PIPQ: Strict Insert-Optimized Concurrent Priority Queue

Olivia Grimes
Lehigh University, Bethlehem, PA, USA

Ahmed Hassan
Lehigh University, Bethlehem, PA, USA

Panagiota Fatourou
FORTH ICS, Heraklion, Greece
Department of Computer Science, University of Crete, Heraklion, Greece

Roberto Palmieri
Lehigh University, Bethlehem, PA, USA

Abstract

This paper presents PIPQ, a strict and linearizable concurrent priority queue whose design differs from existing solutions in literature because it focuses on enabling parallelism of insert operations as opposed to accelerating delete-min operations, as traditionally done. In a nutshell, PIPQ's structure includes two levels: the worker level and the leader level. The worker level provides per-thread data structures enabling fast and parallel insertions. The leader level contains the highest priority elements in the priority queue and can thus serve delete-min operations. Our evaluation, which includes an exploration of different data access patterns, operation mixes, runtime settings, and an integration into a graph-based application, shows that PIPQ outperforms competitors in a variety of cases, especially with insert-dominant workloads.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Priority Queue, Concurrent Data Structures, Synchronization

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.35

Related Version *Full Version*: `https://arxiv.org/abs/2508.16023`

Supplementary Material *Software (Source Code)*: `https://github.com/sss-lehigh/pipq` [12]

Funding This material is based upon work supported by the National Science Foundation under Grant No. CNS-2045976 and by the Greek Ministry of Education, Religious Affairs and Sports through the HARSH project (project no. ΥΠ3ΤΑ - 0560901), which is carried out within the framework of the National Recovery and Resilience Plan "Greece 2.0" with funding from the European Union - NextGenerationEU.

Acknowledgements The authors would like to acknowledge Mike Spear for all his input, feedback, and support.

1 Introduction

Concurrent data structures are at the heart of modern applications and thus, their efficient and scalable implementation is of great importance. Priority queues, particularly, are widely used data structures with many applications in diverse domains [5, 24, 20, 21, 22, 32, 3, 13, 6]. Applications include graph and encoding algorithms, big data analysis, task scheduling and load balancing, event-driven simulation, query optimization, databases (e.g., in transaction management, deadlock handling, process handling), network routing, and more.

A priority queue supports two operations: *insert*, which inserts a key-value pair in the priority queue, and *delete-min*, which deletes and returns the element with the smallest key (indicating the *highest priority*) in the queue.

The last decade has been characterized by innovations in the design of priority queues focusing mainly on improving the performance of delete-min [18, 1, 30, 26, 24, 33], which is unquestionably a sequential bottleneck for most designs, since these operations all target the same highest priority element. These innovations focus mostly on relaxing the semantics of the priority queue itself to allow for more parallelism when serving the delete-min operation. In relaxed designs, the rank of the removed element is within some defined bound of the highest priority element, although it may not be the highest priority element itself. However, relaxed semantics are not sufficient for many applications in which the single highest priority element must be removed each time. Some examples that may require strict semantics include risk management systems to ensure critical tasks are addressed first, real-time systems such as medical systems and devices, which must optimally treat patients, and in security applications in which high-priority messages or threats must be processed ahead of all other requests.

Despite the highlighted importance of such strict behavior of priority queues in many applications, the focus on relaxed priority queues in the last decade left a clear gap in literature and made the performance of state-of-the-art strict priority queues significantly inferior to their relaxed counterparts. However, if relaxation is not a viable option, the performance of applications needing strict semantics will always be limited by the underlying priority queue. In this paper, we fill this gap as follows. Optimizing on delete-min in the strict setting is difficult due to its sequential nature. Thus, while still aiming to alleviate the performance impact of the pessimistic nature of delete-min, we shift our focus to maximizing the performance and parallelism of insert operations. In fact, achieving a significant speedup for insert operations can offset the overhead of the delete-min operations. Beyond that, there is a class of applications for which the performance of insert operations is important. Examples include graph analytics [17], data series analysis [5, 20, 21, 22], and computation over streamed data [23].

The core intuition is to improve the performance of insert operations by using a set of data structures, one per thread, to avoid synchronization with other threads when the inserted element is not among those with the highest priority. When this condition is met, which we claim to be the common case for many applications [17, 5, 20, 21, 22, 23], insertions can effectively be embarrassingly parallel. This ensures perfect scalability, Non-Uniform Memory Access (NUMA)-awareness, and near-optimal performance.

We developed this intuition into PIPQ[1], our linearizable, concurrent priority queue whose main strength is its high-performance insert operation. PIPQ's design encompasses a hierarchical structure:

- In the first level, called the *worker level*, we deploy one min-heap [16] per thread. Insertions are performed to the thread's min-heap unless the thread has observed the highest priority element to be inserted.
- The second level, called the *leader level*, collects the highest-priority elements in PIPQ, hence creating a set of candidate elements to be returned by delete-min.

Delete operations remove elements from the leader level, which contains the most minimal element(s) inserted by each thread, and thus the most minimal element(s) in the entire structure. To avoid degrading the performance of delete-min due to this hierarchical structure, we employ the combining technique [15, 9] to minimize contention between concurrent removals.

[1] Pronounced "Pip Q", standing for Parallel Inserts Priority Queue.

We develop PIPQ in C++[2] and compare its performance against two state-of-the-art strict linearizable priority queues, namely the Lindén-Jonsson priority queue [18], and the Lotan-Shavit priority queue [28]. Experiments are conducted on an Intel server equipped with 96 cores over four processors. Applications include a microbenchmark, in which threads execute a mix of workloads, and benchmarks that mimic the access patterns generated by recent applications of priority queues, in which inserting is a dominating factor. Evaluation of the widely used single-source shortest path (SSSP) algorithm is additionally included in the extended version of the paper. Across experiments, our evaluation reveals that the proposed algorithm exhibits great performance benefits compared to competitors when insertions dominate the workload. Importantly, PIPQ does not sacrifice performance in workloads exhibiting an equal split of both operations, nor in delete-min heavy workloads.

2 Related Work

Lotan and Shavit [28] were the first to use a Skiplist [25] as a concurrent priority queue, a design which has since largely become the standard for implementing concurrent priority queues. Notably, the Lotan and Shavit [28] concurrent priority queue is not linearizable, instead achieving the weaker condition of being quiescently consistent [16]. Sundell and Tsigas [29] created a lock-free, linearizable concurrent priority queue based on a skip-list. Lindén and Jonsson then presented their strict, linearizable priority queue algorithm [18]; their enhancement of Lotan and Shavit's priority queue to the delete-min operation delays physical deletion of elements to improve performance. The evaluation provided by Lindén and Jonsson shows that their priority queue consistently outperforms that of Sundell and Tsigas for various types of experiments. Braginsky et al. [4] similarly use a skiplist-based design, modifying data nodes to be lock-free chunks and leveraging elimination to speed up concurrent delete-mins, and inserts that insert to the first chunk. While PIPQ is optimized to increase the parallelism of insert operations, the work in [4] still uses a non-thread-local structure but reduces synchronization through coalescing elements. Zhang and Dechev [31] implement a lock-free priority queue based on a multi-dimensional list. Their implementation is only quiescently consistent. Similar to these solutions, PIPQ aims to create a strict linearizable priority queue, but with a new design which does not include use of a skip-list and aims to optimize insertions, all while achieving linearizability.

More recent work on priority queues has focused on relaxed solutions. In these designs, the element removed by a delete-min may not be the highest priority element itself, but is within some defined bound of it. Spraylist [1] is a well-known relaxed priority queue, and many other relaxed solutions have been proposed [30, 33, 27, 26, 24]. Unlike these relaxed solutions that optimize for the delete-min operation, PIPQ provides a strict priority queue that optimizes for insert.

A recent related work of interest is the Stealing Multi-Queue (SMQ) priority scheduler by Postnikova et al. [24], which innovates on the prior Multi-Queues work [26]. SMQ uses a design similar to that of the worker level of PIPQ, such that it maintains per-thread heaps. As a result, insert operations are optimized, and, in addition to that, it relaxes the semantics of the delete-min operation with the use of the so-called stealing buffers. Despite their similarities, removing the relaxation of SMQ would require additional synchronization, which is effectively the goal accomplished by PIPQ.

[2] The source code for PIPQ is publicly available here: `https://github.com/sss-lehigh/pipq`

3 Overview of PIPQ

PIPQ is a linearizable priority queue that provides the traditional priority queue API: `Insert`, which inserts some key-value pair, and `DeleteMin`, which removes the element with the smallest (i.e., highest priority) key in the structure. Our priority queue supports duplicate keys and/or values. Figure 1 illustrates PIPQ's architecture, including an example instantiation relevant to a specific thread (red / bolded).

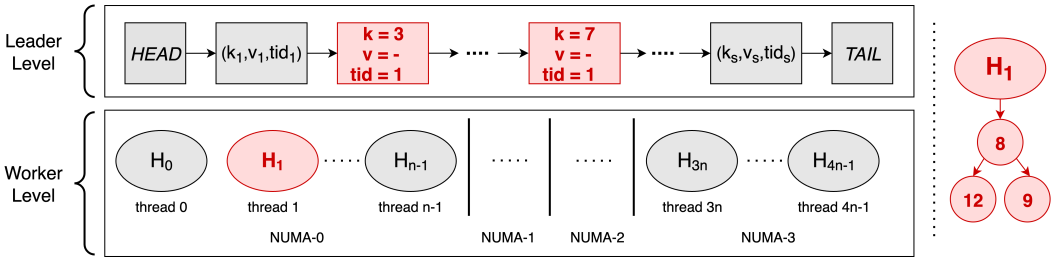


Figure 1 PIPQ on a 4-NUMA node testbed. Each oval represents a per-thread worker-level min-heap; "n" represents the number of threads per NUMA node. Squares represent nodes of the leader-level linked list; "s" represents the size of the leader-level linked list.

