A Lock-free Priority Queue Design Based on Multi-dimensional Linked Lists

Abstract

The throughput of concurrent priority queues is pivotal to multiprocessor applications such as discrete event simulation, best-first search and task scheduling. Existing lock-free priority queues are mostly based on skiplists, which probabilistically create shortcuts in an ordered list for fast insertion of elements. The use of skiplists eliminates the need of global rebalancing in balanced search trees and ensures logarithmic sequential search time on average, but the worst-case performance is linear with respect to the input size. In this paper, we propose a quiescently consistent lock-free priority queue based on a multi-dimensional list that guarantees worst-case search time of $\mathcal{O}(\log N)$ for keys in the range of $[0, N)$. The novel multi-dimensional list (MDList) is composed of nodes that contain multiple links to child nodes arranged by their dimensionality. The insertion operation works by first generating a one-to-one mapping from the scalar keys to a high-dimensional vectors space, then uniquely locating the target position by using the vector as coordinates. The ordering property of the MDList structure is readily maintained during insertion without rebalancing nor randomization. In our experimental evaluation using a micro-benchmark, our priority queue outperforms the state of the art approaches by an average of 50%.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Algorithms

General Terms Algorithms, Performance

Keywords Concurrent Data Structure, Priority Queue, Lock-free, Multi-dimensional List, Skiplist

1. Introduction

Scalable non-blocking priority queues are pivotal to the performance of parallel applications on current multi-core and future many-core systems. Attempts on parallelizing search algorithms, such as best-first search [1], do not achieve the desired performance gains due to the lack of scalable concurrent priority queues. Priority task scheduling [19] and discrete event simulation applications [12] also demand high-throughput priority queues to efficiently distribute workload. A priority queues is an abstract data structure that stores a set of key-value pairs where the keys are totally ordered and interpreted as priorities. A priority queue is defined only by its semantics that specify two canonical operations: INSERT, which adds a key-value pair, and DELETEMIN, which returns the value associated with the key of highest priority and removes the pair from the queue. In sequential execution scenarios, priority queues can be implemented on top of balanced search trees or array-based binary heaps [2]. The latter is more efficient in practice because its compact memory footprint exploits spatial locality that optimizes cache utilization. For concurrent accesses by a large number of threads balanced search trees suffer from sequential bottlenecks due to required global rebalancing. Array-based heaps suffer from heavy memory contention in the heapify operation when a newly inserted key ascends to its target location.

In recent research studies [4, 8, 12, 18], concurrent priority queue algorithms based on skiplists are gaining momentum. A skiplist [14] is a linked list that provides a probabilistic alternative to search trees with logarithmic sequential search time on average. It eliminates the need of rebalancing by using several linked lists, organized into multiple levels, where each list skips a few elements. Links in the upper levels are created with exponentially smaller probability. Due to the nature of randomization, skiplists still exhibit less than ideal linear worst-case search time. Skiplist-based concurrent priority queues have a distributed memory structure that allows concurrent accesses to different parts of the data structure efficiently with low contention. However, INSERT operations on skiplists involve updating shortcut links in distant nodes, which incurs unnecessary data dependencies among concurrent operations and limits the overall throughput. Another bottleneck faced by concurrent priority queues is the inherent sequential semantics required by the DELETEMIN operation [3]. Threads competing to remove the minimal element from the queue squander most of their effort trying to decide which one gets the minimal node. The best existing approach employs logical deletion and batch physical deletion to alleviate the contention on head nodes [12], but the parallelism achieved by this approach is still limited because threads performing deletion have to traverse all logically deleted nodes and are forced to wait for the slow insertions. Semantics relaxation [6] opens up an opportunity to trade strict correctness guarantees, such as linearizability [10], for better overall throughput.

In this paper, we present a quiescently consistent lock-free priority queue design based on a multi-dimensional list (MDList). The proposed multi-dimensional list stores ordered key-value pairs and guarantees worst-case sequential search time of $\mathcal{O}(\log N)$ for keys in the range of $[0, N)$. It is composed of nodes that contain multiple links to the child nodes arranged by their dimensionality, and provides convenient concurrent accesses to different parts of the data structure. The insertion works by first injectively mapping a scalar key into a high dimensional vector space, then uniquely locating the target position using the vector as coordinates. As a result, the ordering property of the data structure is readily maintained during insertion without rebalancing nor randomization. The proposed priority queue has the following algorithmic characteristics that aim to further improve the throughput over existing approaches by exploiting a greater level of parallelism and reducing contention among concurrent operations.

- Each insertion modifies at most two consecutive nodes, which allows concurrent insertions to be executed with minimal interference
- A *deletion stack* is used to provide hints about the next minimal nodes so that DELETEMIN operations do not need to traverse logically deleted nodes
- Insertions proceed optimistically without blocking overlapping deletions; they synchronize with DELETEMIN operations only when necessary by rewinding the deletion stack

In our experimental evaluation, we compare our algorithm with Intel TBB's concurrent priority queue and the best available skiplist-based priority queues using a micro-benchmark on two hardware platforms. The result shows that on average our algorithm outperforms the alternative approaches by 50%. We also show that the dimensionality of an MDList-based priority queue can be tuned to fit different application scenarios: a high-dimensional priority queue behaves more like a tree and speeds up insertions; a low-dimensional priority queue behaves more like a linked list and speeds up deletions.

