $\bar{x}(l)$ denote the l-th least significant bit of $\bar{x}$, and for each i, let $a_i(l)$ denote the l-th least significant bit of a_i. In order to ensure the correct parity of the binary expansion of $\bar{x}$ moving from left to right by *sequentially* choosing a_i in $[0, \log T]$ starting with $i = 0$, it is easy to see that $a_0(0) = \bar{x}(0)$ must hold. More generally, because of the shifting factor of 2^k for each a_k, the following equation (whose left hand side represents bit addition)

$$a_k(0) + a_{k-1}(1) + \ldots + a_0(k) + c \equiv \bar{x}(k) \bmod 2, \tag{2}$$

must hold for all $1 \leq k \leq \log T - 1$, where c is determined by the previous choice of $a_0, a_1, \ldots a_{k-1}$.

We will derive the desired upper bound by considering the probability of sequentially and randomly picking a_i that respect the parity of the binary expansion of $\bar{x}$ as explained above, starting with a_0, then continuing with a_1 and so on until $a_{\log T - 1}$. We say that a chosen a_k is a match, if it respects the parity of the binary expansion of $\bar{x}$ as in Equation (2). Thus we are interested in the probability on the right hand side of the following inequality:

$$\mathtt{F(x)_j} \le \mathtt{Pr[a_0\ \text{is a match}]} \prod_{\mathtt{i=1}}^{\log \mathtt{T}-1} \mathtt{Pr[a_i\ \text{is a match}|A_{i-1}]}, \quad (3)$$

where A_{i-i} is the event where all a_k such that $0 \le k \le i-1$ are a match. First we will explicitly derive the right hand side of Inequality (3) then we will give a simple argument as to why the inequality holds.

First of all, it is clear that $\mathtt{Pr[a_0\ \text{is a match}]} = \frac{1}{2}$. This is because we assumed $\log T$ is odd and exactly half the numbers in $[0, \log T]$ are even or odd. Furthermore, this can be readily generalized for $\mathtt{Pr[a_i\ \text{is a match}|A_{i-1}]}$, where $1 \le i \le \log T - 1$. Given A_{i-1}, i.e., $a_0, a_1, \dots a_{i-1}$ that have respected the parity of $\bar{x}$ as above, exactly half the possible choices in $[0, \log T]$ will contribute an a_i such $a_k(0)$ satisfies Equation (2), where $k = i$. Hence $\mathtt{Pr[a_i\ \text{is a match}|A_{i-1}]} = \frac{1}{2}$, for each $1 \le i \le \log T - 1$. Thus,

$$\mathtt{Pr[a_0\ \text{is a match}]} \prod_{\mathtt{i=1}}^{\log \mathtt{T}-1} \mathtt{Pr[a_i\ \text{is a match}|A_{i-1}]} = \frac{1}{2^{\log \mathtt{T}}} = \frac{1}{\mathtt{T}}. \quad (4)$$

Consider choosing a_k sequentially as before, but this time the chosen a_k in addition to satisfying Equation (2), must also represent an actual possible part of a SPRAY path that satisfies Equation (1). We say that a chosen a_k is an exact match if this is the case. Then, it follows that

$$\mathtt{F(x)_j} = \mathtt{Pr[a_0\ \text{is an exact match}]} \times \prod_{\mathtt{i=1}}^{\log \mathtt{T}-1} \mathtt{Pr[a_i\ \text{is an exact match}|A_{i-1}]},$$

where A_{i-i} is the event where all a_k such that $0 \le k \le i-1$ are an exact match. Hence, Inequality (3) holds, and the desired result follows from Equation (4). □

Theorem 2. *For each* $\mathtt{SPRAY}(\log \mathtt{T}-1, \log \mathtt{T}, 1)_\mathtt{j}$, *for any list j in the maximum level of a perfect skip graph, the position of the node on which the operation lands is at most* $\frac{T}{2} + \log T \cdot (T-1) - 1$.

Proof. Consider a perfect skip graph such that $N \ge \frac{T}{2} + T \log T$. We are going to demonstrate that any $\mathtt{SPRAY}(\log \mathtt{T}-1, \log \mathtt{T}, 1)_\mathtt{j}$ operation can reach from its starting position is $\frac{T}{2} + \log T \cdot (T-1)T$ units forward to the right. For any list j in the maximum level of a perfect skip graph, if the path chosen by a $\mathtt{SPRAY}(\log \mathtt{T}-1, \log \mathtt{T}, 1)_\mathtt{j}$ operation moves forward to the right by $\log T$ nodes on the corresponding lists in every possible level, we arrive at a total of $\sum_{i=0}^{\log T-1} 2^i \log T = \log T \sum_{i=0}^{\log T-1} 2^i = \log T \cdot (T-1)$ units forward to the right.

By Proposition 3, if we label the positions of the heads of lists in the maximum level moving from left to right in order of appearance, we arrive at $pos^0_{\log T-1}, pos^1_{\log T-1}, \dots pos^{2^{\log T-1}-1}_{\log T-1}$, where $pos^i_{\log T-1} = i$, for all $0 \le i \le 2^{\log T-1} - 1$. Consider list k whose head is the node in position $pos^{2^{\log T-1}-1}_{\log T-1} = 2^{\log T-1} - 1 = \frac{T}{2} - 1$. Then, by the argument in the previous paragraph, there is a $\mathtt{SPRAY}(\log \mathtt{T}-1, \log \mathtt{T}, 1)_\mathtt{k}$ operation that lands in position $\frac{T}{2} + \log T \cdot (T-1) - 1$. □

We denote our protocol using the $\mathtt{SPRAY}(\log \mathtt{T}-1, \log \mathtt{T}, 1)_\mathtt{j}$ operations on perfect skip graphs as SPRAY_SG, and on perfect skip lists as SPRAY_SL. Finally we denote our deterministic, mark-along protocol on perfect skip graphs as SGMARK. The protocol is shown in Algorithm 12.

Algorithm 12 bool PQ::removeMin(K key).

```
 1: $r =$ random number $[0, t-1]$ where $t$ is the number of threads
 2: if $r =$ threadId then
 3:     ▷ Clean as described in [2]
 4: while true do
 5:     level = MaxLevel
 6:     while level $\ge$ 0 do
 7:         node = next unmarked node
 8:         if node is tail then
 9:             level = level - 1
10:             continue
11:         if node.mark() then
12:             return true
13:         level = level - 1
14:     while performed $\le$ 2 times do
15:         node = next unmarked node
16:         if node is tail then
17:             return false
18:         if node.mark() then
19:             return true
20: return false
```

We conclude that any SPRAY operation in SPRAY_SG, using the same technique from [2], satisfies the same probability bounds as any SPRAY operation with the same parameters in SPRAY_SL. The difference is that the range of furthest reaching SPRAY operation is greater in SPRAY_SG than SPRAY_SL. In particular for SPRAY operations with parameters as in Proposition 2 the range for SPRAY_SG is $\frac{T}{2} + T \cdot \log T$, whereas the range for SPRAY_SL is $T \cdot \log T$. The range of SGMARK is exactly T.

10.1.2. Skip list sprays over T elements

In this section, we define a skip list spray that disperses threads among the same range as SGMARK, and then we indicate that contention is expected to be smaller in a skip graph due to *structural features of the skip graph itself.*

Definition 3. A skip list is **perfect** if the absolute value of the difference in the positions of any two consecutive nodes on level i is 2^i, and every list on level i contains at least $\frac{T}{2^i}$ nodes, for all $0 \le i \le \log T - 1$.

The operation $\mathtt{SPRAY(H, L, D)}$ defines a random walk starting at height $\mathtt{H}$, i.e., at the head of the list at level $\mathtt{H}$ of the skip list. The walk continues forward to the right a uniformly random number of nodes chosen from $\mathtt{[0, L]}$ then moves down the skip list $\mathtt{D}$ levels. This is repeated until we reach the bottom list and (possibly) walk forward to the right one last time. Thus the walk ends in position x of the bottom list.

Definition 4. Let x be the position of any node in a perfect skip list. Then $\mathtt{F_p(x)}$ denotes the probability that, for some fixed $\mathtt{H}$, $\mathtt{L}$, and $\mathtt{D}$, $\mathtt{SPRAY(H, L, D)}$ lands at position x in the bottom list.

Proposition 1. *Let x be the position of any of the first T nodes of a perfect skip list. There is a* $\mathtt{SPRAY}(\log \mathtt{T}-1, 1, 1)$ *that lands at position x in the bottom list.*

Proof. The furthest SPRAY operation moves forward to the right a total of $T-1$ units as follows, $\sum_{i=0}^{\log T-1} 2^i = 2^{\log T} - 1 = T - 1$. Hence we reach the T-th node. Suppose there is some SPRAY operation that reaches a node whose position is k, where $0 < k < T-1$. Consider the following two cases:

- The path determined by the given SPRAY operation consists of a final step forward. By ignoring the final step forward we arrive at a path that reaches position $k-1$.
- The path determined by the given SPRAY operation does not consist of a final step forward. Consider the smallest level i where the path turns left (moving from bottom to top). The preceding node j on the list in level i is in position $k-2^i$. Moving a level down from j then forward to the right every possible level we cover a total of $\sum_{l=0}^{i-1} 2^n = 2^i - 1$ units horizontally and reach position $k-1$. □

Corollary 1. *Fix* $\mathtt{H} = \log \mathtt{T} - 1$, $\mathtt{L} = 1$, *and* $\mathtt{D} = 1$. *Let x be the position of any of the first T nodes of a perfect skip list. Then* $\mathtt{F_p(x)} = \frac{1}{\mathtt{T}}$.