To create highly parallel insertions, PIPQ maintains a two-level hierarchy of supporting data structures. The two levels include the worker level and the leader level. The worker level provides fast, parallel insertions with minimal synchronization because each thread p inserts to its own worker-level heap, H_p. Each H_p is implemented as a conventional min-heap protected by a single global lock. The simplicity of this level's design is justified by the fact that most of the time no other threads need access to H_p, as subsequently discussed. At the leader level, the algorithm maintains a sorted linked list L of keys, which stores the highest priority elements in PIPQ, and is meant to serve delete-min requests.

Elements may be inserted to the worker or leader level and may move between levels in order to maintain the following invariant: *for each thread p, every element in L tagged with p (that is, p inserted the element into the global data structure) has greater or equal priority than the highest priority element in the worker-level heap of p, H_p.* This ensures that when we remove the highest priority element from the leader level, it is, in fact, the highest priority element in the priority queue, allowing us to achieve linearizability without relaxing operation semantics. The items in red in Figure 1 provide an example instantiation of PIPQ relative to a specific thread, denoted thread 1. The highest priority element in H_1 has priority 8. The corresponding nodes inserted by thread 1 in L have a higher priority (smaller value of the key, `k`) than 8, and are ordered in L by `k`.

A thread p invoking insert on PIPQ will insert to the leader level only when the new element is of higher priority than the highest priority element in H_p (or when H_p is empty). Otherwise, the element will be inserted into H_p. The leader level is intentionally kept small (explained below), so we expect most insertions to occur at the worker level.

The delete-min operation leverages the combining technique [15, 9] as used in several prior works [8, 10, 7, 19]. Electing one thread to perform a sequence of delete-min operations, which are inherently sequential, avoids the costly contention of many threads attempting to access and remove the highest priority element in the leader-level linked list. We call this thread the *coordinator*. A thread p that performs a delete-min operation on PIPQ first announces its operation and then attempts to become the *leader* of its Non-Uniform Memory Access (NUMA) node. Leaders of each NUMA node compete to become the coordinator;

thus, there are up to n leaders competing for the coordinator role at any given time, where n is the number of NUMA nodes in the machine. The coordinator is responsible for serving the delete-min operations of threads that execute on its NUMA node. Thus, either p will become the coordinator and complete its own request along with the other requests of its NUMA node, or its request will be completed by some other thread of its NUMA node that won the race to become the coordinator.

Depending on the sequence of insert and delete-min operations, the leader-level linked list L may grow and shrink arbitrarily. However, the size of L is an important factor in the overall performance of PIPQ. In fact, having many elements in L hinders the performance of insertions for two reasons. First, traversals on L involve many elements. Second, a large size of L increases the likelihood that an insertion has to act at the leader level as opposed to at the worker level, since the range of the priority of elements in L increases with its size. For these reasons, we limit the size of L by requiring insertions to move elements from L down to its worker-level heap if the number of elements that the thread has inserted in L exceeds a certain threshold, which we call `CNTR_MAX`.

On the other hand (and less intuitively), having too few elements in L hinders the performance of delete-min operations. To successfully linearize delete-min operations while ensuring the properties that L needs to satisfy, each thread has to have at least two of the elements it has inserted in L; the justification for requiring at least two elements is detailed in Section 5.1. If this requirement is not met, the coordinator must promote an element from the thread's worker-level heap. Having few elements in L means that the coordinator would need to perform such a promotion more frequently. We address this issue by defining a second threshold, `CNTR_MIN` (with a value less than that of `CNTR_MAX`), and use it to introduce a helping mechanism.

The idea behind helping is that a thread can do work that does not directly impact the operation it is performing but can "help" other operations complete their work in a more efficient manner. Since threads performing the delete-min operation must wait on the coordinator, we initiate helping during this waiting period to reduce the work required by the coordinator. Specifically, if the number of elements inserted by some thread p in L is less than `CNTR_MIN`, p will promote an element from H_p up to the leader level instead of waiting for the coordinator to eventually do so.

We maintain an array of atomic counters containing one slot per thread such that each thread p's slot in the counters array stores the number of elements currently in L that were inserted by p to the global structure. We refer to p's counter value as L_count_p. The use of helping and the gap between `CNTR_MIN` and `CNTR_MAX` is meant to achieve the following two goals: (1) it is less costly for a waiting thread p to help by moving elements from H_p to L, compared to the coordinator needing to do so; thus, we reduce the frequency with which the coordinator must do additional work, and (2) by helping only when L_count_p is less than the minimum threshold (which is, in turn, less than the maximum threshold), we reduce the frequency with which an insertion has to move an element down from the leader level to the worker level.

Before discussing the details of PIPQ, it is important to highlight that the leader-level linked list, L, differs from traditional linked lists in that it requires tracking which thread inserts each element in order to move them to the corresponding worker-level min-heap when necessary. To do so, L stores tuples $(\texttt{key,val,tid}_\texttt{p})$, where $\texttt{tid}_\texttt{p}$ represents the unique identifier of the thread, p, that first inserted this key-value pair in the priority queue. L is sorted based on the key and supports the following operations: *a)* L-Insert(key, val, tid_p), which inserts the tuple $(\texttt{key,val,tid}_\texttt{p})$ in L; *b)* L-DeleteMin, which deletes and returns the

element with the smallest (highest priority) key in L; and *c)* L-DeleteMaxP(lead_largest, $\texttt{tid}_\texttt{p}$), which removes the element with the largest (least priority) `key` that was inserted to PIPQ by the thread with identifier $\texttt{tid}_\texttt{p}$ (to be subsequently inserted into H_p).

4 Operational Workflow

Figure 2 contains flow diagrams for the `DeleteMin` and `Insert` APIs of PIPQ. Recall the invariant maintained between levels, such that the priority of each element in L must be greater than or equal to the priority of the highest priority element from their respective worker level heaps. Deciding on which level to insert, as well as the movement of elements between levels, aims to maintain this invariant. Pseudocode for the leader-level APIs is provided in Section 5.1 and their helper (search) functions are provided in Appendix A. The entire pseudocode of PIPQ is provided in the extended version of the paper.

Insertion Flow. We first describe how an insert operation works in PIPQ, taking into consideration the different levels to which a new element might be inserted. The following description corresponds to the diagram on the left in Figure 2. The steps mentioned below refer to the numbered blocks in the diagram.

Let us consider an insert operation performed by thread p, identified by $\texttt{tid}_\texttt{p}$, that inserts key `k` and value `v`. Per Step 1, the thread begins by locking its local, worker-level min-heap, H_p. In Step 2, it checks the minimum (i.e., highest priority) element in H_p's key, $\texttt{k}_\texttt{p}$, and compares it to `k`. This comparison determines whether the insertion will follow the so called *fast path* of the algorithm or one of two slow paths, named the *slower* and *slowest* path. If `k` is greater than or equal to $\texttt{k}_\texttt{p}$ (i.e., has a lesser priority), then the fast path is taken as the insertion is thus able to be performed in a completely parallel manner. Specifically, p inserts the key-value pair, `k` and `v`, into H_p (Step 3), and then unlocks H_p and returns (Step 11).

On the other hand, if `k` is found to be less than the minimum of H_p at Step 2, then it is necessary to compare `k` to the priority of elements in the leader level in order to maintain proper ordering of elements. The comparisons in Steps 4 and 7 determine if the element should be inserted to the leader-level, in which case the thread follows either the slower path (orange, middle dashed box) or the slowest path (red, right-most dashed box), or if the element should be inserted to the worker level via the fast path (following Steps 7 to 3).

Recall that in order to maximize the number of insertions to a thread's local heap without affecting the efficiency of insertions to L, we set a maximum number of elements for L, `CNTR_MAX`. Assuming Step 4 finds that L_count_p is not equal to `CNTR_MAX` (and is thus smaller than it), we follow the slower path. Thus, in Step 5, we insert to L using L-Insert and then perform the atomic fetch-and-add operation on L_count_p (Step 6) to increment it. Finally, we unlock H_p and return in Step 11.

On the contrary, if L_count_p is determined to be equal to `CNTR_MAX` back in Step 4, this indicates that if the new element is inserted to L, then an element must also be moved down to the worker-level. If `k` is the largest element in L tagged with $\texttt{tid}_\texttt{p}$, then it would be wasted work to insert the element into L just to subsequently remove it to be inserted into H_p. To avoid this situation, we maintain a per-thread pointer, $\texttt{lead\_largest}_\texttt{p}$, which points to the node containing the largest key in L tagged with some $\texttt{tid}_\texttt{p}$. Step 7 compares `k` to the priority of the node pointed to by $\texttt{lead\_largest}_\texttt{p}$ to determine if we can insert the element to the worker level. If `k` is in fact larger or equal in priority to the current largest element (Step 7), then p inserts the key-value pair, `k` and `v`, into H_p (Step 3), and then unlocks H_p and returns (Step 11).