The rest of the paper is organized as follows. In Section 2, we review and compare our approach with a number of existing concurrent priority queue algorithms. In Section 3, we define the multi-dimensional list data structure and introduce the sequential search algorithm. We present the concurrent INSERT and DELETEMIN operations for our MDList-based lock-free priority queue in Section 4. We reason about its correctness and progress properties in Section 5. The performance evaluation and result analysis is given in Section 6. We conclude the paper in Section 7.

2. Related Work

As a fundamental data structure, concurrent priority queues have been extensively studied in the literature. Early concurrent priority queue algorithms are mostly adaptations of the sequential heap data structure. Hunt et al. [11] present a fine-grained locking approach that is built on a number of earlier heap-based algorithms. Herlihy [7] uses an array-based binary heap as an example for the universal construction of lock-free data structures. It requires that each thread only makes modification to a local copy of the heap and updates the pointer to the heap using atomic operations. This serves as a fault tolerance scheme to avoid deadlocks rather than exploit fine-grained parallelism. Both of these approaches, like all heap-based algorithms, suffer from sequential bottlenecks and increased contention due to the compact memory layout of the data structure. We omit detailed discussion on more heap-based approaches as empirical evidence collected on modern multi-core platforms shows that they are outperformed by recent skiplist-based structures [16, 18].

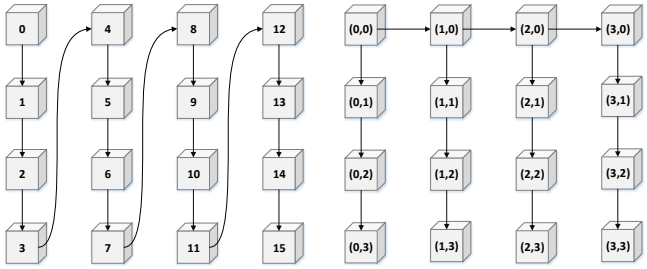
Pugh [13] designs a concurrent skiplist with per-pointer locks, where an update to a pointer must be protected by a lock. While both skiplist and MDList use multiple pointers in one node to link adjacent nodes, they are used in different ways. In a skiplist, upper level links are redundant pointers (i.e. a node can be reached by more than one path from the head), the only purpose of which is to speed up searches. In contrast, an MDList is a tree that has no redundant pointers, where links to multiple dimension point to different partitions of the data structure. Shavit [17] discovers that the highly decentralized skiplist is suitable for shared-memory systems and presents a bounded priority queue based on Pugh's algorithm using fixed bins. However, this approach only supports a small set of fixed priorities whereas our approach supports an arbitrary range of keys. Shavit and Lotan [16] also propose the first unbounded concurrent priority queue based on a skiplist. Their locking approach employs logical deletion (by marking the target) that kept a specialized pointer to the current minimal item, and each DELETEMIN operation has to traverse the lowest level list until it finds an unmarked item. A lock-free adaptation of this algorithm was later presented by Herlihy and Shavit [9]. Sundell and Tsigas [18] present the first linearizable lock-free priority queue based on a skiplist. They guarantee linearizability by forcing threads to help physically remove a node before moving past it. Herlihy et al. [8] propose an optimistic concurrent skiplist that simplifies previous work and allows for easy proof of correctness while maintaining comparable performance. Recently, Linden and Jonsson [12] propose a skiplist-based lock-free priority queue that addresses the sequential bottleneck of the DELETEMIN operation. Their design uses a logical deletion scheme similar to Shavit and Lotan's approach described above, but provides significant performance improvement by performing physical deletions in batch. The drawback was that the logically deleted nodes have to always form a prefix of the lowest level list, and physical deletions cannot pass ongoing insertions.

3. Multi-dimensional List

The core idea of a multi-dimensional list is to partition a linked list into shorter lists and rearrange them in a multi-dimensional space to facilitate search. Just like a point in a D-dimensional space, a node in a D-dimensional list can be located by a D-dimensional coordinate vector. The search operation examines one coordinate at a time and locates correspondent partitions by traversing nodes that belong to each dimension. The search time is bounded by the dimensionality of the data structure and logarithmic search time is achieved by choosing D to be a logarithm of the key range.

3.1 From Linked List to 2DList

An ordered linked list keeps its nodes sorted linearly as if they were attached to a one-dimensional line. To find an existing node or insert a new node, one starts from the head and examines each node consecutively. Given a list of nodes, the search operation takes linear time to complete on average. The linked list illustrated in Fig. 1a has 16 nodes partitioned into 4 columns. This arrangement reveals a property concealed by the typical one-dimensional representation: the nodes are sorted along the columns as well as across the rows, and each column contains a unique range of nodes that have greater keys than the nodes in the previous column. Because a column is substantially shorter than the entire list, we can locate a node much faster if we firstly determine which column the node belongs to and then search for it within the column. However, a one-dimensional linked list lacks two essential properties required by the above search scheme: 1) short-cut links among columns which allow the search operation to switch from one column to another; 2) a function that maps keys into key ranges covered by each column.