Proof. From Proposition 1 we know that there exist some SPRAY $(\log \mathtt{T} - 1, 1, 1)$ operation that reaches any of the first T nodes, and from its proof we see that no SPRAY operation can reach beyond the first T nodes. Given any position x, where $0 \le x \le T-1$, next we show that the SPRAY operation that lands on position x is path-wise unique.

Fix an arbitrary x, where $0 \le x \le T-1$, and consider a SPRAY$(\log \mathtt{T} - 1, 1, 1)$ operation that lands on the node in position x. Suppose there exists another path-wise distinct SPRAY operation that lands on the node in position x, which we denote by SPRAY$_2$. Consider the largest level i and node j where the paths of the two path-wise distinct SPRAY operations differ. Let the position of node j be k. W.l.o.g. assume the path which belongs to the first SPRAY operation moves forward to the right from node j on level i before descending a level. This ensures that $k + 2^i \le x$. The path induced by the SPRAY$_2$ operation does not move forward to the right and instead directly descends a level from node j. This implies the path can move forward to the right from position k at most $\sum_{n=0}^{i-1} 2^n = 2^i - 1$. Thus the furthest position a path induced by SPRAY$_2$ can reach is $k + 2^i - 1 < x$, which is a contradiction.

Since there is only one unique path that reaches each x, at each level i, where $0 \le i \le \log T - 1$, the probability that any given SPRAY generates the correct portion of the path at level i is exactly $1/2$. Since this probability is independently uniform at each level i, we arrive at $\mathtt{F_p(x)} = \prod_{\mathtt{i=0}}^{\log \mathtt{T}-1} \frac{1}{2} = \frac{1}{2^{\log \mathtt{T}}} = \frac{1}{\mathtt{T}}$. □

Let X denote the number of SPRAY operations in a perfect skip list until every position x, where $0 \le x \le T-1$, is reached. We are interested in the expected number of SPRAY operations which ensure that every position x is reached at least once. To arrive at the expected number of SPRAY operations we show our question of interest is simply the well-known Coupon Collector's Problem [16]. Thus the desired result is proportional to $T \log T$.

10.1.3. Number of successful CAS operations when spraying on skip lists

Now that we defined a skip list spray that disperses threads among a segment of size T, we show that the expected number of operations to mark all elements in that segment is $\Theta(T \log T)$.

Theorem 3. $\mathbb{E}[X]$ *is in* $\Theta(T \log T)$.

Proof. Let X_i denote the number of SPRAY operations performed while exactly $i-1$ positions in the bottom list have been already reached. Then $X = \sum_{i=1}^{T} X_i$ holds. Thus, each X_i is a geometric random variable. If exactly $i-1$ positions have already been reached by previous SPRAY operations then the probability that next SPRAY operation reaches a different position is simply $p_i = 1 - \frac{i-1}{T}$. This is exactly the Coupon Collector's Problem. We outline the analysis for completion and clarity as follows:

$$\mathbb{E}[X] = \mathbb{E}[\sum_{i=1}^{T} X_i] = \sum_{i=1}^{T} \mathbb{E}[X_i] = \sum_{i=1}^{T} \frac{T}{T-i+1}$$
$$= T \sum_{i=1}^{T} \frac{1}{i} = T \cdot H(T),$$

where $H(T)$ is the T-th harmonic number. Using elementary calculus one can show that $H(T) = \ln T + \Theta(1)$ [39, 33] and hence $T \cdot H(T) = T \ln T + \Theta(T)$. Thus it follows that $\mathbb{E}[X]$ is in $\Theta(T \log T)$. □

Corollary 2. *Let* $T \ge 4$. *Then* $\mathbb{E}[X] \ge 2T$.

Proof. We can use well-known elementary techniques to bound $H(T)$ as follows:

$$\begin{aligned} H(T) &= 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{2^{\log T}} \\ &= 1 + \frac{1}{2} + \left(\frac{1}{3} + \frac{1}{4}\right) + \left(\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}\right) \\ &\quad + \left(\frac{1}{2^{\log T - 1} + 1} + \ldots + \frac{1}{2^{\log T}}\right) \\ &\ge 1 + \frac{1}{2} + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) \\ &\quad + \left(\frac{1}{2^{\log T}} + \ldots + \frac{1}{2^{\log T}}\right) \\ &\ge 1 + \frac{1}{2} \log T \end{aligned}$$

Then from Theorem 3 we have that $\mathbb{E}[X] = T \cdot H(T) \ge T\big(1 + \frac{1}{2}\log T\big)$, which gives the desired result for all $T \ge 4$. □

An expected *constant* contention on nodes indicates that the spraying operation in the skip graph produces paths without inducing memory hotspots, which we interpret as a qualitative argument for the spraying protocol over skip graphs.

10.2. Number of successful CAS operations when spraying on skip graphs with our algorithm

We now characterize how a custom relaxed priority queue algorithm, designed precisely in order to exploit the central structural features of the skip graph, can remove elements from a range of T elements with proven contention being exactly 2 *for any number of threads T*.

Definition 5. Let $T = 2^n$, where $n \ge 1$. A perfect skip graph is **minimal** if the number of nodes in the bottom list is exactly T.

Thus each $n \ge 1$ determines a unique minimal skip graph (up to permutation of lists in each level) which we denote be $PSG_{\mathtt{min}}(n)$. Given $PSG_{\mathtt{min}}(n)$, we will show that it contains all smaller perfect minimal skip graphs.

Definition 6. Consider $PSG_{\mathtt{min}}(n)$ and $PSG_{\mathtt{min}}(m)$, where $n > m \ge 1$. Then $PSG_{\mathtt{min}}(n)$ **contains** $PSG_{\mathtt{min}}(m)$, if by ignoring consecutive levels and consecutive nodes of $PSG_{\mathtt{min}}(n)$ we arrive at $PSG_{\mathtt{min}}(m)$ (up to permutation of lists in each level).

Proposition 2. *For all* $n \ge 2$, $PSG_{\mathtt{min}}(n)$ *contains all smaller minimal perfect skip graphs.*

Proof. For $n = 1$ by definition the minimal perfect skip graph is a list with 2 nodes. Consider a list with 4 nodes (2 copies of $PSG_{\texttt{min}}(1)$). Partitioning this list into two other lists with consecutive node positions $0, 2$ and $1, 3$ we arrive at $PSG_{\texttt{min}}(2)$. Let $n = 3$. We will construct $PSG_{\texttt{min}}(3)$ from $PSG_{\texttt{min}}(2)$ in the following way. We glue two copies of $PSG_{\texttt{min}}(2)$ together so that the height of the new structure is the same as $PSG_{\texttt{min}}(2)$ but the bottom list has length 8. It is straightforward to see these are the first 2 levels of $PSG_{\texttt{min}}(3)$. Thus we arrive at $PSG_{\texttt{min}}(3)$ by adding its maximum level.

Suppose it holds for all $n \le k$, for some $k > 3$. Consider $PSG_{\texttt{min}}(k+1)$. Ignoring the maximum level, we see by translational symmetry that the remaining structure consists of two glued copies of $PSG_{\texttt{min}}(k)$. By the induction hypothesis $PSG_{\texttt{min}}(k)$ contains all other smaller minimal perfect skip graphs. □

Let $T = 2^n$ for $n \ge 1$. Then our partition scheme ensures there are exactly two threads acting on the heads of each list of the maximum level of the perfect skip graph at the beginning of SGMARK. In the following arguments we assume that the process of marking a node by one of the threads on level $n-1$ then having the other thread descend to a list on level $n-2$ occurs *simultaneously* for all 2^{n-1} lists k of level $n-1$. More generally, whenever threads are on level i, the ones which fail to mark nodes will *simultaneously* move down a level. Furthermore we assume, that if two threads traversing the same list where the nearest unmarked node j is the same for both, then the threads reach j *simultaneously*. Suppose two threads are acting on every list k of some fixed level i of a perfect skip graph, where $1 \le i \le n-1$, such that for every list k the nearest unmarked node j is the same for both threads. Then we will assume all threads will *simultaneously* reach their respective contested nodes.

Under these assumptions, we can model SGMARK as a game with n rounds. The last round is a special case where the surviving thread will walk forward to the right one last time instead of descending. At the beginning of a regular round all surviving threads up to that round are on level i, where $0 \le i \le n-2$, whereas at the end of the round some threads have moved down a level and have traversed to the right. We will see that it suffices to consider our protocol on minimal perfect skip graphs. First, we will need the following definitions, which are well-defined by Proposition 2.