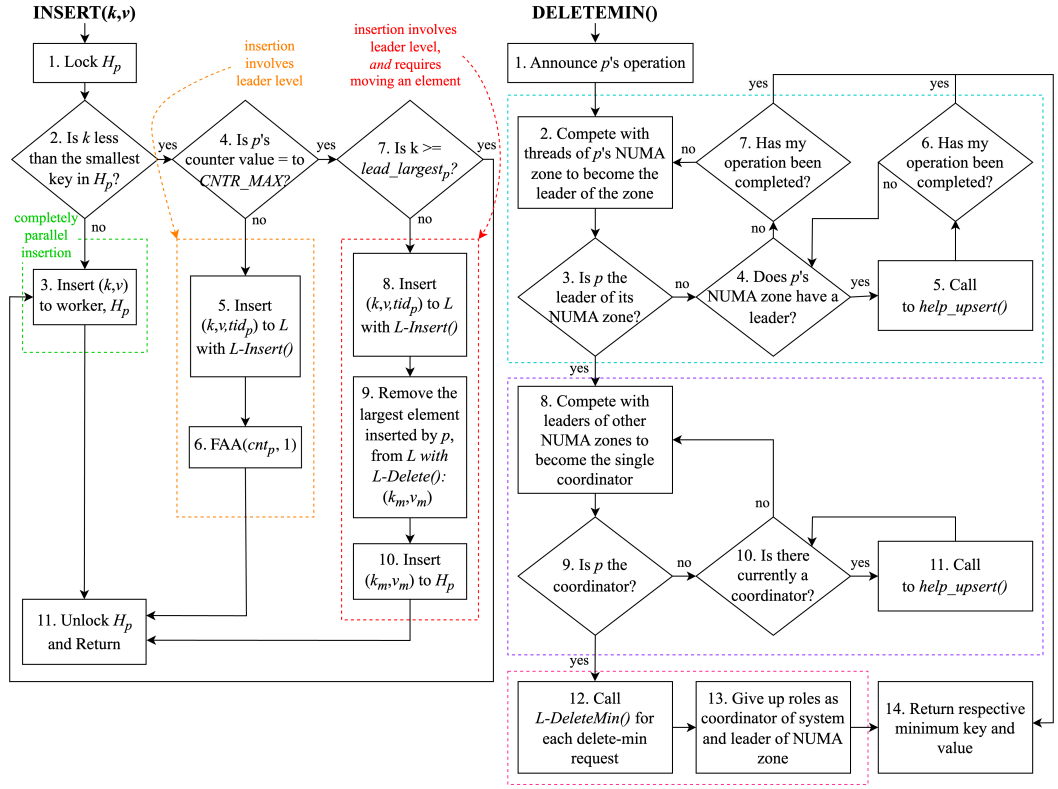


Figure 2 Flow diagrams for the Insert (left) and Delete-Min (right) operations. In each diagram, p represents the thread performing the respective operation.

Otherwise, the new element must be inserted to L, and thus the slowest path is followed. In Step 8, L-INSERT is called to insert tuple ($\texttt{k}$, $\texttt{v}$, $\texttt{tid}_\texttt{p}$) into L. Then in Step 9, L-DELETEMAXP is called to remove the node pointed to by $\texttt{lead\_largest}_\texttt{p}$ from L. During the traversal of L, the algorithm tracks the relevant predecessor of $\texttt{lead\_largest}_\texttt{p}$ (i.e., the first node tagged with $\texttt{tid}_\texttt{p}$ that appears before the node pointed to by $\texttt{lead\_largest}_\texttt{p}$) in order to reset the pointer. Thus, the node pointed to by $\texttt{lead\_largest}_\texttt{p}$ is removed from L, $\texttt{lead\_largest}_\texttt{p}$ is reset, and L-DELETEMAXP returns the relevant key and value ($\texttt{k}_\texttt{m}$, $\texttt{v}_\texttt{m}$). Finally, ($\texttt{k}_\texttt{m}$, $\texttt{v}_\texttt{m}$) is inserted to H_p in Step 10, and in Step 11 H_p is unlocked and the operation completes.

Delete-Min Flow. We now describe the flow of the delete-min operation. Steps mentioned refer to those in the right diagram of Figure 2.

The delete-min operation begins with thread p announcing its operation in Step 1. The blue (top-most) dashed box in the Figure shows the competition between threads of the same NUMA node to become the leader of their node. Per Steps 3 and 4, if p fails to become the leader, it waits until either its operation is completed by the current coordinator (Step 6) or it eventually succeeds in becoming the leader. While waiting in Steps 4-6, the operation calls a helper routine, HELP-UPSERT, which checks the calling thread's counter value, L_count_p, and compares it to $\texttt{CNTR\_MIN}$. If the comparison finds that L_count_p is less than $\texttt{CNTR\_MIN}$, thread p will move the highest priority element from H_p to L, updating its counter value accordingly.

If a thread becomes the leader of its NUMA node (Step 8), then it competes to become the coordinator of the entire system, as reflected in the purple (middle) dashed box. If it does not succeed, it follows a nearly identical waiting routine as previously described, the only difference being that it does not check if its operation has been completed since the coordinator only completes operations carried by threads in its own NUMA node.

Once a thread becomes the coordinator (Step 12), it completes the delete-min operations in a combining manner [15, 9]. This is done by calling L-DeleteMin for each entry in the NUMA-local announce array. After each L-DeleteMin call, it atomically decrements L_count_p. Recall that in order to maintain correctness at the leader level, each thread p must have at least two elements in the linked list associated with its identifier, $\texttt{tid}_\texttt{p}$. Thus, after decrementing, if the new value of the counter is less than two, the coordinator must move an element from the relevant worker-level heap up to the leader-level. To do so, the coordinator first gets the lock on the relevant worker-level heap, H_p (unless the coordinator happens to be the thread associated with identifier $\texttt{tid}_\texttt{p}$). It next removes the highest priority from H_p, and then inserts it (augmented with $\texttt{tid}_\texttt{p}$) into L, and finally unlocks H_p if necessary (i.e., if the lock was acquired). Before proceeding to the next active request, the coordinator informs the originating thread that its operation has been completed via the announce array. Once the coordinator has completed all the active operations, it gives up its role as the coordinator and then as the leader of its NUMA node (Step 13) and finally returns its operation (Step 14) to the application.

5 Internal Data Structures Design

5.1 Leader-Level Linked List

The implementation of the leader-level linked list structure, L, combines characteristics from two state-of-the-art lock-free algorithms, namely Harris' sorted singly-linked list algorithm [14] and the Lindén-Jonsson priority queue algorithm [18], which implements the priority queue using a skip list; we borrow ideas from the level-zero list implementation.

The Harris work introduces the idea of marking nodes for removal before performing physical deletion (i.e., unlinking the node) to implement a lock-free linked list. The actual technique used for marking nodes relies on stealing [2] and manipulating the last bit of the pointer[3] pointing to the next node. The physical deletion can then be performed.

An important difference between the Harris algorithm and Lindén-Jonsson's level-zero algorithm lies in their approach to logically marking nodes for removal before performing physical deletion. In fact, in order for the Lindén-Jonsson algorithm to mark a node as logically removed, the preceding node's next pointer is marked instead of the next pointer of the node itself. The algorithm must mark the previous node for deletion in order to maintain a prefix of logically deleted nodes; having this prefix is an optimization that allows for delaying physical removal, which improves performance.

In our leader-level linked list implementation, we use the same optimization as the Lindén-Jonsson algorithm in the L-DeleteMin operation. However, we cannot use this marking approach for the L-DeleteMaxP operation because L-DeleteMaxP removes elements from the middle of the list (as opposed to the front, as in L-DeleteMin). Instead, the L-DeleteMaxP operation follows the approach of Harris' algorithm and marks the next pointer of the node to be deleted. In fact, if we were to mark the previous node's next

[3] Memory is properly aligned to enable the bit-stealing technique.

pointer for removal, it may cause a concurrent insertion to be falsely inserted. Specifically, say a concurrent insertion inserts its new node, n, directly after some node r. It is then possible that n is inserted before the physical removal of r, and upon the physical removal of r, n would also be removed. This situation is prevented by marking r's next pointer since this would prevent the concurrent insertion after r.

These two different marking approaches present a correctness issue: an L-DeleteMin operation and an L-DeleteMaxP operation cannot simultaneously mark the same node. Note that it is only possible for one L-DeleteMin and one L-DeleteMaxP operation on L to interfere at any single time, since at each moment there is only one active thread performing L-DeleteMin (i.e., the coordinator), and only one thread performing L-DeleteMaxP on a specific element, since a thread p only ever moves an element tagged with $\texttt{tid}_p$. To remedy this issue and guarantee that the L-DeleteMaxP and L-DeleteMin operations never interfere, we maintain at least two elements in L from each thread. For this reason, we require that `CNTR_MIN` (and thus, `CNTR_MAX` as well) must be at least two.

Note that we explored other designs for the leader level and found our current implementation to be the most effective. Despite the linear complexity of linked lists, our use of `CNTR_MIN` and `CNTR_MAX`, which reduce the likelihood that either operation must follow its worst-case path, combined with maintaining a small size of the list, makes the complexity not a practical concern. More details are provided in Appendix C.