(a) Linked List (b) 2D List

Figure 1: From linked list to 2DList

We show an example construction of a 2DList that satisfies these requirements in Fig. 1b. We replace the pointers linking the bottom nodes to the top nodes in Fig. 1a by links among the top nodes so that the top elements of each column form a one-dimensional linked list. This essentially converts the linked list into a tree, but we find it more convenient to visualize it as a multi-dimensional list when discussing the algorithms in the following sections. We also compute a 2-dimensional vector $\mathbf{k} = (k_0, k_1)$ based on the integer key

of each node, which serves as the node's unique coordinate in the 2-dimensional space. The mapping from a key to its coordinate is one-to-one, and we will discuss this in more detail in Section 3.4. The nodes are arranged such that the top nodes of each column are ordered by k_0, while every node in one column shares the same k_0 and is ordered by k_1. We call the top nodes *dimension 0 nodes* (because they are ordered by the 0th element in the coordinate vector), and the nodes in each column *dimension 1 nodes*. To search for a node, for example node $(2, 3)$, we begin by traversing the dimension 0 nodes as if we were traversing a one-dimension linked list. The only difference is that instead of examining the actual key we examine the k_0 of each node and compare it against the target's k_0. We stop at node $(2, 0)$, and continue to traverse the third column, which consists of dimension 1 nodes. We look for the node with $k_1 = 3$ and eventually locate node $(2, 3)$. The average and worst-case search time are halved comparing to the one-dimensional linked list, and we can further improve it by extending the 2DList into higher dimensions.

3.2 Definition

We generalize the construction of the 2-dimensional list to a list of arbitrary dimensions and give the following definition of our multi-dimensional list.

Definition 1. *A D-dimensional list is a rooted tree in which each node is implicitly assigned a dimension of $d \in [0, D)$. The root node's dimension is 0. A node of dimension d has no more than $D - d$ children, where the mth child is assigned a dimension of $d' = d + m - 1$.*

In an ordered multi-dimensional list, we associate every node with a coordinate vector $\mathbf{k}$, and sort the dimension d nodes according to k_d as depicted in Fig. 1b. The following requirement prescribes the exact arrangement of nodes according to their coordinates.

Definition 2. *Given a non-root node of dimension d with coordinate $\mathbf{k} = (k_0, ..., k_{D-1})$ and its parent with coordinate $\mathbf{k}' = (k'_0, ..., k'_{D-1})$ in an ordered D-dimensional list: $k_i = k'_i,\ \forall\ i \in [0, d) \wedge k_d > k'_d$.*

A fundamental property is that a high-dimensional list can be decomposed into a number of low-dimensional lists: every dimension 0 node in a D-dimensional list can be seen as the root node of a $(D-1)$-dimensional list. For example, the 2DList in Fig. 1b consists of four 1DLists. This is analogous to the process of slicing a cube into planes and a plane into lines. If we repeat the decomposition process recursively, we obtain multi-dimensional lists with fewer and fewer dimensions and eventually we arrive at a single node. The design of the search algorithm, described in more details in Section 3.5, is based on this decomposability property.

3.3 Structures

Following Definition 1, it is straightforward to define the structures of a sequential MDList. In Algorithm 1, we define the dimension of an MDList as a constant integer D and the range of the keys as N. The class of MDList itself also contains a pointer to the head (root) node. The functions presented in the following sections are member functions of the MDList class unless otherwise specified, so they have access to the class fields by default. A node in MDList contains a key-value pair, an array $k[D]$ of integers as the coordinate vector, and an array of child pointers in which the dth pointer links to a dimension d child node.[1] We do not need to store the dimension of a node in the node, it is implicitly deduced by the search algorithm based on the index of the node's pointer in its parent's child array.

[1] Since nodes of high dimensions have less children than nodes of low dimensions, for a d dimension node it is possible to allocate a child array that fits only d pointers to reduce memory consumption. However, for simplicity, we choose to allocate child arrays of size D where D is the dimension of the MDList for all nodes. For a dimension d node, only the indices d through $D - 1$ of its child array are valid while the rest are unused.

Algorithm 1 MDList Structures

```
1: class MDList
2:     const int D
3:     const int N
4:     Node* head
5: struct Node
6:     int key, k[D]
7:     void* val
8:     Node* child[D]
```

3.4 Vector Coordinates

There are two requirements for the function that maps a scalar key into a high dimensional vector: 1) it is injective or one-to-one, i.e. distinctness among the keys are preserved; 2) it is monotonic, i.e. the original order among the scalar keys are preserved in the vector space (vector ordering are decided by Definition 2). Additionally, it would be beneficial if the vector coordinates are uniformly distributed in the vector space so that each dimension of the MDList holds comparable number of nodes. Uniform distribution minimizes the number of nodes in each dimension and consequently optimize the search time. There are infinitely many functions that meet the above requirements. In Algorithm 2, we present a simple method that uniformly maps a range of integer keys into a vector space.

Algorithm 2 Mapping from integer to vector

```
1: function KeyToCoord(int key)
2:     int basis ← ⌈ ∛N ⌉, quotient ← key, k[D]
3:     for i ∈ (D, 0] do
4:         k[i] ← quotient mod basis
5:         quotient ← ⌊quotient ÷ basis⌋
6:     return k
```

We first compute the maximum number of nodes in each dimension based on the range of the keys N. In practice, the range of keys is usually known prior to the execution either explicitly through the specification of the user application or implicitly through the key's data type. For example, if the keys are 32-bit integers and we choose the dimension of MDList to be 8, then in each dimension there are at most 16 (8th root of 2^{32}) keys. We can then obtain the coordinate vector by converting the base of the original key. In the above case, each digit in the 16-based number corresponds to a coordinate in an 8-dimension vector. Given a key of 1000, its 16-based representation is `0x3E8` and the coordinates would be (0,0,0,0,0,3,E,8). In general, to uniformly distribute keys within range $[0, N)$ in a D-dimension space, the maximum number of keys in each dimension is $b = \lceil \sqrt[D]{N} \rceil$. The mapping from an integer key to its vector coordinates is essentially converting it to an b-based number and using each digit as the correspond value in the D-dimension vector.