Definition 7. For all $n \ge 2$, we arrive at the **right hand side copy** of $PSG_{\texttt{min}}(n-1)$ in $PSG_{\texttt{min}}(n)$, by ignoring level $n-1$ of $PSG_{\texttt{min}}(n)$ and the first 2^{n-1} consecutive nodes of the bottom list in $PSG_{\texttt{min}}(n)$.

Definition 8. For all $n \ge 2$, we arrive at the **left hand side copy** of $PSG_{\texttt{min}}(n-1)$ in $PSG_{\texttt{min}}(n)$ via translational symmetry by shifting the **right hand side copy** of $PSG_{\texttt{min}}(n-1)$ in $PSG_{\texttt{min}}(n)$ to the left 2^{n-1} positions.

Proposition 3. *Let $T = 2^n$, where $n \ge 2$. Consider any level i of $PSG_{\texttt{min}}(n)$, where $0 \le i \le n-1$. Consider nodes from left to right in order of appearance and let $pos_i^0, pos_i^1, \dots pos_i^{2^i-1}$ denote the positions of the heads of the respective 2^i list in level i. Then $pos_i^j = j$, for all $0 \le j \le 2^i - 1$.*

Proof. This is trivially true for $n = 2$ by inspection. Suppose it holds for some integer $k > 2$. Consider $PSG_{\texttt{min}}(k+1)$. If we ignore level k of $PSG_{\texttt{min}}(k+1)$, we are left with the left hand side and right hand side copy of $PSG_{\texttt{min}}(k)$ in $PSG_{\texttt{min}}(k+1)$. Therefore, for all levels i such that $0 \le i \le k-1$, the statement holds by the induction hypothesis. By symmetry, the maximum level of $PSG_{\texttt{min}}(k+1)$ is partitioned into two parts, one part of the maximum level located above the left hand side copy of $PSG_{\texttt{min}}(k)$ in $PSG_{\texttt{min}}(k+1)$, and the other above the right hand side copy of $PSG_{\texttt{min}}(k)$ in $PSG_{\texttt{min}}(k+1)$. Since the maximum level of any minimal perfect skip graph contains lists with only two nodes, the portion of each list in the maximal level that is above the left hand side copy of $PSG_{\texttt{min}}(k)$ in $PSG_{\texttt{min}}(k+1)$ contains only one node. These are the 2^k heads of the lists in level k of $PSG_{\texttt{min}}(k+1)$, and since the maximum level is a partition of the nodes on the bottom list, after possibly reordering they have positions $0, 1, \dots 2^k - 1$. □

Proposition 4. *For any $n \ge 2$ and $PSG_{\texttt{min}}(n)$, at the end of the first round of SGMARK on $PSG_{\texttt{min}}(n)$, we have 2 threads acting on the heads of the lists of the maximum level of the right hand side copy of $PSG_{\texttt{min}}(n-1)$ in $PSG_{\texttt{min}}(n)$.*

Proof. Fix $n \ge 2$, where $T = 2^n$. According to SGMARK, at the beginning of the first round there are exactly two threads acting on the heads of each of the lists in level $n-1$. By Proposition 3, positions $pos_{n-1}^0, pos_{n-1}^1, \dots pos_{n-1}^{2^{n-1}-1}$, are marked by 2^{n-1} threads. The remaining 2^{n-1} threads move simultaneously down to level $n-2$. Each list in level $n-2$ is partitioned into two lists in level $n-1$ (first two nodes in each level $n-2$ become heads of lists in level $n-1$), and it follows that for each list k of level $n-2$, we have exactly two threads acting on list k, one for each of the first two consecutive nodes on that particular list. There are now a total of 2^{n-2} threads positions at the heads of all list k in level $n-2$. By Proposition 3 it follows that the threads are in positions $pos_{n-2}^0, pos_{n-2}^1, \dots pos_{n-2}^{2^{n-2}-1}$. Since there is also a thread on the second node of every list k on level $n-2$, for every two threads on any such list k the nearest unmarked node is the next consecutive one. Counting from the heads of the lists on level $n-2$ the third consecutive node will be 2^{n-1} positions away. This implies, that at the end of the first round we will have two threads on each of the positions $pos_{n-2}^{2^{n-1}}, pos_{n-2}^{2^{n-1}+1}, \dots pos_{n-2}^{(2^{n-1}+2^{n-2}-1)}$. These are just the heads of the lists of the maximum level of the right hand side copy of $PSG_{\texttt{min}}(n-1)$ in $PSG_{\texttt{min}}(n)$. □

Corollary 3. *Let $n \ge 2$. The end of the first round of SGMARK on $PSG_{\texttt{min}}(n)$ is equivalent to the start of a first round of SGMARK on the right hand side copy of $PSG_{\texttt{min}}(n-1)$ in $PSG_{\texttt{min}}(n)$.*

Proof. Follows directly from Proposition 4. □

Theorem 4. *Let $n \ge 2$. SGMARK on $PSG_{\texttt{min}}(n)$ ensures that exactly $T = 2^n$ nodes are marked. Furthermore, 2 threads contend for each of the first $T-1$ consecutive nodes, and 1 thread tries to CAS (logically delete) the last node.*

Proof. Let $n = 2$. This means there are 2 threads acting on each of the nodes in position 0 and 1 on the second level of $PSG_{\texttt{min}}(2)$. According to SGMARK, one thread each will mark nodes in positions 0 and 1 respectively, followed by the other two threads that will descend to the bottom list. Then, both threads will move to the right simultaneously to the node in position 2, and one of the threads will succeed in marking the node in position 2. Finally the last thread will move to the right and mark the last node. Thus, two threads contended for each node in positions $0, 1$ and 2, and there was no contention in the last node, since at most 1 thread can try to CAS (logically delete) the last node.

Suppose the claim holds for some $k > 2$. Consider the first round of SGMARK on $PSG_{\texttt{min}}(k+1)$. Following the proof of Proposition 4, we see that the first 2^k nodes are marked at the start of the first round, and there are two threads contending for each of the 2^k nodes that are marked. By Corollary 3 we see that the end of the first round on $PSG_{\texttt{min}}(k+1)$ is the start of the first round on the right hand side copy of $PSG_{\texttt{min}}(k)$ in $PSG_{\texttt{min}}(k+1)$. Therefore by the induction hypothesis SGMARK will mark exactly $2^k + 2^k = T^{k+1}$ nodes. Furthermore 2 threads contend for the first $T^{k+1} - 1$ consecutive nodes, and 1 thread tries to CAS (logically delete) the last node. □

11. Evaluation

We performed experiments in a system with 2 Intel Xeon Platinum 8275CL CPUs, each with 24 cores running at 3.0 GHz (96 hardware threads total). The system has 192 GB of memory and two NUMA nodes. The NUMA-distance tool `numactl -hardware` reports relative intra-node distances of 10 and inter-node distances of 21. In addition to operations/ms throughput measurements, we demonstrate NUMA locality with software instrumentation supporting a graphical visualization of remote accesses. The system runs Ubuntu Linux 18.04 LTS with kernel 4.15.0. We compile tests with `g++ -std=c++11 -O3 -m64 -fno-strict-aliasing`. We used `g++` version 7.5.0.

11.1. Experiment setup

We report the *total* number of operations per millisecond achieved in trials having from 2 up to 96 threads. Each trial is an average of 5 runs of 10 s each, and follows exactly the testing procedure of Synchrobench [22] with the flag `-f 1`. This flag indicates that the testing procedure tries to match each trial's requested percent of *update operations* (inserted and remove) as much as possible, and that only successful inserts or removals count as update operations. The testing procedure, as well as random number generation, are identical to Synchrobench. We run a *read-heavy* (RH) load, with a requested 20% of update operations, a *write-heavy* (WH) load, with a requested 50% of update operations, and a *priority queue* (PQ) load, with 50% of insertions and 50% of removals, all distributed uniformly at random across all threads (except for our load-balancing tests). If X% of operations correspond to *successful* updates[1] in each individual experiment, we say we had X% of *effective updates*, and we report that percentage in each associated graphic caption. Our experiments are defined to be *high contention* (HC) when the key space is 2^8, *medium contention* (MC) when it is 2^{11}, and *low contention* (LC) when it is 2^{17}. The structures are preloaded with 20% of their maximum capacity before any measurements, except for the LC tests, which are preloaded with 2.5%. Because these experiments have even workload among threads, we do not include the load-balancing algorithm (in Sec. 11.5, we have experiments designed exclusively to measure the effectiveness of this particular mechanism). Threads are pinned to each CPU, and we fill a socket before adding threads to another socket. We allocate memory with `libnuma`, in chunks capable of holding 2^{20} objects, in order to amortize the expensive cost of `numa_alloc_local()`. Membership vectors are generated as described in Sec. 3. We obtain data from `/proc/cpuinfo` on Linux, then renumber threads so the larger the absolute difference between thread identifiers $1 \dots T$, the larger the physical distance between their associated CPUs. We consider NUMA domains, core collocation, and hardware-thread collocation in order to access distance and define our membership vectors. Memory allocation/deallocation is done through a custom, NUMA-aware allocator similar to the one in [11], even for the C++ maps representing local structures.