Implementation Details

The leader-level linked list has two immutable sentinel nodes pointed to by `head` and `tail`, that contain $-\infty$ and $+\infty$, respectively. The actual elements of the list are stored in nodes between the `head` and `tail` nodes. Each node contains a next pointer and the three fields comprising the tuple `key`, `val`, and $\texttt{tid}_p$.

The leader-level linked list (L) supports the following APIs: L-Insert, L-DeleteMaxP, and L-DeleteMin. Internally, L additionally includes helper functions search, searchDelete, and searchPhysDel. These helper functions are all variations of the search function provided by Harris' linked list [14], and more details are included in Appendix A.2 due to space constraints (see Algorithms 4 and 5). The implementation is lock-free, like that of Harris' linked list [14] and Lindén and Jonsson's priority queue [18].

Helper Functions & Marking. Each of the three helper functions, search, searchDelete, and searchPhysDel, must traverse the leader-level linked list by following pointers which may be marked as either *logically deleted*, or *moved*. We steal two bits from each node's next pointer to support the two marking techniques. We denote the bit associated with logical deletion as the DELMIN bit, and the bit associated with moving an element as the MOVING bit. Recall that a node is considered logically deleted if the next pointer of its predecessor node's DELMIN bit is set, and moved if its own next pointer's MOVED bit is set.

The three search functions use the following methods to safely traverse the nodes and set or unset the two bits. Note that there are variants for each function, expressed between the "{}" braces, and the description of the return values for each function discusses them from left to right, respectively.

- **Is_{ LogDel | Moving }_Ref(ptr*)**: returns True/False depending on whether the LOGDEL bit or MOVING bit, respectively, is set in `ptr`.
- **Get_{ LogDel | Moving }_Ref(ptr*)**: returns modified `ptr` with LOGDEL bit set or MOVING bit set, respectively.
- **Get_{ NotLogDel | Unmarked }_Ref(ptr*)**: returns modified `ptr` *without* the LOGDEL bit set, or neither bit set, respectively.

L-Insert. The L-INSERT operation (see Algorithm 1) begins by calling helper function SEARCH (Algorithm 4 in the Appendix), which locates the two nodes to insert between, namely `l_node` and `r_node`. The search function is very similar to that of Harris'; the differences in our implementation account for the different types of marking used by PIPQ.

Once the search returns to L-INSERT, `new_node` is allocated by calling NEWNODE (Line 4) which sets fields `key`, `val`, $\texttt{tid}_\texttt{p}$, and `next`. Finally, a CAS is performed in Line 5 of L-INSERT to swing `l_node`'s next field to point to `new_node`. Upon a successful CAS, the algorithm returns. Otherwise, the operation is attempted again.

L-DeleteMaxP. Recall that the L-DELETEMAXP function removes the last element in L tagged with the calling thread's identifier, to be inserted to the worker level. L-DELETEMAXP begins by storing a copy of `lead_largest` in pointer `start_lead_largest`, to be later used when removing this node from L.

The algorithm then enters a loop and calls helper method SEARCHDELETE (Algorithm 5) in Line 15, which identifies and returns `l_node` and `r_node`, such that `r_node` is the node to be removed from L to move down to the worker-level, and `l_node` is its direct predecessor. The search function additionally resets the thread's `lead_largest` pointer.

Once the left and right nodes are returned to L-DELETEMAXP, a successful CAS in Line 18 indicates that the node has been logically removed from the list and the algorithm breaks from the loop; else it tries again. Lines 21-22 ensure that the node is physically removed from the list; if the CAS in Line 22 fails, the search function SEARCHPHYSDEL is called to ensure physical removal of `r_node`. SEARCHPHYSDEL varies only slightly from SEARCH, such that instead of searching for a certain key value, it searches for `r_node`. If the algorithm finds the node, then it will detect that it needs to perform physical removal before returning, as the other search methods do; if it is not found, then another thread has physically unlinked it. Finally, the key and value of the element to move down to the worker-level are returned, which is subsequently handled by the thread in the calling function.

L-DeleteMin. L-DELETEMIN removes the highest priority element from L, which is also the highest priority element in the entire data structure. Like the Lindén-Jonsson algorithm, L-DELETEMIN performs batch physical deletions once some predetermined offset of logically deleted nodes has been reached.

The thread performing L-DELETEMIN first traverses past the prefix of logically deleted nodes in lines 27-35. When the thread successfully marks the first element as logically deleted, which is verified by checking `x_next` in Line 35 (i.e., the return value of the fetch-and-or from Line 34), it exits the loop. If the offset has exceeded the maximum offset, the thread then resets the head's next pointer to the node just marked for logical deletion (lines 38-39). Note that, unlike the Lindén-Jonsson algorithm, we do not need to perform a compare-and-swap to unlink the prefix since only one thread is ever performing a delete-min operation at the time. Nonetheless, it is still beneficial for performance to delay physical deletion, as concurrent insertions face fewer cache misses when traversing the list.

5.2 Worker-Level Min-Heap

The min-heap is implemented as a pre-allocated array. If it becomes full, an additional array is allocated and linked to the first array. Each min-heap is guarded by a single lock. There are only ever up to two threads competing for a heap H_p's lock: the thread p, which is local to the heap, and the thread that is the coordinator performing a delete-min operation. In practice, however, we find it very uncommon for a coordinator thread to need access to the worker level thanks to our helping mechanism, as detailed in Section 3.

Algorithm 1 Pseudocode for the interface methods of the leader-level linked list.

```
1: function L-Insert(list_t list, Node *lead_largest,
   key_t key, val_t val, int tid)
2:   repeat
3:     l_node, r_node ← search(list, key)
4:     new_node ←
           NewNode(key, val, tid, r_node)
5:     if CAS(&l_node.next, r_node, new_node)
         then
6:       if not *lead_largest or
           key > *lead_largest.key then
7:         *lead_largest = new_node
8:       return
9:   until True
10:
11: function L-DeleteMaxP(list_t list,
    Node *lead_largest, int tid)
12:   start_lead_largest ← *lead_largest
13:   repeat
14:     l_ptrs ← (lead_largest, start_lead_largest)
15:     l_node, r_node ←
            searchDelete(list, l_ptrs, tid)
16:     r_node_next ← r_node.next
17:     if not is_marked_ref(r_node_next) then
18:       if CAS(&r_node.next, r_node_next,
            get_moving_ref(r_node_next)) then
19:         break
20:   until True
21:   if not CAS(&l_node.next, r_node,
        r_node_next) then
22:     searchPhysDel(list, r_node)
23:   return [r_node.key, r_node.val]
24: function L-DeleteMin(list_t list)
25:   offset ← 0
26:   x ← list.head
27:   repeat
28:     offset ← offset + 1
29:     x_next ← x.next
30:     if get_notlogdel_ref(x_next) = list.tail
          then
31:       return EMPTY
32:     if is_logdel_ref(x_next) then
33:       continue
34:     x_next ← fetch_and_or(&x.next, 1)
35:   until x ← get_notlogdel_ref(x_next) and
        not is_logdel_ref(x_next)
36:   new_head ← x
37:
38:   if offset > MAX_OFFSET then
39:     list.head.next ←
            get_logdel_ref(new_head)
40:   return [x.key, x.val, x.tid]
```

6 Correctness

A sketch of the proof of PIPQ's linearizability is provided in Appendix B due to space constraints. The complete proof and an analysis of PIPQ's single-threaded worst-case performance are included in the extended version of the paper.

7 Evaluation

We conduct our experiments on a machine running Ubuntu 22.04.3 LTS equipped with four Intel Xeon Platinum 8160 processors containing a total of 96 cores split between four NUMA nodes. Our code is written in C++ and compiled with `-std=c++20 -O3 -mcx16`. Our data points represent the average of three trials performed for five seconds each. Threads are pinned to cores using a NUMA node by NUMA node policy. Hyperthreading is disabled.

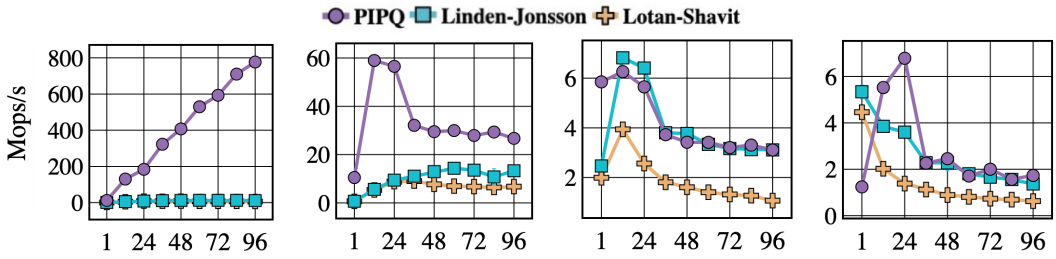
We compare PIPQ with the following two state-of-the-art, strict priority queues: Lotan-Shavit [28] and Lindén-Jonsson [18]. Lotan-Shavit [28] is a priority queue implemented on top of Fraser's skiplist [11]. The delete-min operation is implemented by traversing the lowest-level linked list, and atomically marking the first unmarked node as logically deleted. This data structure is not linearizable but is quiescently consistent [16]. Lindén-Jonsson [18] extends the Lotan-Shavit design by batching the physical deletions once a certain threshold of logically deleted nodes has been met. Lindén-Jonsson outperforms Sundell and Tsigas' linearizable priority queue algorithm [29] and Herlihy and Shavit's [16] lock-free adaptation of the LS priority queue [28].