We focus on unique integer keys in this paper, but duplicate keys can be handled by reserving a few bits to achieve uniqueness as detailed in previous work [4, 18]. If other types of keys such as strings are needed, one needs to devise a custom mapping or simply convert the keys into integers and use the above mapping. Since the dimension of an MDList D affects the memory layout and the computation of vector coordinates, it must be decided before the execution of the program. We provide detailed discussion on the performance impact of different dimensionality in Section 6.

3.5 Search

We outlined the search process for a simple 2DList in Section 3.1. Following the same methodology, we present the search algorithm for a D-dimensional list in Algorithm 3. The Find function returns the node containing the coordinates $\mathbf{k}$ if it exists. Given a D-dimensional list, we search for an element by traversing the dimension d nodes that do not overshoot the node containing the coordinates being searched for (line 3.4)[2]. When it is not possible to make further progress at the current dimension, the search advances to the

[2] Throughout the paper, we use $a.b$ to denote line b in algorithm a

Algorithm 3 Search for a node with coordinates **k**

```
1: function SEARCH(int[] k)
2:     Node*, curr ← head, int d ← 0
3:     while d < D do
4:         while curr ≠ NIL and k[d] > curr.k[d] do
5:             curr ← curr.child[d]
6:         if curr = NIL or k[d] < curr.k[d] then return NIL
7:         else d ← d + 1
8:     return curr
```

next dimension (line 3.7). The outer while loop terminates when the search has visited all of the nodes in the highest dimension (line 3.3). If so, $curr$ must be immediately in front of the node that contains the desired coordinates , i.e. the search must have exhaustively compared every coordinate of $curr$ with **k** and they match. Otherwise, the search failed to proceed to the highest dimension (early termination on line 3.6), and the target node is not in the list. Fig. 2 illustrates a 3DList with up to four nodes in each dimension. To search a node, say $(3, 3, 2)$. The traverse begins at the root node and proceeds through all dimension 0 nodes. It then increases the search dimension d and continues to traverse the 2DList rooted at $(3, 0, 0)$. The search further increases dimension and continues to traverse the 1DList rooted at $(3, 3, 0)$ before reaching the target node.

It is straightforward to deduce that the worst-case time complexity of the search algorithm is $\mathcal{O}(D \cdot M)$ where M is the maximum number of nodes in one dimension. For keys within the range of $[0, N)$, if we uniformly map them into the D-dimensional space using Algorithm 2, M is bounded by $\sqrt[D]{N}$. This gives $\mathcal{O}(D \cdot \sqrt[D]{N})$, which is equivalent to $\mathcal{O}(\log N)$, if we choose $D \propto \log N$ (Note that $\sqrt[\log N]{N} = 2$). This serves as a guideline for choosing D in sequential scenarios. For example, a 32DList that holds 2 nodes in each dimension provides comparable performance to that of balanced search trees with 32-bit integer keys. As the capacity of MDList grows exponentially with respect to its dimensionality, MDList of higher dimensions can accommodate an even wider range of keys.

3.6 Insertion and Deletion

A unique property of MDList, which makes it suitable for concurrent accesses, is the locality of its insertion and deletion: each operation requires updating at most two consecutive nodes in the data structure. For brevity, we outline the sequential insertion and deletion in this section. The pseudo code is presented in Section 4 when we explain in detail the concurrent versions of the algorithms.

The insertion operation involves two steps: node splicing and child adoption. In the first step, we search and splice as depicted by the dashed green arrows in Figure 2. The appropriate insert position of the new node is determined by using a modified version of the SEARCH algorithm, which keeps a record of the predecessor and the dimension of the new node (the index of the new node in its predecessor's child array). Splicing involves pointing to the old child of the predecessor from the new node and updating the child pointer of the predecessor. In Figure 2, we insert a new node $(2, 0, 0)$ between its predecessor $(1, 0, 2)$ and the predecessor's old child $(2, 0, 1)$. The new node becomes a dimension 0 child of its predecessor and the old child becomes a dimension 2 child of the new node. The second step is needed when the dimension of the old child has been changed due to the insertion of a new node. This extra child adoption process ensures that the nodes which are no longer accessible through the old child can be reached through the new node. According to Definition 1 the dimension of a node's children should be no less than the dimension of the node itself. If the dimension of a node increases from d to d', its children within the range of $[d, d')$ must be adopted. As marked by the dotted red arrows in Figure 2, the new node takes over two children $(3, 0, 0)$ and $(2, 1, 0)$ from the old child $(2, 0, 1)$, the dimension of which increased from 0 to 2.

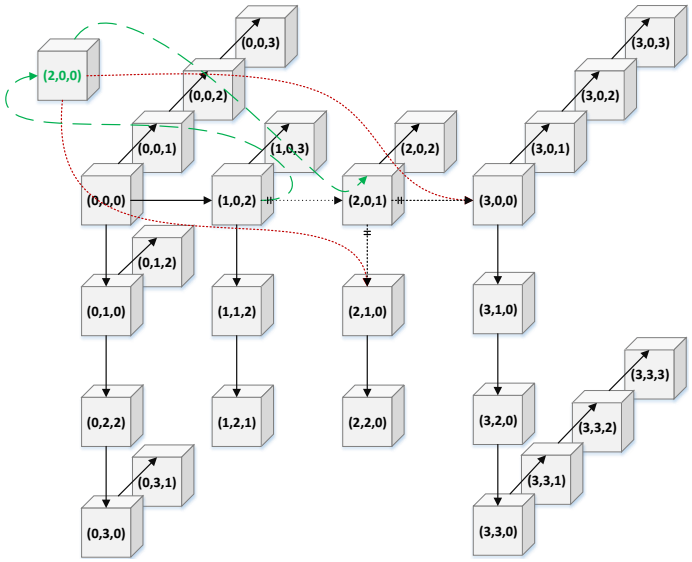


Figure 2: INSERT operation in a 3Dlist

The deletion operation is performed in a similar way: the child (n_c) with the highest dimension of the node being deleted (n) is linked to its parent; next, n transfers its children to n_c, if the dimension of n_c is changed.