[1] Failed inserts due to pre-existing keys, or failed removals due to absent keys are essentially "contains" operations, as they both return immediately after identifying the respective scenarios above.

In our experiments, we pin threads to cores, which is a standard approach when evaluating NUMA-aware data structures and algorithms. However, the reader may consider a hypothetical situation where pinning is not appropriate or possible. In that case, we could allocate nodes using a membership vector dependent not on the thread's membership vector (now defined in terms of the thread's estimated CPU affinity), but dependent instead on the current running location, still following the protocol described in Sec. 3. Data partitioning would still occur, and if we can employ the reasonable assumption that the OS takes into account thread affinity when scheduling, we would expect to preserve locality. One additional problem with this approach is that our local structures would not only contain local nodes anymore, even with uniform workload among threads. This could be addressed by using our donation mechanism to force the donation of nodes that have been allocated outside the thread's affinity to the appropriate thread. Evaluating with pinned threads makes much more sense because the impact of the OS scheduling is removed, and this is certainly one of the reasons why pinning threads is standard for evaluating NUMA-aware data structures and algorithms. It is important to note, however, that our approach still benefits from hypothetical situations where this might not be possible or appropriate.

11.2. General performance

Figs. 4 and 5 show write-heavy (WH) results for the HC and MC contention scenarios. Read-heavy (RH) results are presented in Appendix B. In our graphs, `layered_map_{sg,ssg}` refers to using C++ `std::map` in conjunction with the hash [3] as local structures, respectively over regular or sparse skip graphs (Sec. 3.3) as shared structures; `lazy_layered_sg` is the lazy variant of `layered_map_sg`; `rotating` is [14], `nohotspot` is [10], and `numask` is [11] as found in Synchrobench's GitHub (mid August 2019). Our experiments did not modify the codebase of `rotating`, `nohotspot`, or `numask` in any way. Our only change was adjusting the Makefile so that the compiling options among these systems and our layered skip graph are matched exactly. In our NUMA setup, using our compiling options, `rotating` and `nohotspot` generally outperform `NUMASK`, although we note that the similar tests in [11] yield better performance for `NUMASK`, when using different compiling options and under a different architectural setup. We invite the reader to consider that such tests are prone to variability when different architectural and compiling options are exercised. For the purpose of isolating individual design components in our analysis, we developed as control: (i) a fine-grain, locked skip list; a concurrent skip list with the same codebase and practices as our skip graph code, including our relink optimization (Sec. 3.6); (ii) a skip graph without layering; and (iii) our layered design over (iii-a) a linked list (`layered_map_ll`) and (iii-b) over a skip list (`layered_map_sl`). The former (iii-a) is essentially a `layered_map_sg` with maximum level 0, and the latter (iii-b) is a `layered_map_ssg` with a single constituent skip list (hence, with no opportunity to implement our partitioning scheme). Non-layered skip lists or skip graphs have maximum level x if the test's key space is of size 2^x, and layered versions follow our partitioning scheme definitions (Sec. 3.2).

With a small key space (HC-WH), `layered_map_ll` performs better than `layered_map_sg` and `layered_map_sl` up to 32 threads, but the performance degrades quickly as we have more threads or the key space gets bigger (MC-WH, Fig. 5; LC-WH, Fig. 6). The reason is that with more threads or bigger key spaces, more elements need to be traversed in the unique linked list upon searches. Then, we could be tempted to say that the multilevel shared structure in `layered_map_sg` is the reason it performs better in MC-WH, but note that `layered_map_sl` performs sim-

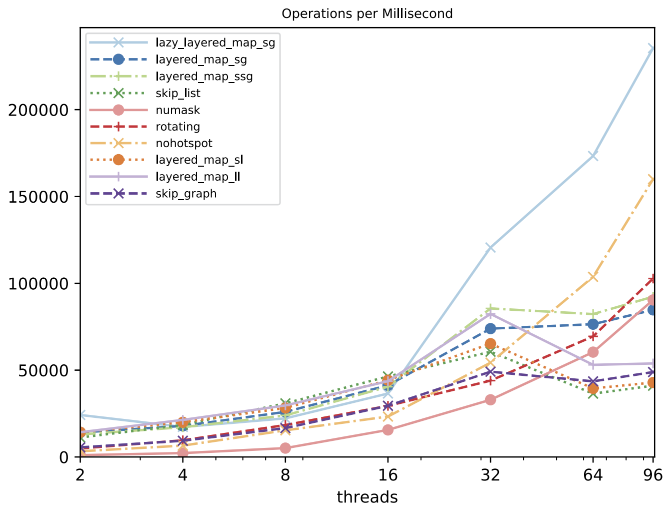

Fig. 4. HC, WH: 32% effective updates.

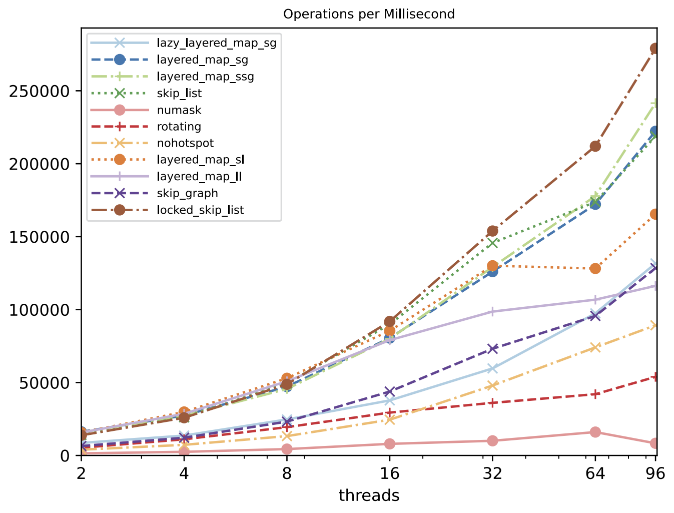

Fig. 6. LC, WH: 4% effective updates.

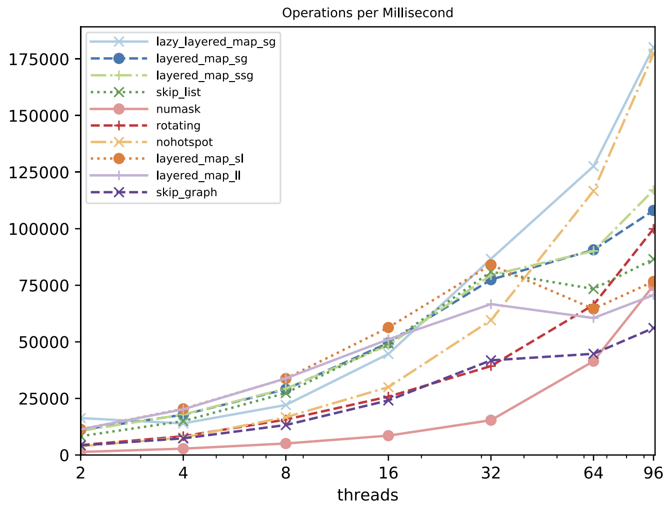

Fig. 5. MC, WH: 32% effective updates.

ilarly than `layered_map_ll` in the same case MC-WH for high thread counts. The *reason* why `layered_map_sg` performs better in the MC-WH scenario is, therefore, the unique differentiating factor: the partitioning scheme in the shared structure (the skip graph). Further, in the same MC-WH scenario, we note a clear difference in performance between `layered_map_ssg` and `layered_map_sl` after 32 threads. The unique differentiating factor here is multiple vs one skip list as a shared structure. So, the existence of multiple, overlapping skip lists, employing a partitioning scheme across threads is the differentiating scalability factor for the good performance of our `layered_map_sg`.

As far as the lazy implementation performance, we see it as a combination of (i) the effectiveness of our partition scheme for increasing NUMA locality and reducing contention (implied above and *verified* in Sec. 11.3); (ii) the commission policy to unlink invalid, marked nodes (isolated right below); and (iii) the fact that with smaller key spaces, threads will more commonly find unmarked nodes through their local hashtable, which performs much better compared to the `std::map` local structure. Under HC-WH, [14] performs well, and our control implementation is comparable to [10,11]. Under MC-WH, [10] performs well, and our control implementation is comparable to [11,14]. In any case, we confirm our expectation that naive skip graphs scale poorly, because while in a skip list the expected number of levels of each node is 2, in a skip graph it is always the maximum. Further, on Table 2, we see how `layered_map_sg`, without any commission period, requires a *lot* more CASes per operation than other structures. With that in mind, and considering our indications that the partitioning scheme works, we have *first* to make sure that skip graphs become *viable* with techniques such as lazy insertions/removals, the commission period, and our relink optimizations mentioned in Sec. 3.5.

We believe the LC-WH scenario is very meaningful from an analytic standpoint. First, it shows that our layered approaches take full advantage of a low contention setting, matching the performance of a locked skip list (expected to work very well in such scenarios).