We test different applications, workloads, and data access patterns: a traditional microbenchmark to evaluate concurrent data structures; a designated thread experiment, where each thread performs either insert *or* delete-min, but not both; and a phased experiment, in which the workload changes in phases. We additionally evaluate the single-source shortest path graph algorithm, which is included only in the extended version of the paper.

7.1 Microbenchmark

Results for the microbenchmark are shown in Figures 3, 4 and 5 where we evaluate throughput (millions of operations per second), latency (microseconds), and which path is followed by PIPQ's insert operations, respectively, as we increase threads. In our microbenchmark, threads perform a series of insert and delete-min operations based on the given mixed workload. The key and value are integers ranging between 1 and 100M and are randomly generated using a uniform distribution. We include four workloads: 100% insert, 95% insert, 50% insert, and 100% delete-min operations. Via experimentation, we find that using values of 10 for `CNTR_MIN` and 100 for `CNTR_MAX` achieves the goal of maintaining a proper size of the leader-level linked list, as discussed in Section 3.

We also performed experiments (not included in the paper) in which we tracked the average number of nodes traversed by the two functions that traverse L past its logically deleted prefix of nodes, namely L-Insert and L-DeleteMaxP, for various values of `CNTR_MIN` and `CNTR_MAX`. The following discussion relates to results captured for a 50% inserts workload. Our results show that modifying the value of `CNTR_MAX` has small effect on the length of traversals, whereas modifying the value of `CNTR_MIN` has a clear correlation. Despite this, however, the overall throughput achieved by the runs hardly varied. Thus, we conclude that the traversal on L does not introduce a performance cost on PIPQ.



(a) 100% insert. **(b)** 95% insert. **(c)** 50% insert. **(d)** 100% delete-min.

Figure 3 Throughput using a microbenchmark. The x-axis represents the number of threads.

Throughput Analysis. Refer to Figure 3 for the results containing throughput achieved by PIPQ and its competitors. Overall, PIPQ performs better or as well as Lindén-Jonsson in all experiments, both of which always perform better than Lotan-Shavit. When insertions dominate the workload, the parallel nature of insertions provided by PIPQ showcases the performance advantages of our approach compared to competitors.

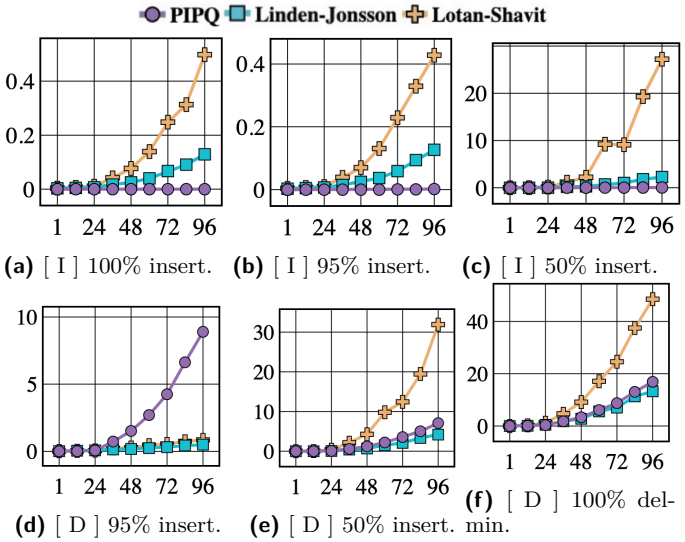
We direct attention to Figure 3a, which shows the 100% insert workload. Since the leader-level linked list remains small, containing up to 100 elements per worker, and there are no threads removing the minimal keys, it becomes highly likely that threads will follow the fast path of the insert operation as more elements are inserted. Thus, insertions become nearly embarrassingly parallel and benefit from cache locality and the speed of the sequential worker min-heap operations, providing significant speedup. At 96 threads, PIPQ performs over 76x better than its closest competitor, Lindén-Jonsson.

Figure 3b shows the results of the 95% insert workload. In this case, we outperform Lindén-Jonsson by almost one order of magnitude at 12 threads (i.e., within a NUMA node) and by 2.4x at 96 threads. To explain the performance drops in the presence of the delete-min operation, we perform an experiment to determine how many operations the coordinator performs on average for the various workloads. We experimentally find that the coordinator

performs, on average, about 24 operations in the case of the 95% insertions workload. Thus, even with a workload of only 5% delete-min operations, nearly every thread in a NUMA node eventually has to wait for the coordinator to complete that thread's operation (or to become the coordinator itself). Because our insert operation is so fast compared to the delete-min operation, potentially many insert requests could be served during the waiting required by the delete-min operation.

That said, however, PIPQ scales significantly better than competitors within a NUMA node, showing the algorithm's ability to increase overall performance for insert-dominant workloads despite such a pessimistic operation (i.e., delete-min) in the mix. The drop in performance after 24 threads is due to the additional synchronization needed to coordinate delete-min operations carried by threads on other NUMA nodes. Nonetheless, PIPQ still outperforms all competitors in the 95% insert case.

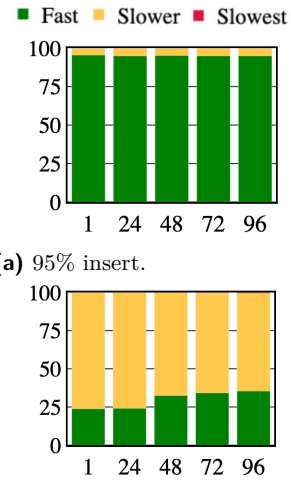
In the 50% insert workload (Figure 3c), PIPQ performs similarly to Lindén-Jonsson, and outperforms Lotan-Shavit. PIPQ also performs similarly to Lindén-Jonsson for the 100% delete-min workload (Figure 3d), though exhibits performance benefits within a NUMA node (i.e., scaling up to 24 threads). Performing so comparably to Lindén-Jonsson for the 100% delete-min workload for 24 threads and beyond shows an interesting correlation. Essentially, the overhead of our multi-level hierarchy, which requires moving elements between levels to ensure strict ordering, is seemingly comparable with the overhead of adjusting Lindén-Jonsson's skip list indexing levels.



(a) [I] 100% insert. **(b)** [I] 95% insert. **(c)** [I] 50% insert.

(d) [D] 95% insert. **(e)** [D] 50% insert. **(f)** [D] 100% delmin.

Figure 4 Latency using microbenchmark (less is better). [I] indicates insert latency, [D] indicates delete-min latency. # of threads in x-axis; microseconds in y-axis.



(a) 95% insert.

(b) 50% insert.

Figure 5 % path is followed by insert operation; # of threads in x-axis.

Latency Analysis. Figure 4 shows the latency of operations for PIPQ and its competitors. The top row of plots shows the average latency of the insert operation for relevant workloads (100% insert, 95% insert, and 50% insert). In each case, PIPQ has the lowest latency compared to competitors by a large margin, revealing the benefit of PIPQ's parallel insertions. For the 100% insertions workload (Figure 4a) PIPQ's insert latency is 265x lower than the latency of Lindén-Jonsson, and in the case of 95% inserts, PIPQ's insert latency is 49x lower than that of Lindén-Jonsson. Finally, in the case of 50% inserts, PIPQ's latency is 40x lower than the latency of Lindén-Jonsson.

The bottom row of the plots in Figure 4 shows the average latency of the delete-min operation. For the 50% insert and 100% delete-min workloads (Figures 4e and 4f, respectively), PIPQ's latency is comparable to that of Lindén-Jonsson, both of which have lower latency than Lotan-Shavit. For the 95% insert workload on the other hand, PIPQ's latency is higher than competitors. Note that PIPQ's delete-min latency at 96 threads in the 95% inserts workload (Figure 4d) is very similar to the latency of delete-min at 96 threads in the 50% workload (Figure 4e), thus revealing that the latency of the operation remains somewhat constant across workloads. This is due to PIPQ's use of combining, and the implicit added cost of Coordinator's operating in a sequential manner.

Path Followed by PIPQ Insert. In order to understand how often the benefit of inserting at the worker-level occurs for various workloads, we perform additional experiments in which we track the path (fast, slower, and slowest) followed by each insert operation. Recall that the fast path is when an insert is to the worker-level. The slow paths involve inserting to the leader-level, and the slowest path requires the additional step of moving an element to the worker-level. Per the results in Figure 5, the slowest path is rarely followed (the color red is hardly visible on the plots); the percentage of the time that it is followed is less than 0.5% for all thread counts in both workloads, significantly less so in many cases.

In the 95% insert workload (Figure 5a), the fast path is followed approximately 95% of the time for all thread counts, and the slower path is followed about 4.9% of the time for all thread counts. In the case of 50% inserts (Figure 5b) the fast path is taken by inserts ranging from 24% of the time with one thread, scaling up to 35% of the time at 96 threads, and the slower path is followed ranging from 76% of the time at one thread, scaling down to 64% of the time at 96 threads. The increased percentage in the slower path being followed for the 50% insert workload compared to the 95% insert workload is simply due to the fact that significantly more elements are being removed in the 50% case, so the probability that an element needs to be inserted to the leader-level given a uniformly generated workload is largely increased compared to that of the 95% case.