4. Lock-free Priority Queue

The proposed MDList provides convenient support for concurrent INSERT operations because of its decentralized structure. As mentioned in Section 3.6, each insertion takes one or two steps and updates no more than two consecutive nodes. We guarantee lock-free progress in the node splicing step by employing the techniques used by Harris's linked list algorithm [5]. Lock-freedom ensures system wide progress while allowing individual threads to starve [9]. To provide for lock-free progress in the child adoption step, we need to announce the operation globally. This allows the interrupting threads help finish the operation in case the insertion thread is preempted.

Algorithm 4 Priority Queue Structures

```
1: class PriorityQueue
2:     const int D, N, R
3:     Node* head
4:     Stack* stack
5:     PurgeDesc* pdesc

6: struct Node
7:     int key, k[D], seq
8:     void* val
9:     Node* child[D], prg
10:    AdoptDesc* adesc

11: struct AdoptDesc
12:     Node* curr
13:     int dp, dc

14: struct PurgeDesc
15:     Node* head, prg

16: struct Stack
17:     Node* head, node[D]
```

Algorithm 5 Pointer Marking

```
1: const int F_adp ← 0x1, F_prg ← 0x2, F_del ← 0x1
2: define SetMark(p, m) (p | m)
3: define ClearMark(p, m) (p & ~m)
4: define IsMarked(p, m) (p & m)
```

The delete method described in Section 3.6 poses a scalability challenge for concurrent DELETEMIN operations because constant child adoption incurs heavy contention on the head node. To avoid

Algorithm 6 Concurrent Insert

```
1: function INSERT(int key, void* val)
2:   Node* node, pred, curr      ▷ new node, its parent and child
3:   int dp, dc                  ▷ dimension of node in pred and curr
4:   AdoptDesc* ad               ▷ descriptor for child adoption task
5:   Stack* s                    ▷ stack of search path for the insertion
6:   PurgeDesc* pd               ▷ descriptor object for purge operation
7:   node ← new Node, node.key ← key, node.val ← val
8:   node.k[0 : D] ← KEYTOCOORD(key)[0 : D]
9:   node.child[0 : D] ← NIL
10:  while true do
11:    pd ← pdesc
12:    if pd ≠ NIL then FINISHPURGING(pd)
13:    pred ← NIL, curr ← head, dp ← 0, dc ← 0
14:    s ← new Stack, s.head ← curr
15:    LOCATEPRED( )
16:    if dc = D then break
17:    ad ← pred.adesc
18:    if ad ≠ NIL and dp ∈ [ad.dp, ad.dc] then
19:      FINISHINSERTING(pred, ad)
20:      continue
21:    ad ← curr ≠ NIL ? curr.adesc : NIL
22:    if ad ≠ NIL and dp ≠ dc then
23:      FINISHINSERTING(curr, ad)
24:    ad ← NIL
25:    if dp ≠ dc then
26:      ad ← new Desc
27:      ad.curr ← curr, ad.dp ← dp, ad.dc ← dc
28:    node.child[0 : dp] ← F_adp
29:    node.child[dp : D] ← NIL
30:    node.child[dc] ← curr, node.adesc ← ad
31:    if CAS(&pred.child[dp], curr, node) then
32:      if ad ≠ NIL then
33:        FINISHINSERTING(node, ad)
34:      REWINDSTACK( )
35:      break

36: inline function LOCATEPRED( )
37:   while dc < D do
38:     while curr ≠ NIL and node.k[dc] > curr.k[dc] do
39:       pred ← curr, dp ← dc
40:       curr ← curr.child[dc]
41:     if curr = NIL or node.k[dc] < curr.k[dc] then
42:       break
43:     else
44:       s.node[dc] ← curr, dc ← dc + 1
```

such performance penalties, we introduce several improvements to the existing logical deletion techniques. Namely, we use a *deletion stack* that hints the search operation about the next minimal node. This prevents deletion operations from repeatedly traversing logically deleted nodes as is observed in [12, 16]. We also design a stack rewind mechanism that allows insertions to proceed optimistically without requiring logically deleted nodes to form a prefix, as is the case in [12]. As a result, our approach presents a relaxation to the DELETEMIN semantics in order to trade for overall throughput.