We attribute this to (i) highly effective local structures, with no overhead of atomic operations or locks; (ii) to the fact that new nodes need to be inserted in only 2 levels on expectation, instead of a fixed 2-6, depending on the number of threads, as prescribed by our partitioning policy; (iii) to a better cache efficiency, documented in Table 1; and (iv) to better NUMA locality (verified in Sec. 11.3, below). The performance of `layered_map_ssg` in fact reveals an interesting tradeoff: although less shared nodes reach the topmost level and get included in the local structures, thus making the starting point of shared operations not as “close” as in the non-sparse version, the number of CAS operations required to insert nodes is substantially lower than `layered_map_sg`. The poor performance of non-layered skip graphs also reflects a higher number of required CAS operations for insertion. We further discuss CAS locality in Sec. 11.3.

Another reason for why the LC-WH scenario is meaningful is that it allows us to visualize the lower performance of `lazy_layered_map_sg` as compared to `layered_map_sg` in this particular setting. Here, we believe the increased performance of the non-lazy version to be a direct consequence of our conservative commission policy to unlink invalid, marked nodes (Sec. 3.5). We think this happens as the commission policy (essentially, a lazy unlink policy) is making the shared structure contain more invalid or marked nodes present in the LC-WH scenario compared to the HC-WH or MC-WH scenarios (and increased cache miss cost). In any case, we do think that the commission policy overhead is a fair price for the highly increased performance in the MC and HC scenarios, which allows us to outperform competitor maps in the LC-WH as well.

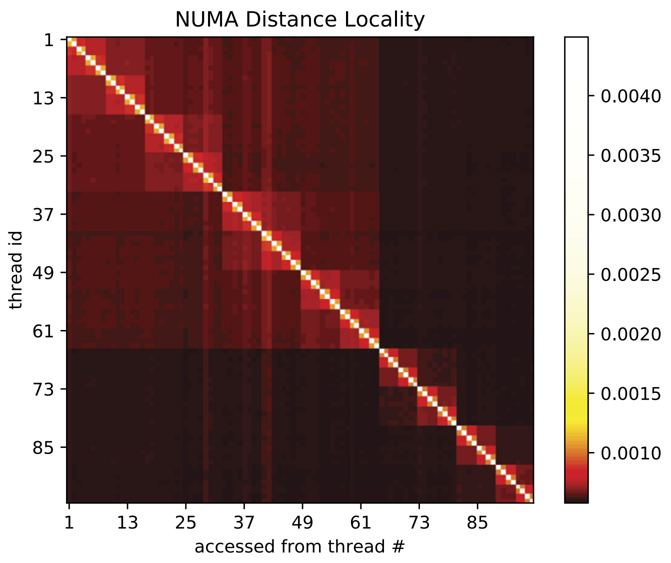

Fig. 7. CAS heatmap: lazy layered skip graphs, MC-WH.

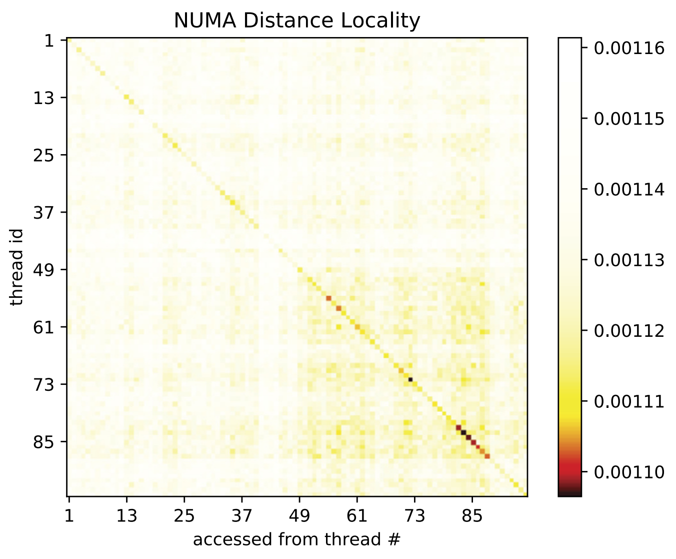

Fig. 8. CAS heatmap: skip lists, MC-WH.

11.3. NUMA locality and contention reduction

We *verify* that our partitioning mechanism promotes better NUMA locality with the experiment below. We define thread membership vectors to be the reversed bit notation of their respective thread IDs, and examine the heatmaps of Figs. 7 and 8, where coordinates (i, j) indicate the distribution of CAS instructions per operation performed by thread i into a node allocated by thread j, instrumented manually on node access functions on the 96-thread MC-WH scenario. The memory access pattern in Fig. 7 shows that the larger the distance between thread IDs, the smaller the number of memory accesses.

Comparing the lazy skip graph and a skip list, the latter serving as control (thus implemented using the same codebase and practices), the heatmaps in Figs. 7 and 8 indicate a *dramatic* increase in CAS NUMA locality on the lazy layered skip graph compared to skip lists. In addition, as expected, the first 64 threads share more accesses among themselves, because the thread membership vectors were defined to be the reversed bit notation of the respective thread IDs. More specifically, the first 64 threads are in the 0-labeled first-level linked list in the skip graph (highest bit 0 in thread IDs, lowest bit 0 in membership vectors), while and the remaining 32 threads are in the 1-labeled first level linked list (highest bit 1 in thread IDs, lowest bit 1 in membership vectors). Referring to Sec. 2, this means that the top-level linked lists in the skip graph, those performing the largest jumps in the search procedure, will navigate among those first 64 threads, giving them slightly more efficient searches. If we added another 32 threads, the graph would become perfectly symmetrical. We conclude that the memory access pattern is a *direct consequence* of our membership vector assignment, which validates precisely our claim that the skip graphs introduce better NUMA locality – as long as we *physically* allocate threads respecting that structured access pattern. In practice, we will define the membership vectors based on the system's characteristics, as defined in Sec. 3, and not simply based on thread ID. Our heatmaps abstract physical placement completely, and are meant to indicate that our partitioning mechanism indeed imposes a particularly well-structured and desirable access pattern. We do provide scripts that read the system's characteristics and generate thread membership vectors upon launch.

Figs. 9 and 10 (for `layered_map_sg` and `layered_map_ssg`) show more clearly the "halving pattern" which is a direct consequence of our thread membership vector allocation. In Fig. 9,

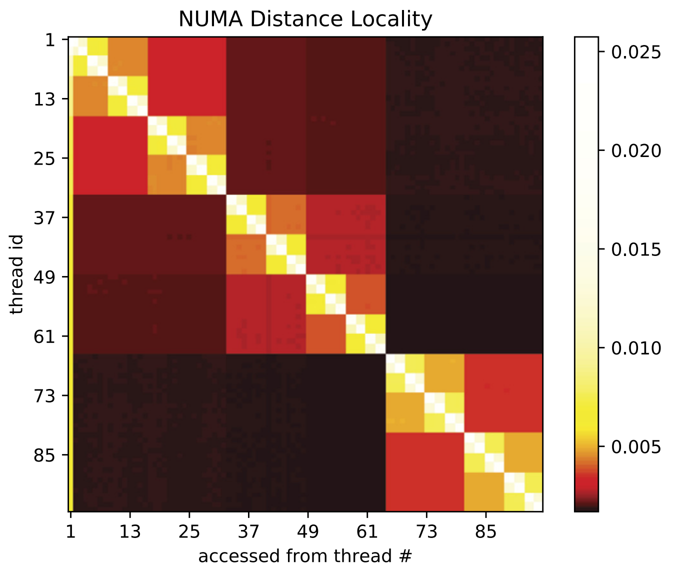

Fig. 9. CAS heatmap: (non-lazy) layered skip graphs, MC-WH.

the vertical line in the left represents accesses to the head of the skip graph, which is located at thread number 1. In addition, the heatmap for the sparse skip graph (Fig. 10) shows a similar pattern than the one for the non-sparse version (Fig. 9), but with a more course granularity. This *directly* reflects our definition of the underlying data structure: as we increase a level in a sparse skip graph, we have only 50% of elements of a shared list redistributed in the corresponding upper level shared lists. So we have twice the sparsity as compared to regular skip graphs, and the sizes of the squares in the halving pattern in Fig. 10 are exactly two times the size of the squares in the corresponding pattern in Fig. 9.

Please note that our heatmaps are all about *distribution* of CAS operations, not their *absolute number*. In fact, if we consider the absolute number of CAS operations, we see that the sparse skip graph performs about 2.84 times *less* CAS operations than the regular skip graph (Table 2). However, `layered_map_sg` and `layered_map_ssg` perform similarly in the MC-WH workload (Fig. 5). We conclude that the performance gain stemming from skip graph sparsity (i.e. a reduced number of CAS operations) must be compensated with a reduced ability to find good starting points

Table 1
Average (instructions & data) cache misses *per operation*, HC-WH, 32 threads. Numbers collected with PAPI.