7.2 Designated Thread Experiments

Note that designating threads causes calls to Help-Upsert by delete-min operations to be ineffective, since the threads performing delete-min do not ever insert. To address this issue, we move the helping effort to the insert operation, such that after performing its insert, but before it releases its lock on its local H_p, an insert carries out the logic of help-upsert, removing an element from H_p and inserting it into L when necessary. We omit single-threaded performance as it is not a relevant data point for this experiment.

We denote d as the fraction of the total running threads that are designated to perform the delete-min operation. We compared PIPQ with our competitors for values of $d = 1/24$, $1/12$, and $1/2$. Making $d = 1/2$ represents the case where half of all threads perform the delete-min operation, and the other half perform the insert operation. The values of $d = 1/24$ and $1/12$ assign more threads for insertions. The results are in Figure 6. We separated the throughput of insertions and delete-mins into distinct plots to better understand the behavior of each operation.

PIPQ significantly outperforms competitors for insert throughput for all values of d (see Figures 6a, 6b, and 6c). PIPQ achieves the largest speedup for insert throughput in Figure 6a when $d = 1/24$, producing 50x speedup compared to the closest competitor. These experiments show that our insert operation is able to retain performance in the presence

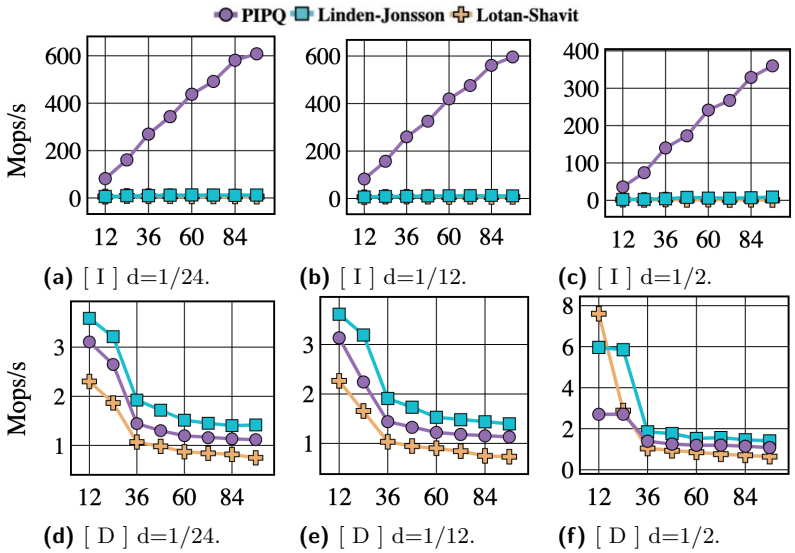


Figure 6 Throughput for the designated thread experiment. "d" denotes the fraction of threads performing delete-min. [I] indicates insert throughput, [D] indicates delete-min throughput. The x-axis represents the number of threads.

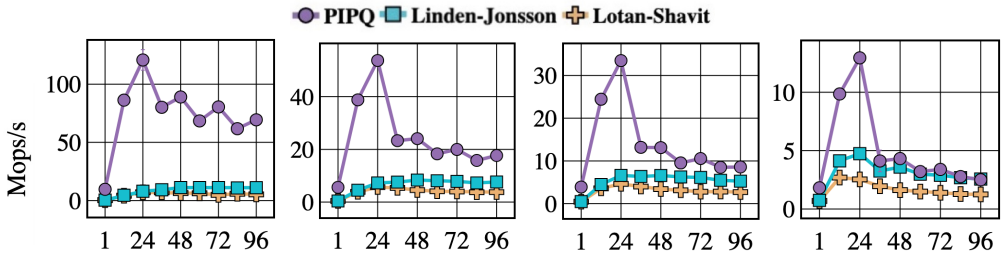
of delete-mins, as long as the threads performing insertions are not stalled due to also performing delete-min operations, as was the case in our microbenchmark experiment. For delete-min, PIPQ achieves higher throughput than Lotan-Shavit and slightly less than that of Lindén-Jonsson. As a result of the high speed of inserts, when we compare the cumulative overall throughput (that is, the summation of the insert and remove-min operations), PIPQ also outperforms its closest competitor by as much as 47x.

Comparing Figures 6f and 6c with Figure 3c exemplifies the effect of dedicating threads for each operation. In Figure 3c, it is expected that half of the threads perform delete-min, and half perform insert at any given time (similar to Figures 6f and 6c). However, PIPQ retains higher performance of insert when threads do not alternate between the two operations.

7.3 Phased Experiment

Some applications of priority queues involve periods of only insertions followed by periods of only delete-mins. We define this workload as "phased" to represent the various periods experienced by the application. During each phase, the total work is split among active threads. The results are shown in Figure 7. The experiments consist of the following phases: in each experiment, 50 million elements are inserted into the priority queue in Phase 1. Once all elements have been inserted, in Phase 2, we remove 1 million (Figure 7a), 5 million (Figure 7b), 10 million (Figure 7c), and 50 million (Figure 7d), i.e. all, elements.

PIPQ outperforms competitors in the three experiments that only remove a fraction of the elements inserted, significantly so within a NUMA node (i.e., up to 24 threads). In each case, Lindén-Jonsson is the closest competitor. As the number of delete-min operations performed in Phase 2 increases, the gap in performance becomes smaller. This aligns with one of our initial motivations that PIPQ's design fits applications where inserts outnumber deletes. In Figure 7a, PIPQ achieves the largest performance gains, as expected, since insertions most significantly dominate the workload in this experiment. Specifically, PIPQ produces up to 15x speedup compared to Lindén-Jonsson at 24 threads and 6.3x speedup compared to



(a) 50->1. **(b)** 50->5. **(c)** 50->10. **(d)** 50->50.

Figure 7 Phased experiments. Workloads are denoted "A->B", meaning Phase 1 is A million inserts and Phase 2 is B million delete-mins. The x-axis represents threads.

Lindén-Jonsson at 96 threads. In Figure 7b, which shows removing 10% of inserted items, 7.4x speedup is produced compared to Lindén-Jonsson at 24 threads, and 2.3x speedup at 96 threads. For the case of removing 20% of the inserted items, shown in Figure 7c, PIPQ gains 5.1x speedup over Lindén-Jonsson at 24 threads and 1.6x speedup at 96 threads.

In Figure 7d, we show when 100% of insertions are then removed. Similar to before, the lesser gap in performance compared to Figures 7a, 7b, and 7c is due to the asymmetry in performance between our two operations; since our insertions are very fast, the amount of time consumed during the second phase is greater compared to the first phase, and so the effect of the delete-min becomes the overwhelmingly dominating factor in performance. That said, however, PIPQ still gains 2.7x speedup over Lindén-Jonsson within a NUMA node.

8 Conclusion

In this paper we presented PIPQ, a strict, linearizable concurrent priority queue. Its design focuses on increasing the performance and parallelism of insert operations as opposed to focusing on the sequential bottleneck of delete-min operations, as done by the majority of prior work. PIPQ achieves this by allowing insert operations to utilize per-thread data structures in the common case. Our evaluation confirms that with PIPQ's design, it is possible to achieve high performance and scalability under various access patterns and runtime configurations.

References

1 Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir 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, pages 11–20, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2688500.2688523.

2 Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. Bit-stealing made legal: Compilation for custom memory representations of algebraic data types. *Proc. ACM Program. Lang.*, 7(ICFP):813–846, 2023. doi:10.1145/3607858.

3 Mohamed Benaïchouche, Van-Dat Cung, Salah Dowaji, Bertrand Le Cun, Thierry Mautor, and Catherine Roucairol. Building a parallel branch and bound library. In Afonso Ferreira and Panos M. Pardalos, editors, *Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques*, volume 1054 of *Lecture Notes in Computer Science*, pages 201–231. Springer, 1996. doi:10.1007/BFB0027123.

4 Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. Cbpq: High performance lock-free priority queue. In *Proceedings of the 22nd International Conference on Euro-Par 2016:*

Parallel Processing - Volume 9833, pages 460–474, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-319-43659-3_34.

5 Manos Chatzakis, Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, and Botao Peng. Odyssey: A journey in the land of distributed data series similarity search. *Proc. VLDB Endow.*, 16(5):1140–1153, January 2023. doi:10.14778/3579075.3579087.

6 Yanhao Chen, Fei Hua, Yuwei Jin, and Eddy Z. Zhang. Bgpq: A heap-based priority queue design for gpus. In *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, ICPP '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3472456.3472463.

7 Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining numa locks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1989493.1989502.

8 Panagiota Fatourou, Nikos Giachoudis, and George Mallis. Highly-efficient persistent fifo queues. In *Structural Information and Communication Complexity: 31st International Colloquium, SIROCCO 2024, Vietri Sul Mare, Italy, May 27–29, 2024, Proceedings*, pages 238–261, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-60603-8_14.

9 Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2145816.2145849.

10 Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios Kosmas. The performance power of software combining in persistence. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 337–352, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503221.3508426.

11 Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, UK, 2004. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193.

12 Olivia Grimes, Ahmed Hassan, Panagiota Fatourou, and Roberto Palmieri. PIPQ Priority Queue. Software (visited on 2025-10-13). URL: https://github.com/sss-lehigh/pipq, doi:10.4230/artifacts.24906.