We define the structure of the concurrent priority queue in Algorithm 4, which also contains definitions for the deletion stack and the descriptor objects. A descriptor object [9] is a data structure that stores operation context and guides helping threads to finish the operation. The priority queue is initialized with a dummy head node, which has the minimal key 0. The node array in the deletion stack is initially filled with the dummy head. We employ the pointer marking technique described by Harris [5] to mark adopted child nodes as well as logically deleted nodes. The macros for pointer marking are

Algorithm 7 Child Adoption

```
1: function FINISHINSERTING(Node* n, AdoptDesc* ad)
2:   Node* child, curr ← ad.curr
3:   int dp ← ad.dp, dc ← ad.dc
4:   for i ∈ [dp, dc) do
5:     repeat
6:       child ← curr.child[i]
7:     until !ISMARKED(child, F_adp) and !CAS(&curr.child[i],
       child, SETMARK(child, F_adp)
8:     child ← CLEARMARK(child, F_adp)
9:     CAS(&n.child[i], NIL, child)
10:  CAS(&n.adesc, ad, NIL)
```

defined in Algorithm 5. F_{adp} and F_{prg} flags are co-located with the $child$ pointers while F_{del} flag is co-located with the prg field.

4.1 Concurrent Insert

We list the concurrent INSERT function in Algorithm 6. The insertion operation locates the target position and updates the child pointer of the predecessor node. The search begins from the head of the queue (line 6.13). Given a new node, LOCATEPRED[3] tries to determine its immediate parent $pred$ and child $curr$ (line 6.15 and 6.36). It also keeps the dimension of the node being inserted in dp and the dimension of the child in dc (line 6.39 and 6.39). The new node should be inserted as the dimension dp child of the $pred$ node, while a non-empty $curr$ node will become the dimension dc child of the new node. The deletion stack s is also updated during the search (the usage of s will be detailed in the next two sections). The conditional check on line 6.16 is true when a node with the same coordinates already exists, thus the insertion terminates. If the progress of the insertion is delayed, it helps finish any pending child adoption tasks on nearby nodes. The code between lines 6.17 and 6.23 reads the $adesc$ fields from $pred$ and $curr$ and tests if helping is needed. Since a child adoption process updates the children indexed from dp to dc, the insertion must help $pred$ node if it intends to insert the new node into this range. Likewise, the insertion does not need to help $curr$ node if it is not going to adopt children from $curr$ node. Prior to atomically updating the link to the new node, we fill the remaining fields of the new node (line 6.24 and 6.30). If the new node needs to adopt children from $curr$ node, we use an adopt descriptor to store the necessary context (line 6.27). The pointers within the range $[0, dp)$[4] of the new node's child array are marked with F_{adp}. This effectively invalidates these positions for future insertions. The pointers within the range $[dp, D]$ are set to NIL meaning they are available for attaching child nodes. On line 6.31, the standard COMPAREANDSWAP (CAS) atomic synchronization primitive is used to update the $pred$ node's child pointer. The CAS would fail under three circumstances: 1) the desired child slot has been updated by another competing insertion; 2) the desired child slot has been invalidated by a child adoption process; and 3) the desired child slot has been invalidated by a purge process. If any of the above cases is true, the loop restarts. Otherwise, the insertion proceeds to finish its own child adoption process and rewind the deletion stack. The deletion stack needs to be rewound if the new node was inserted into a position that cannot be reached by future DELETEMIN operations. We explain this synchronization mechanism in details in Section 4.3.

The FINISHIINSERTING function in Algorithm 7 performs child adoption on a given node n with the descriptor ad. This is a helping procedure that must correctly handle duplicate and simultaneous executions. The function first reads the adoption context from the descriptor into its local variables. It then transfers $curr$ node's chil-

[3] We use **inline** functions so that they have access to the caller's local variables without implicit argument passing.

[4] We use indexing notion $[a : b]$ to address elements within the range of $[a, b)$.

Algorithm 8 Concurrent DeleteMin

```
1: function DELETEMIN( )
2:   Node* min, prg, last, child
3:   AdoptDesc* ad                          ▷ descriptor for child adoption task
4:   PurgeDesc* pd                          ▷ descriptor for batch deletion task
5:   Stack* s, s_old                        ▷ new and old deletion stack
6:   min ← NIL, s_old ← stack, s ← new Stack, *s ← *s_old
7:   for d ← D − 1; d > 0 do
8:     last ← s.node[d], ad ← last.desc
9:     if ad and d ∈ [ad.dp, ad.dc) then
10:      FINISHINSERTING(last, ad)
11:    child ← CLEARMARK(last.child[d], F_adp|F_prg)
12:    if child = NIL then
13:      d ← d − 1
14:      continue
15:    prg ← child.prg
16:    if ISMARKED(prg, F_del) then
17:      if CLEARMARK(prg, F_del) = NIL then
18:        s.node[d : D] ← child, d ← D − 1
19:      else
20:        s.head ← CLEARMMARK(prg, F_del)
21:        s.node[d : D] ← s.head, d ← D − 1
22:    else if CAS(&child.prg, prg, SETMARK(prg, F_del)) then
23:      s.node[d : D] ← child, min ← child
24:      CAS(&stack, s_old, s)
25:      if distance > R then
26:        pd ← new PurgeDesc
27:        pd.head ← s.head, pd.prg ← child
28:        if CAS(&pdesc, NIL, pd) then
29:          FINISHPURGING(pd)
30:      break
31:  return min
```