T	lazy_sg			map_sg			map_ssg			sl		
	L1	L2	L3	L1	L2	L3	L1	L2	L3	L1	L2	L3
8	52.8	12.6	3.1	56.5	12.4	3.1	57.6	12.5	2.7	65.0	15.1	3.1
16	53.6	14.5	3.4	55.9	13.7	3.0	59.0	14.1	3.1	73.1	16.7	3.5
32	87.5	18.7	3.5	73.9	14.5	3.0	93.7	18.1	2.9	93.0	24.6	3.7

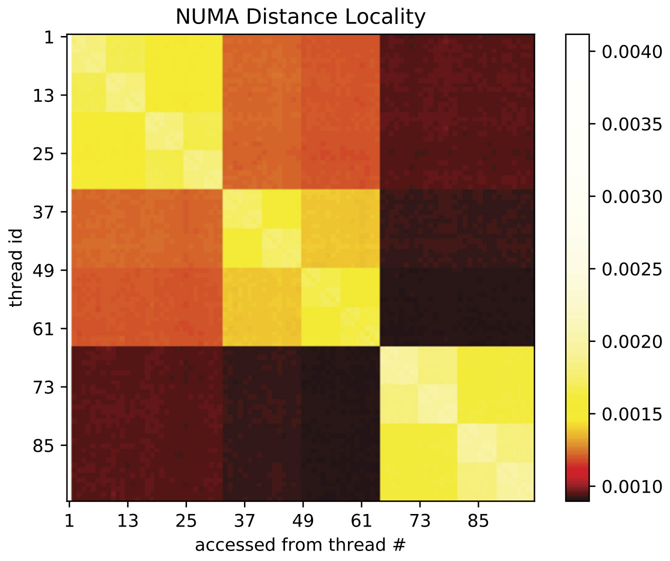

Fig. 10. CAS heatmap: sparse layered skip graphs, MC-WH.

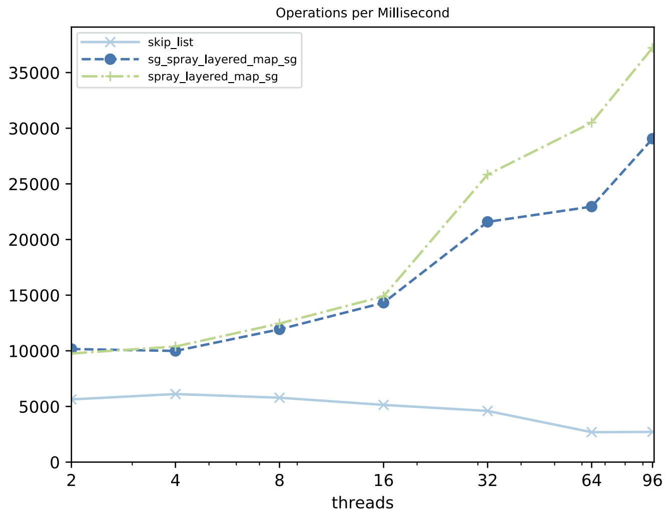

Fig. 11. MC-PQ: 82% effective updates.

Table 2
96 threads, HC-WH. CAS/op does not include uncontended CAS operations upon node insertion. Comparing the lazy map/sg with the skip list, we can observe 6x more CAS locality, 65% less CAS/operation, and substantially better CAS success rate.

	lazy map/sg	map/sg	map/ssg	skip list
loc. reads/op	9.105	8.933	4.264	0.477
rem. reads/op	48.076	54.521	65.123	45.392
loc. CAS/op	0.02508	0.177	0.0137	0.012
rem. CAS/op	0.3493	2.524	0.888	1.113
CAS succ. rate	0.999	0.986	0.982	0.883

for shared operations, as expected for sparse skip graphs. This should increase the number of cache misses, compensating for the reduction of CAS operations. We confirm this explanation in Table 1, which shows a clearly higher cache-miss ratio for sparse skip graphs as compared to non-sparse versions. This table also shows that our layered skip graph has a reduction of 21% in L1 misses, 41% in L2 misses, and 18% in L3 misses with 32 threads as compared to skip lists. We except fewer caches misses due to our partition scheme, which has been designed precisely to increase locality in the shared structure operations.

Related to contention, Table 2 shows additional metrics collected via manual code instrumentation, on the 96-thread HC-WH scenario. Both our heatmaps and Table 2 *do not count* CAS, read, or write operations performed over an inserting node, otherwise locality would be artificially inflated with operations that are inherently local, as threads have to initialize their allocated nodes. Any CAS operation metric presented is a *maintenance CAS*: an operation required to link, unlink, or cleanup nodes.

Although `lazy_layered_map_sg` performs slightly more reads per operation than skip lists, it performs 68% less remote maintenance CASes per operation. The CAS success rate is substantially higher (99% in `lazy_layered_map_sg` vs. 88.3% in the skip list). Both the increase in NUMA locality, as previously discussed, and the contention reduction, are attributed to our *partitioning scheme*, designed precisely with those goals in mind (Sec. 3.2). Note that atomic writes are only used to initialize nodes before insertion. Hence, these operations are all contention-free and 100% local, so they are not measured.

11.4. Relaxed priority queues

Fig. 11 tests multiple implementations for relaxed priority queues using skip graphs. The `spray` implementation consists of the application of the spraying technique of [2] over skip graphs; the `sg_spray` implementation is our custom protocol that traverses the skip graph deterministically, marking elements along this traversal (Algorithm 12). We also have a *control* skip list (implemented using the same codebase and practices) where we perform spray operations as in [2] but in skip lists, not skip graphs, and without our layering approach.

We note that `spray` scales better than `sg_spray`. The reason for the better scalability, although Theorem 4 indicate that `sg_spray` is subject to a very small contention, is that the range of `spray` is slightly larger (Theorem 2). Indeed, Fig. 12 shows an experiment similar to the one in [2], where we perform traversals and only note which nodes would be marked, without actually marking any element. The experiment shows that `sg_spray` is less relaxed than `spray`, so the reduced scalability of `sg_spray` is explained with the experiment of Fig. 11. In conjunction, Figs. 11 and 12 essentially exhibit a tradeoff between priority queue relaxation and scalability.

11.5. Load balancing

In order to evaluate load balancing, we show (i) two experiments measuring the effectiveness of the load balancing protocol (that is, making local structures similar in size, containing elements evenly distributed across all key space); and (ii) an experiment demonstrating the overhead of the mechanism.

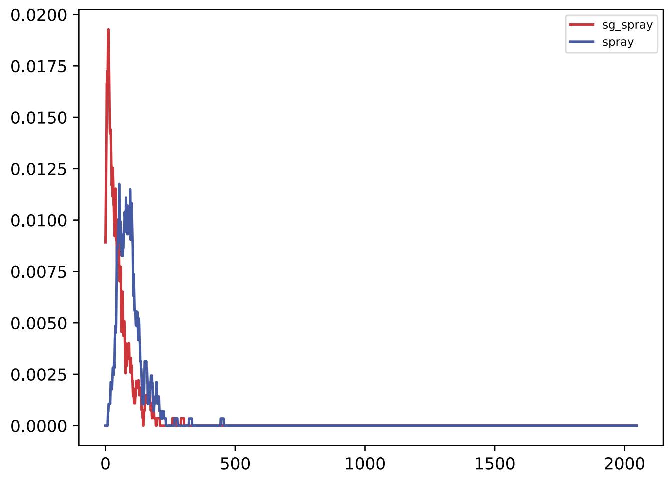

Fig. 12. MC-PQ: Key range of removed elements (%).

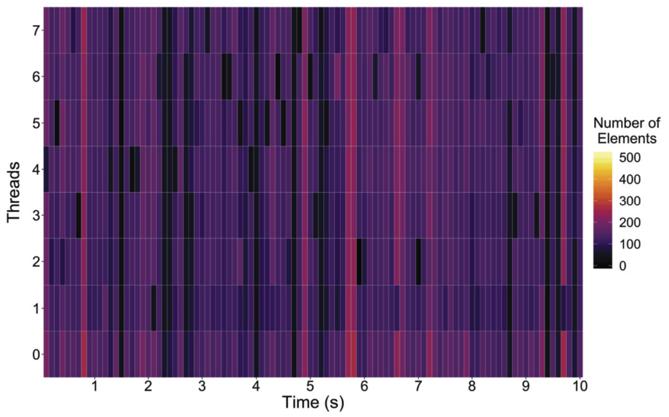

Fig. 13. Local structure sizes, keyspace = 2048. The color indicates the number of elements per thread, per time.

11.5.1. Donation effectiveness

As far as the *effectiveness* of the protocol, we consider two *extremal* experiments: (i) a scenario where only thread T_0 inserts, and all others perform removals and contains operations in an MC-WH experiment (done in Fig. 13); and (ii) a scenario where 1/2 of the threads insert only on the 0-1023 range of a 2048 keyspace, and 1/2 of the threads insert only on the 1024-2047 range of the keyspace (done in Fig. 14). We use $p = 1$ for the "aggressiveness factor" in the load balancing protocol. In both experiments described below, we run the background thread in a hardware thread separate from the application threads.

We run experiments for 10 s, each starting with the extremal load imbalance described above, and we take snapshots of the element distribution among threads at 100 discrete time points during this interval. Fig. 13 shows that the *sizes* of the local structures are similar in the 10-spaced time points in our experiment (we chose to graph one time point per second for the sake of better visualization). Specifically, the X-axis represents time, varying from 0-10 s, the Y-axis represents 8 different threads, and the color indicates the local structure size (not counting marked elements) for each thread. Note that even after our first snapshot, the work distribution is already similar among the threads – a consequence of the "aggressiveness" factor of 1 used in our experiments.