13 Binglei Guo, Jiong Yu, Dexian Yang, Hongyong Leng, and Bin Liao. Energy-efficient database systems: A systematic survey. *ACM Comput. Surv.*, 55(6):111:1–111:53, 2023. doi:10.1145/3538225.

14 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, Berlin, Heidelberg, 2001. Springer-Verlag. doi:10.1007/3-540-45414-4_21.

15 Danny Hendler, Itai Incze, Nir Shavit, and Moran 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, pages 355–364, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1810479.1810540.

16 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, April 2008.

17 Yuchen Li, Shixuan Sun, Hanhua Xiao, Chang Ye, Shengliang Lu, and Bingsheng He. A survey on concurrent processing of graph analytical queries: Systems and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–20, 2024. doi:10.1109/TKDE.2024.3393936.

18 Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Principles of Distributed Systems. OPODIS 2013*, pages 206–220. Springer International Publishing, December 2013. doi:10.1007/978-3-319-03850-6_15.

19 Xiangzhen Ouyang and Yian Zhu. Core-aware combining: Accelerating critical section execution on heterogeneous multi-core systems via combining synchronization. *J. Parallel Distrib. Comput.*, 162(C):27–43, April 2022. doi:10.1016/j.jpdc.2022.01.001.

20 Botao Peng, Panagiota Fatourou, and Themis Palpanas. Fast data series indexing for in-memory data. *VLDB J.*, 30(6):1041–1067, 2021. doi:10.1007/S00778-021-00677-2.

21 Botao Peng, Panagiota Fatourou, and Themis Palpanas. Paris+: Data series indexing on multi-core architectures. *IEEE Trans. Knowl. Data Eng.*, 33(5):2151–2164, 2021. doi:10.1109/TKDE.2020.2975180.

22 Botao Peng, Panagiota Fatourou, and Themis Palpanas. SING: sequence indexing using gpus. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 1883–1888. IEEE, 2021. doi:10.1109/ICDE51399.2021.00171.

23 Stefano Petrangeli, Jeroen Van Der Hooft, Tim Wauters, and Filip De Turck. Quality of experience-centric management of adaptive video streaming services: Status and challenges. *ACM Trans. Multimedia Comput. Commun. Appl.*, 14(2s), May 2018. doi:10.1145/3165266.

24 Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. Multi-queues can be state-of-the-art priority schedulers. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 353–367, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503221.3508432.

25 William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990. doi:10.1145/78973.78977.

26 Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 80–82, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2755573.2755616.

27 Konstantinos Sagonas and Kjell Winblad. The contention avoiding concurrent priority queue. In Chen Ding, John Criswell, and Peng Wu, editors, *Languages and Compilers for Parallel Computing*, pages 314–330, Cham, 2017. Springer International Publishing.

28 N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 263–268, 2000. doi:10.1109/IPDPS.2000.845994.

29 H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 11 pp.–, 2003. doi:10.1109/IPDPS.2003.1213189.

30 Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. *SIGPLAN Not.*, 50(8):277–278, January 2015. doi:10.1145/2858788.2688547.

31 Deli Zhang and Damian Dechev. A lock-free priority queue design based on multi-dimensional linked lists. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):613–626, 2016. doi:10.1109/TPDS.2015.2419651.

32 Guozheng Zhang, Gilead Posluns, and Mark C. Jeffrey. Multi bucket queues: Efficient concurrent priority scheduling. In Kunal Agrawal and Erez Petrank, editors, *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2024, Nantes, France, June 17-21, 2024*, pages 113–124. ACM, 2024. doi:10.1145/3626183.3659962.

33 Tingzhe Zhou, Maged Michael, and Michael Spear. A practical, scalable, relaxed priority queue. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3337821.3337911.

A Algorithm details of PIPQ

A.1 Structures and Definitions

Algorithms 2 and 3 contain definitions, variables, and structures regarding the worker-level and PIPQ as a whole. Note that not all details are provided, but what is present should be sufficient to properly understand the provided pseudocode.

Algorithm 2 Worker-level min-heap and leader-level linked list definitions.

```
1: struct Heap
2:     list : pointer to HeapNode
3:     size : integer
4:     lock : pointer to integer

5: struct HeapNode
6:     key : integer                                   ▷ the priority of the element
7:     val : val_t

8: struct LeaderListNode
9:     key : integer                                   ▷ the priority of the element
10:    val : val_t
11:    tid : integer                                   ▷ the thread identifier
12:    next : pointer to LeaderListNode

13: struct LeaderList
14:    head : pointer to LeaderListNode (sentinel)
15:    tail : pointer to LeaderListNode (sentinel)
16:    max_offset : int
```

Algorithm 3 PIPQ Implementation.

```
1: struct PIPQ
2: ▷ The leader-level linked list, and array of worker-level min-heaps ◁
3:    worker_heaps[THREADS] : pointer to Heap
4:    leader_list : LeaderList
5:
6: ▷ Locks protecting the Leaders (one per NUMA node) and the Coordinator ◁
7:    compete_coord_locks[NUM_NUMA_NODES] : pointer to integer
8:    coord_lock : pointer to integer
9:
10:   announce[THREADS] : AnnounceStruct          ▷ Announcement for DeleteMin combining
11:   lead_largest_ptrs[THREADS] : pointer to LeaderListNode   ▷ Least priority element in L
12:   leader_counters[THREADS] : pointer to integer            ▷ Number of elements in L
13:
14:   HLS, THREADS, CNTR_MIN, CNTR_MAX, MAX_OFFSET : constant integer
15:
16:   PIPQ(heap_size, nthreads, cntr_min, cntr_max, max_offset) :
17:     HLS(heap_size)
18:     THREADS(nthreads)
19:     CNTR_MIN(cntr_min)
20:     CNTR_MAX(cntr_max)
21:     MAX_OFFSET(max_offset)
```

A.2 Adapted Harris Search Functions for L

Recall that SEARCH (Algorithm 4) is used to support the L-INSERT function of the leader-level linked list, returning the two nodes to insert between (`l_node` and `r_node`). Text marked in red in Algorithm 4 denote modifications made to the original Harris search algorithm.

In lines 8-17, SEARCH must first pass by the prefix of logically deleted nodes, and then when finding `l_node` and `r_node`, account for the fact that a concurrent operation may have marked a node between the left and right nodes as moving, but not yet performed physical deletion (see the check in Line 21). If this is the case, a compare-and-swap operation in Line 27 of SEARCH is performed to carry out the physical deletion. Finally, `l_node` and `r_node` are returned to L-INSERT.

Now direct attention to Algorithm 5, SEARCHDELETE. Recall that SEARCHDELETE supports the L-DELETEMAXP function of the leader-level linked list, which removes the least priority element tagged with some given $\texttt{tid}_\texttt{p}$, to be moved down to the worker level. The function begins by traversing L until it encounters `start_lead_largest` (i.e., the last node in L tagged with the relevant thread identifier). Note that in L-DELETEMAXP we must store pointer `start_lead_largest` (which initially points to the same node as `lead_largest`), in case `lead_largest` is reset in Line 21, and then the subsequent CAS in Line 28 fails and the function must be re-run.

Algorithm 4 Adapted Harris search, used to support the L-INSERT method (see Algorithm 1) of the leader-level linked list.

```
1: function SEARCH(list_t list, int key)
2:     repeat
3:         ▷ 1. Find l_node and r_node ◁
4:         l_node, r_node
5:         prev_log_del ← False
6:         x ← list.head
7:         x_next ← x.next
8:         repeat
9:             if not IS_MOVING_REF(x_next) then
10:                l_node ← x
11:                l_node_next ← GET_NOTLOGDEL_REF(x_next)
12:            x ← GET_UNMARKED_REF(x_next)
13:            if x = list.tail then
14:                break
15:            prev_log_del = IS_LOGDEL_REF(x_next)
16:            x_next ← x.next
17:        until x.key ≥ key and not IS_MOVING_REF(x_next) and not prev_log_del
18:        r_node ← x
19:
20:        ▷ 2. Check if l_node and r_node are adjacent ◁
21:        if l_node_next = r_node then
22:            if IS_LOGDEL_REF(l_node.next) or (r_node.next and IS_MOVING_REF(r_node.next)) then
23:                continue
24:            return [l_node, r_node]
25:
26:        ▷ 3. Remove one or more "moving" nodes ◁
27:        if CAS(&l_node.next, l_node_next, r_node) then
28:            if IS_LOGDEL_REF(l_node.next) or (r_node.next and IS_MOVING_REF(r_node.next)) then
29:                continue
30:            return [l_node, r_node]
31:    until True
```

During traversal, SEARCHDELETE tracks the nearest predecessor of the node pointed to by `start_lead_largest` that is also tagged with $\texttt{tid}_p$ in variable `new_lead_largest`, in order to reset `lead_largest`. Once the traversal is complete (i.e., `start_lead_largest` has been found), `lead_largest` is reset to `new_lead_largest` (Line 21 of SEARCHDELETE). Note that access to `lead_largest` is protected by the underlying lock acquired on the associated thread's heap. As in SEARCH, lines 24-29 of the algorithm ensures that there are not any marked nodes between `l_node` and `r_node` (removing any if found), before finally returning the two pointers.