Algorithm 9 Batch Physically Deletion

```
1: function FINISHPURGING(PurgeDesc* pd)
2:   Node* curr, child, hd, prg, hd_new, prg_copy
3:   hd ← pd.head, prg ← pd.prg
4:   if hd ≠ head then return
5:   hd_new ← new Node, prg_copy ← new Node
6:   *prg_copy ← *prg, prg_copy.child[0 : D] ← NIL
7:   hd_new.prg ← F_del, hd_new.seq ← hd.seq + 1
8:   for d ← 0, curr ← hd; d < D do
9:     while prg.k[d] > curr.k[d] do
10:      curr ← CLEARMARK(curr.child[d], F_adp|F_prg)
11:      desc ← curr.desc
12:      if desc ≠ NIL and d ∈ [desc.dp, desc.dc) then
13:        FINISHINSERTING(curr, desc)
14:    repeat
15:      child ← curr.child[d]
16:    until !ISMARKED(child, F_adp|F_prg) and !CAS(&curr.child[i], child, SETMARK(child, F_prg))
17:    if !ISMARKED(child, F_prg) then
18:      curr ← hd, d ← 0
19:      continue
20:    child ← CLEARMARK(child, F_prg)
21:    if curr = hd then
22:      hd_new.child[d] ← child, prg_copy.child[d] ← F_del
23:    else
24:      prg_copy.child[d] ← child
25:      if d = 0 or prg_copy.child[d − 1] = F_del then
26:        hd_new.child[d] ← prg_copy
27:    d ← d + 1
28:  CAS(&hd.prg, F_del, SETMARK(prg, F_del))
29:  CAS(&prg.prg, F_del, SETMARK(hd_new, F_del))
30:  CAS(&head, hd, CLEARMARK(prg.prg, F_del))
31:  CAS(&pdesc, pd, NIL)
```

dren within the range of $[dp, dc)$ to n. Before a child pointer can be copied, we must safeguard it so that it cannot be changed while the copy is in progress. This is done by setting the F_{adp} flag in the child pointers (line 7.7). Once the flag is set either by this thread through a successful CAS operation or by other threads, the function proceeds to copy the pointer to n (line 7.9). Finally, the descriptor field in n is cleared to designate the operation's completion.

4.2 Concurrent DeleteMin

One limitation with previous logical deletion approaches is that they need to traverse all logically deleted nodes before reaching the target. This proves to be troublesome especially for an MDList-based priority queue. Since an MDList is a rooted tree, reaching the minimal node is done by a procedure similar to depth first search. As the search extends to higher dimensions, it traverses exponentially more nodes, which slows down logical deletion. Additionally, tree traversal algorithms usually involve recursion that leads to poor performance because of the overhead of function calls. We design an iteration-based DELETEMIN algorithm by using a *deletion stack*. The deletion stack, as defined in Algorithm 4 is an array of D node pointers with an extra head pointer. The node at index d corresponds to a d-dimensional node in the MDList, and all nodes in the array form a path through which the target node can be reached. The advantage of employing the deletion stack is twofold: 1) it reduces node traversal by providing hints about the position of the next minimal node; 2) it converts the typically recursion-based search algorithm into an iteration-based one.

Algorithm 8 lists the concurrent DELETEMIN operation, which traverses the priority queue and finds the first node that is not marked for deletion. It starts from reading the stack (s_{old}) and making a local copy, s (line 8.6). The search iterates through the stack starting from the highest dimension (line 8.7) because children with smaller keys are assigned higher dimensionality according to Definition 2. The last known minimal node ($last$) in dimension d is read from the stack, and the new minimal node ($child$) must be a dimension d child of $last$. This is because we have traversed nodes with smaller keys in previous iterations. We clear both F_{adp} and F_{prg} from $child$ and test if it is empty. If so, we continue to search in lower dimensions. We test if the node has already been logically deleted on line 8.16. A node is considered logically deleted once its prg field is marked with F_{del}. If $child$ is marked, which could happen due to competing DELETEMIN operations, we update the local stack and proceed to find the next minimal node. The prg field of a purged node points to the new head of the queue. If $child$ is not purged as indicated by a NIL prg field on line 8.17, the local stack is updated with $child$. Otherwise, the stack is set to the new head. In both cases, the search starts anew from the highest dimension. On line 8.22, we try to set the deletion flag in case $child$ is not already marked for deletion. Upon success, we update the stack to reflect the new minimal node, and try to announce the stack globally (line 8.24). We do not need to retry posting the announcement because DELETEMIN operations can always reach unmarked nodes from the old stack. Starting from the most recent stack helps boost the search speed. We determine if a physical deletion is needed (line 8.25) by comparing the distance of a node to the user-defined threshold R (values ranging from 8 to 64 prove to perform well in our empirical study). The distance is measured by the number of intermediate nodes between the newly marked node and the head node, which can be estimated by counting the number of logically deleted nodes traversed by the search process.

The physical deletion or purge is announced globally through PURGEDESC, and executed by calling FINISHPURGING, which is

Algorithm 10 Rewind Deletion Stack

```
1:  inline function REWINDSTACK()
2:    Stack* s_new ← stack
3:    Node* prg
4:    repeat
5:      if s.head.seq = s_new.head.seq then
6:        if node.key ≤ s_new.node[D − 1].key then
7:          s.node[dp : D] ← pred
8:        else if firsttime then *s ← *s_new
9:        else break
10:     else if s_new.head.seq > s.head.seq then
11:       prg ← CLEARMARK(s.head.prg)
12:       if prg.key ≤ node.key then
13:         s.head ← CLEARMARK(prg.prg, F_del)
14:         s.node[0 : D] ← s.head
15:       else s.node[dp : D] ← pred
16:     else
17:       prg ← CLEARMARK(s_new.head.prg)
18:       if prg.key ≤ s_new.node[D − 1].key then
19:         s.head ← CLEARMARK(prg.prg, F_del)
20:         s.node[0 : D] ← s.head
21:       else if firsttime then *s ← *s_new
22:       else break
23:   until !CAS(&stack, s_new, s) and !ISMARKED(node.prg, F_del)
```