Fig. 14 shows that the key distribution of two threads that belong to two different groups are spread throughout the *whole* element space. In the figure, the X-axis represents possible keys, the Y-axis represents 10 discrete time points, one per second, and the color represents whether a particular key belongs to thread 1 or

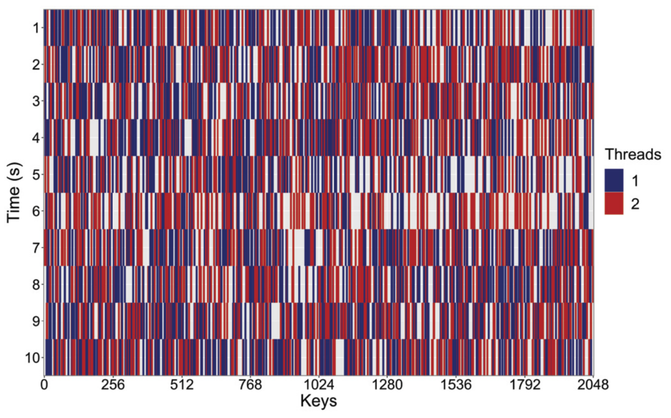

Fig. 14. Key distribution, 2048 keyspace. The color indicates the owning thread, per key, per time.

2 in their local structure. Once again, even after the first second, the key distribution is already roughly balanced among the two threads, despite their extremely uneven key distribution for insertions. In terms of performance, we verified experimentally that our load balancing mechanism has a 20% of scalability impact at the highest thread count we tested (96 threads) (and smaller impact for smaller thread counts). We note that we designed our protocol so that it gives total flexibility to the programmer: the load balancing can be turned off completely if the application has a uniform operation distribution among threads, or the aggressiveness factor can be otherwise adjusted to the imbalance expected by the application.

11.5.2. Load imbalance overhead

In order to have a sense of how much overhead the load balancing mechanism introduces, we conducted experimental trials under a MC-WH workload (25% of insertion, 25% of removals, and 50% of contains operations, with a 2^{11} key space size). The tests were run for 10 s before terminating. We vary the number of threads from 4 to 95 (leaving one core running the background thread exclusively). We compare two executions: one labeled `synchrobench` and one labeled `load_bench`. The `synchrobench` protocol runs the `lazy_stacked_map_sg` without the load balancing mechanism, following the same experimental setup as discussed in the previous sections. We remind the reader that `synchrobench` tries to match the user specified percentages of insertions, removals, and containment operations as much as possible by uniformly distributing update operations across all threads. That is, when the user specifies 25% insertions, 25% removals, and 50% contains operations, the `synchrobench` protocol assigns all threads to make 25% of their operations insertions, 25% of their operations removals, and the rest contains. This test is run without the load balancing mechanism to show the performance of the lazy stacked data structure when all threads are inserting equally and have equal local structure sizes. The `load_bench` protocol is identical to `synchrobench` except for a few modifications. First, they run `lazy_stacked_map_sg` *with* the load balancing mechanism, launching the concurrent background thread in addition to the specified number of worker threads. Additionally, `load_bench` tries to match the user specified percentages of operations by distributing operations *between* threads, rather than *across* threads. That is, if the program specifies 25% insertions, 25% removals, and 50% contains operations, `load_bench` assigns 25% of threads to *only* do insertions, 25% of threads to do only removals, and the other 50% of threads to do only contains operations, with thread identities chosen randomly. Our test aims to show that the load balancing mechanism allows `lazy_stacked_map_sg` to be viable even when substan-

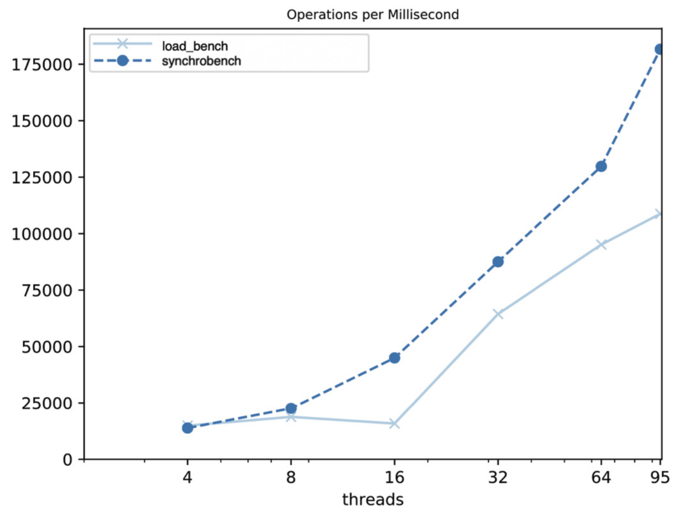

Fig. 15. MC, WH: comparing a load balanced setup (`synchrobench`), with one where the workload among threads is unbalanced (`load_bench`).

tial work imbalance is introduced, yet still providing a reasonable comparison as the percentage distribution of operation types is identical when we consider all threads in the system.

The experiments described above are presented in Fig. 15. Note that the load imbalance that we introduce in the `load_bench` test is *extreme*, as most threads (75% of them) would have *empty* local structures if the load balancing mechanism was disabled. The cost of some operations would be prohibitive in that case, because without load balancing we could not even guarantee searches in expected logarithmic time: 75% of threads would start their search in the head of the skip graph. With the load balancing mechanism, we are able to scale up to 100000 operations per millisecond even with such high imbalance, matching the performance of our non-lazy stacked skip graph (Fig. 5).

12. Conclusion

We presented a technique to promote NUMA-aware data parallelism inside the concurrent data structure, bringing significant quantitative and qualitative improvements on NUMA locality, as well as reduced contention for synchronized memory accesses. Our design is based on integrating thread-local sequential maps with skip graphs, while performing a data partitioning scheme over the skip graphs for increased NUMA locality. By "qualitative" increase in NUMA locality, we mean that remote memory accesses are not only reduced in number, but the larger the distance between threads in the system, the larger the reduction is. We provide an optional load-balancing mechanism for applications where the types of operation are not uniformly distributed among threads. Our load balancing mechanism is based on donating inserted nodes across thread-local indexes of all threads, coordinated by a single background thread.

For relaxed priority queues, we consider two alternative implementations: (a) using "spraying", a well-known random-walk technique usually performed over skip lists, but now performed over skip graphs; and (b) a custom protocol that traverses the skip graph deterministically, marking elements along this traversal. We provide formal arguments indicating that the second approach is slightly more *relaxed*, that is, that the span of removed keys is larger, yet shows smaller contention and higher scalability. We also explore the design of an additional skip graph variant, called *sparse skip graph*, with benefits for low-contention/low-update settings, and provide an optional mechanism for handling non-uniform work distributions.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jpdc.2022.04.006.

References

[1] D. Akkoorath, J. Brandão, A. Bieniusa, C. Baquero, Global-local view: scalable consistency for concurrent data types, in: M. Aldinucci, L. Padovani, M. Torquati (Eds.), Euro-Par 2018: Parallel Processing, Springer International Publishing, Cham, 2018, pp. 492–504.
[2] D. Alistarh, J. Kopinsky, J. Li, N. Shavit, The spraylist: a scalable relaxed priority queue, in: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, ACM, New York, NY, USA, 2015, pp. 11–20, http://doi.acm.org/10.1145/2688500.2688523.
[3] M. Ankerl, link github.com/martinus/robin-hood-hashing.
[4] J. Aspnes, G. Shah, Skip graphs, ACM Trans. Algorithms 3 (4) (Nov. 2007), https://doi.org/10.1145/1290672.1290674, http://doi.acm.org/10.1145/1290672.1290674.
[5] S. Blagodurov, A. Fedorova, S. Zhuravlev, A. Kamali, A case for NUMA-aware contention management on multicore systems, in: 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2010, pp. 557–558.
[6] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, M. Moir, Message passing or shared memory: evaluating the delegation abstraction for multicores, in: R. Baldoni, N. Nisse, M. van Steen (Eds.), Principles of Distributed Systems, Springer International Publishing, 2013, pp. 83–97.
[7] I. Calciu, J. Gottschlich, M. Herlihy, Using elimination and delegation to implement a scalable NUMA-friendly stack, in: 5th USENIX Workshop on Hot Topics in Parallelism, USENIX, San Jose, CA, 2013, https://www.usenix.org/conference/hotpar13/workshop-program/presentation/Calciu.
[8] I. Calciu, H. Mendes, M. Herlihy, The adaptive priority queue with elimination and combining, in: F. Kuhn (Ed.), Distributed Computing, in: Lecture Notes in Computer Science, vol. 8784, Springer, Berlin/Heidelberg, 2014, pp. 406–420.
[9] I. Calciu, S. Sen, M. Balakrishnan, M.K. Aguilera, Black-box concurrent data structures for NUMA architectures, SIGPLAN Not. 52 (4) (2017) 207–221, https://doi.org/10.1145/3093336.3037721, http://doi.acm.org/10.1145/3093336.3037721.
[10] T. Crain, V. Gramoli, M. Raynal, No hot spot non-blocking skip list, in: Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems, ICDCS '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 196–205.
[11] H. Daly, A. Hassan, M.F. Spear, R. Palmieri, NUMASK: high performance scalable skip list for NUMA, in: U. Schmid, J. Widder (Eds.), 32nd International Symposium on Distributed Computing (DISC 2018), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 121, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2018, pp. 18:1–18:19, http://drops.dagstuhl.de/opus/volltexte/2018/9807.
[12] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, M. Roth, Traffic management: a holistic approach to memory placement on NUMA systems, Comput. Archit. News 41 (1) (2013) 381–394, https://doi.org/10.1145/2490301.2451157, http://doi.acm.org/10.1145/2490301.2451157.
[13] D. Dice, V.J. Marathe, N. Shavit, Lock cohorting: a general technique for designing NUMA locks, SIGPLAN Not. 47 (8) (2012) 247–256, https://doi.org/10.1145/2370036.2145848, http://doi.acm.org/10.1145/2370036.2145848.
[14] I. Dick, A. Fekete, V. Gramoli, A skip list for multicore, Concurr. Comput. 29 (4) (2017) e3876, https://doi.org/10.1002/cpe.3876, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3876.
[15] P. Fatourou, N.D. Kallimanis, Revisiting the combining synchronization technique, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, ACM, New York, NY, USA, 2012, pp. 257–266, http://doi.acm.org/10.1145/2145816.2145849.
[16] M. Ferrante, M. Saltalamacchia, The coupon collector's problem, Mater. Mat. 2014 (2) (2014) 1–35.
[17] M. Fomitchev, E. Ruppert, Lock-free linked lists and skip lists, in: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC '04, ACM, New York, NY, USA, 2004, pp. 50–59, http://doi.acm.org/10.1145/1011767.1011776.
[18] S. Ghemawat, P. Menage, Tcmalloc: thread-caching malloc, goog-perftools.sourceforge.net.
[19] J. Giceva, G. Alonso, T. Roscoe, T. Harris, Deployment of query plans on multicores, Proc. VLDB Endow. 8 (3) (2014) 233–244, https://doi.org/10.14778/2735508.2735513.