The final search algorithm is SEARCHPHYSDEL. The algorithm differs only slightly from SEARCH (Algorithm 4), such that it searches for a specific node instead of the location to insert a specific key. Its only goal is to ensure physical deletion of `search_node`, and thus does not return node pointers. The pseudocode is included in the extended version of the paper.

Also included in the extended version of the paper is pseudocode for the following algorithms: WORKER-INSERT, WORKER-DELETE-MIN, HELP-UPSERT, and the insert and delete-min APIs of PIPQ.

B Correctness

We assume a typical asynchronous system with n threads which communicate by accessing shared variables. To prove the correctness of PIPQ, we break it down to its two-level components: the worker level and the leader level. At the worker level, each thread p has its own min-heap H_p protected by a single global lock l_p. Lock l_p also protects access to $lead\_largest_p$. At the leader level, PIPQ maintains a linked list L of keys. Consider an execution α of PIPQ. All our claims below are in reference to this execution.

Algorithm 5 Adapted Harris search, used to support the L-DeleteMaxP method (see Algorithm 1) of the leader-level linked list.

```
1: function searchDelete(list_t list, node_t start_node,
   pair_t lead_largests = (lead_largest, start_lead_largest), int tid)
2:     x, l_node, r_node ← start_node
3:     x_next ← x.next
4:     repeat
5:         ▷ 1. Find l_node and r_node ◁
6:         while True do
7:             if not is_moving_ref(x_next) then
8:                 cur_l_node ← x
9:                 cur_l_node_next ←
                       get_notlogdel_ref(x_next)
10:            x ← get_unmarked_ref(x_next)
11:            if x = list.tail then
12:                break
13:            x_next ← x.next
14:            if not is_moving_ref(x_next) and
                   x.tid = tid then
15:                new_lead_largest ← r_node
16:                l_node ← cur_l_node
17:                l_node_next ← cur_l_node_next
18:                r_node ← x
19:                if x = start_lead_largest then
20:                    break
21:        *lead_largest ← new_lead_largest
22:
23:        ▷ 2. Check if l_node and r_node are adjacent ◁
24:        if l_node_next = r_node then
25:            return [l_node, r_node]
26:
27:        ▷ 3. Remove one or more "moving" nodes ◁
28:        if CAS(&l_node.next, l_node_next, r_node) then
29:            return [l_node, r_node]
30:        x ← list.head
31:        x_next ← x.next
32:        r_node ← new_lead_largest
33:    until 1
```

Recall that L contains two sentinel nodes, one storing the key $-\infty$ pointed to by pointer *head*, and another storing the key $+\infty$ pointed to by the pointer *tail*. The *elements* (or *nodes*) of L at some point t are the keys (nodes) that are traversed if we start from the node pointed to by *head* at t and we move forward following the value (at t) of the next pointer of each node. We call p-element an element of L which is tagged with p (i.e., the *tid* field of the element's struct has the value p). Initially, L contains just the two sentinel nodes and it is sorted.

The complete proof of PIPQ's linearizability is included in the extended version of the paper due to space limitations. To show the conceptual flow of the proof, in the following we list the definitions, invariants, lemmas, and corollaries formally discussed in the extended version of the paper.

▶ **Lemma 1.** *For each thread p,*

1. *H_p is a linearizable priority queue; each worker operation that is applied on H_p is atomic (as it is applied while the thread that executes it holds the lock l_p);*
2. *All accesses to $lead\_largest_p$ are performed atomically (while lock l_p is held). Whenever $lead\_largest_p$ is read, it points to the p-element with the largest key in L among all p-elements.*
3. *While p executes an* L-Insert *to insert a p-element in L, it holds the lock l_p; moreover, the inserted element is of a greater or equal priority than that of the highest priority element in p's worker-level heap, H_p, at the time that p acquired l_p;*
4. *While p inserts a p-element into L it holds the lock l_p;*
5. *Every key that appears as an argument of some* PIPQ-Insert *operation invoked by p (in α) is inserted either in H_p or in L (and not in both).*

▶ **Lemma 2.** *For each thread p, whenever a* L-DeleteMin *is invoked, if H_p is not empty, then the value of L_count_p is greater than or equal to 2 indicating that there are at least two active p-elements in L.*

▶ **Lemma 3 ([14], Sections 4.1 and 5.1).** *Consider any instance I of* Search *with key k. Let l_node and r_node be the pointers returned by I; call* left node *and* right node *the nodes pointed to by these pointers, respectively. Then, there is some point t during the invocation and the response of I, at which all the following conditions hold:*

1. *the left node and the right node are nodes of L at t,*
2. *the key of the left node must be less than k (unless k is the highest priority in L, in which case the left node may be a logically deleted node whose key is greater than or equal to k) and the key of the right node must be greater than or equal to k,*
3. *neither node is marked as moved and the right node is not logically deleted, and*
4. *the right node must be the immediate successor of the left node in L.*

▶ **Invariant 4.** *L is a list whose last element is the sentinel node pointed to by tail.*

▶ **Lemma 5.** *An instance of* L-DeleteMin *and an instance of* L-DeleteMaxP *that are concurrent never attempt to delete the same node in L.*

▶ **Corollary 6.** *A node in L will never have both its DELMIN and its MOVING bit marked.*

▶ **Invariant 7.** *The following claims hold:*

1. *The logically deleted nodes in L form a prefix of the list;*
2. *The suffix of L that is comprised of nodes that are not logically deleted is sorted;*

▶ **Corollary 8.** *The element with the highest priority in L is its leftomost (first) active node.*

▶ **Lemma 9.** *The linearization point of each* L-Insert, L-DeleteMin, *or* L-DeleteMaxP *is within the execution interval of the operation.*

▶ **Lemma 10.** *Denote by ℓ_L the linearization order we derive by the linearization points of the* L-Insert, L-DeleteMin, *and* L-DeleteMaxP *operations (of α). The response of a* L-DeleteMin *or a* L-DeleteMaxP *operation is consistent with respect to ℓ_L.*

▶ **Corollary 11.** *The implementation of L is linearizable.*

▶ **Lemma 12.** *The linearization point of each* PIPQ-Insert *and* PIPQ-DeleteMin *operation op of PIPQ is within the execution interval of op.*

▶ **Definition 13.** *Denote by ℓ_H the linearization of the worker operations (of α). Define $\ell_H(p,t)$ (for a specific thread p), as the subsequence of ℓ_H containing those worker operations that have been linearized by a fixed time t. For each p, the abstract state of H_p at t is the set of elements contained in the priority queue that results when the operations in $\ell_H(p,t)$ are applied* sequentially *on an initially empty priority queue.*

▶ **Definition 14.** *Denote by $\ell_L(t)$ the prefix of ℓ_L containing those operations on L that have been linearized by a fixed time t. The abstract state of L at t is the set of elements contained in the singly-linked list that results when the operations in $\ell_L(t)$ are applied* sequentially *on an initially empty linked list.*

▶ **Invariant 15.** *For each process p, the keys of the p-elements contained in the abstract state of L are the smallest among all the p-elements in the union of the sets that comprise the abstract state of H_p and the abstract state of L.*

▶ **Definition 16.** *At each point t, the abstract state of PIPQ is defined by the (multi-set) union of the abstract states of H_1, H_2, ..., H_n and the abstract state of L.*

▶ **Lemma 17.** *Denote by ℓ the linearization order we derive by the linearization points of the* PIPQ-Insert *and* PIPQ-DeleteMin *operations (of α). The responses of* PIPQ-DeleteMin *operations in PIPQ are consistent with respect to ℓ.*

▶ **Theorem 18.** *PIPQ is a linearizable implementation of a priority queue.*

C Complexity Analysis and Discussion

Given that PIPQ is blocking, and the leader-level linked list is lock-free (not wait-free), it is impossible to give a worst-case bound for either. However, for completeness, it is worth discussing the single-threaded worst-case bound, which is $O(lg(m) + k)$ for both insert and delete-min, where m is the size of the relevant worker-level heap, and k is the size of the leader-level linked list. This is directly inherited from the worst-case performance of its components.

We believe the above worst-case analysis is not a practical issue, due to several reasons. First, for delete-min, this worst case will only occur when the coordinator must upsert an element from a worker heap (i.e., when it removes an element and finds that its relevant counter value is less than 2). In all other cases, the coordinator removes the highest-priority element, which is the first non-logically-deleted element in the list. Also, our helping mechanism offloads most of this "upsert" mechanism to the pending delete-min operations that are waiting for the coordinator, which further reduces the probability that the coordinator will ever need to "upsert" elements. For insert, the worst case represents the scenario in which an element is inserted at the leader level, and an element is subsequently moved down to the worker level. Similar to delete-min, as shown in Figure 5, this scenario (which we call the "slowest path") rarely occurs in practice.

Even when this worst case scenario is unavoidable, our design mitigates its effects by configuring `CNTR_MIN` and `CNTR_MAX`. In particular, reducing `CNTR_MAX` makes the size of the linked list as small as the number of threads (which is practically constant and comparable to the cost of any alternative with logarithmic-time performance). Also, by increasing `CNTR_MIN` we increase the chance that helper threads upsert elements ahead of the coordinator, which reduces the likelihood of this worst-case scenario. In fact, some of our experimental trials that we did not include in the paper aimed at assessing the effect of replacing the leader-level linked list with other data structures, and we observed no performance gains in any of them.