[20] E. Gidron, I. Keidar, D. Perelman, Y. Perez, Salsa: scalable and low synchronization numa-aware algorithm for producer-consumer pools, in: Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 151–160.
[21] M.T. Goodrich, M.J. Nelson, J.Z. Sun, The rainbow skip graph: a fault-tolerant constant-degree distributed data structure, in: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm, SODA '06, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006, pp. 384–393, http://dl.acm.org/citation.cfm?id=1109557.1109601.
[22] V. Gramoli, More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms, in: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, ACM, New York, NY, USA, 2015, pp. 1–10, http://doi.acm.org/10.1145/2688500.2688501.
[23] N.J.A. Harvey, B. Jones, S. Saroiu, Skipnet: a scalable overlay network with practical locality properties, in: Proceedings of USENIX Symposium on Internet Technologies and Systems, 2003.
[24] D. Hendler, N. Shavit, L. Yerushalmi, A scalable lock-free stack algorithm, in: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '04, ACM, New York, NY, USA, 2004, pp. 206–215, http://doi.acm.org/10.1145/1007912.1007944.
[25] D. Hendler, I. Incze, N. Shavit, M. Tzafrir, Flat combining and the synchronization-parallelism tradeoff, in: Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '10, ACM, New York, NY, USA, 2010, pp. 355–364, http://doi.acm.org/10.1145/1810479.1810540.
[26] T.A. Henzinger, C.M. Kirsch, H. Payer, A. Sezgin, A. Sokolova, Quantitative relaxation of concurrent data structures, in: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 317–328.
[27] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, 2008.
[28] M. Herlihy, N. Shavit, On the nature of progress, in: A. Fernàndez Anta, G. Lipari, M. Roy (Eds.), Principles of Distributed Systems, Springer, Berlin, Heidelberg, 2011, pp. 313–328.
[29] M. Herlihy, J. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (1990) 463–492, https://doi.org/10.1145/78969.78972.
[30] M. Herlihy, Y. Lev, V. Luchangco, N. Shavit, A simple optimistic skiplist algorithm, in: G. Prencipe, S. Zaks (Eds.), Structural Information and Communication Complexity, Springer, Berlin, Heidelberg, 2007, pp. 124–138.
[31] M. Herlihy, D. Kozlov, S. Rajsbaum, Distributed Computing Through Combinatorial Topology, Morgan Kaufmann, 2013.
[32] D. Klaftenegger, K. Sagonas, K. Winblad, Queue delegation locking, IEEE Trans. Parallel Distrib. Syst. 29 (3) (2017) 687–704, https://doi.org/10.1109/TPDS.2017.2767046.
[33] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, A. Kemper, Massively parallel NUMA-aware hash joins, in: A. Jagatheesan, J. Levandoski, T. Neumann, A. Pavlo (Eds.), Memory Data Management and Analysis, Springer International Publishing, 2015, pp. 3–14.
[34] V. Leis, P. Boncz, A. Kemper, T. Neumann, Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, ACM, New York, NY, USA, 2014, pp. 743–754, http://doi.acm.org/10.1145/2588555.2610507.
[35] Y. Lev, M. Herlihy, V. Luchangco, N. Shavit, A provably correct scalable skiplist (brief announcement), in: Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS 2006), 2006.
[36] Y. Li, I. Pandis, R. Müller, V. Raman, G.M. Lohman, NUMA-aware algorithms: the case of data shuffling, in: CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings, 2013, https://www.microsoft.com/en-us/research/publication/numa-aware-algorithms-the-case-of-data-shuffling/.
[37] H. Mendes, C.G. Fernandes, A concurrent implementation of skip graphs, Electron. Notes Discrete Math. 35 (2009) 263–268, https://doi.org/10.1016/j.endm.2009.11.043.
[38] Z. Metreveli, N. Zeldovich, M.F. Kaashoek, Cphash: a cache-partitioned hash table, in: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, ACM, New York, NY, USA, 2012, pp. 319–320, http://doi.acm.org/10.1145/2145816.2145874.
[39] M. Mitzenmacher, E. Upfal, Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis, Cambridge University Press, 2017.
[40] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, in: Workshop on Algorithms and Data Structures, 1989, pp. 437–449, citeseer.ist.psu.edu/pugh90skip.html.
[41] W. Pugh, Concurrent maintenance of skip lists, Tech. rep., University of Maryland at College Park, 1990.
[42] N. Shavit, I. Lotan, Skiplist-based concurrent priority queues, in: IEEE International Symposium on Parallel and Distributed Processing, 2000, pp. 263–268.
[43] H. Sundell, P. Tsigas, Fast and lock-free concurrent priority queues for multi-thread systems, in: IEEE International Symposium on Parallel and Distributed Processing, 2003, 11 pp.
[44] J. Talbot, R.M. Yoo, C. Kozyrakis, Phoenix++: modular MapReduce for shared-memory systems, in: Proceedings of the Second International Workshop on MapReduce and Its Applications, MapReduce '11, ACM, New York, NY, USA, 2011, pp. 9–16, http://doi.acm.org/10.1145/1996092.1996095.
[45] S. Thomas, H. Mendes, Brief announcement: layering data structures over skip graphs for increased numa locality, in: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 422–424.
[46] S. Thomas, R. Hayne, J. Pulaj, H. Mendes, Using skip graphs for increased NUMA locality, in: 32nd IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2020, Porto, Portugal, September 9–11, 2020, IEEE, 2020, pp. 157–166.
[47] M. Wagle, D. Booss, I. Schreter, D. Egenolf, NUMA-aware memory management with in-memory databases, in: R. Nambiar, M. Poess (Eds.), Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things, Springer International Publishing, 2016, pp. 45–60.
[48] M. Wimmer, F. Versaci, J.L. Träff, D. Cederman, P. Tsigas, Data structures for task-based priority scheduling, in: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 379–380.
[49] M. Wimmer, J. Gruber, J.L. Träff, P. Tsigas, The lock-free k-lsm relaxed priority queue, in: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, ACM, New York, NY, USA, 2015, pp. 277–278, http://doi.acm.org/10.1145/2688500.2688547.

Samuel Thomas completed his undergraduate degree with High Honors in Computer Science at Davidson College in 2020. He is currently an Ph.D. student at Brown University. This work was done while he was an undergraduate student at Davidson College.

Roxana Hayne completed her undergraduate degree with High Honors in Computer Science at Davidson College in 2020. She is currently working in the industry in the United States. This work was done while she was an undergraduate student at Davidson College.

Jonad Pulaj is an Assistant Professor of Mathematics and Computer Science at Davidson College. He received his Ph.D. in Mathematics from Technische Universität Berlin in 2017, and completed his undergraduate degree in Mathematics at UNC Chapel Hill in 2009.

Hammurabi Mendes is an Assistant Professor of Mathematics and Computer Science at Davidson College. He received his Ph.D. in Computer Science from Brown University in 2016. He also completed a Master's Degree in Computer Science at the University of São Paulo, Brazil, and his undergraduate degree at the University of Brasília, Brazil.