listed in Algorithm 9. We read the head (hd) and the purged node (prg) from the descriptor. The goal is to discard all nodes between hd and prg from the priority queue so that they are not visible to future operations. To allow simultaneous execution of this helping function, we allocate and modify local copies of the new head (hd_{new}) and the purged node (prg_{copy}) before committing them. The seq field in the head nodes serves as a version counter that is increased every time a new head is created (line 9.7). Starting from hd, the process tries to find one node in each dimension with the coordinate that is no less than that of prg (line 9.9). Once found, the node ($curr$) divides the list in its dimension such that its predecessors will be purged and its successors will be kept. The next step is to transfer the dimension d child of $curr$ to either hd_{new} or prg_{copy}. To safeguard the child, we employ the same pointer marking technique as the child adoption process except using a different flag F_{prg} (line 9.16). If the child has been adopted (line 9.17), the search process needs to start over again from hd (line 9.17). The child is transferred to the new head if it is a child of the old head. Otherwise it is transferred to prg_{copy}. The prg field of hd and prg is updated allowing DELETEMIN to reach the new head (line 8.19). Finally, the new head is updated and the descriptor is cleared.

4.3 Synchronizing Insert with DeleteMin

In our implementation, INSERT and DELETEMIN are synchronized through the stack rewind mechanism. Intuitively, a deletion thread moves the deletion stack "forward" while an insertion thread "rewinds" the stack when it detects that the stack points to a position beyond the new node. The insertion rewinds the stack to a position before the newly inserted node so that it is made visible to future deletion threads. This allows insertion threads to proceed optimistically without blocking deletion threads until the new nodes are in place.

The REWINDSTACK function listed in Algorithm 10 consists of a CAS-based loop that reads the deletion stack and determines the rewind position. An insertion records the head node in its local stack on line 6.14. Based on the version (determined by the seq field) of the head node from the new stack, the rewind falls into three scenarios. In the first case, the head node is still the same (line 10.5). If the last node that has been deleted has a key larger than the new node, we need to rewind the stack to the new node's predecessor (line 10.7). Otherwise, deletion threads have yet to reach the new node and we can skip the rewind except for the first iteration. In that case we try to update the stack with a new object with the same contents in order to make sure the stack will not be overwritten by other threads (line 10.8). In the second case, the head from the new stack is newer than the head from the local stack (line 10.10), which means the head of the queue where the insertion took place has been purged. Under such circumstances, the prg field of the old head should contain a pointer to the purged node. If the key of the purged node is no greater than the key of the new node, we have to rewind the stack as far as the new head of last purge. Otherwise, we rewind the stack to the predecessor of the new node as we did in the first case. In the third case, the head from the new stack is older than the head from the local stack. Here, we only need to rewind the stack to the new head node to ensure the new node is reachable (line 10.19).

5. Correctness

In this section, we sketch a proof of the main correctness property of the presented priority queue algorithm, which is quiescent consistency. Additionally, we discuss how stricter correctness conditions such as linearizability can be guaranteed through the use of timestamps. We explain the relaxation of the DELETEMIN operation when the concurrent object is executed without quiescent periods. We begin by defining the *abstract state* of a sequential priority queue and then show how to map the internal state of our concrete priority queue object to the abstract state. We denote the abstract state of a sequential priority queue to be a set P. Equation 1 specifies that an INSERT operation grows the set if the key being inserted does not exist. Equation 2 and 3 specify that a DELETEMIN operation shrinks a non-empty set by removing the key-value pair with the smallest key.

$$\text{INSERT}(\langle k, v\rangle) = \begin{cases} P \cup \{\langle k, v\rangle\} & \forall \langle k', v'\rangle \in P\ ,\ k' \neq k \\ P & \exists \langle k', v'\rangle \in P : k' = k \end{cases} \quad (1)$$

$$\text{DELETEMIN}() = \begin{cases} P \setminus \langle k, v\rangle & P \neq \emptyset \\ \emptyset & P = \emptyset \end{cases} \quad (2)$$

$$\text{where } \forall \langle k', v'\rangle \in P,\ k' \neq k \implies k' > k \quad (3)$$

5.1 Invariants

Now we consider the concurrent priority queue object. By a *node*, we refer to an object of type **Node** that has been allocated and successfully linked to an existing node (line 6.31). We denote the $head$ node with $seq = i$ by $head_i$, and the set of nodes that are reachable from $head_i$ by L_i. At any point of time, the set of all nodes can be denoted by $L = \bigcup_{i=0}^{m} L_i$ where $m = head.seq$. The following invariants are satisfied by the concrete priority queue object at all times. Invariant 1 states that if a node has no pending child adoption task, its dimension d child must have d invalid child slots leaving $D - d$ valid ones.

Invariant 1. $\forall n, n' \in L, \text{CLEARMARK}(n.child[d], F_{adp}|F_{prg}) = n' \wedge n.adesc = NIL \implies \forall i \in [0, d), \text{ISMARKED}(n'.child[i], F_{adp}) =$ ***true***

Proof. By observing the statements at line 6.28 and 7.7 we see that the F_{adp} flags are properly initialized before linking a new node to its predecessor and updated properly whenever a child is adopted. □

Invariant 2. $\forall n \in L_i, \exists p = \{d_0, d_1, ..., d_m\} : d_0 \leq d_1 \leq ... \leq d_m \wedge head_i.child[d_0].child[d_1]...child.[d_m] = n$

Proof. Initially, the invariant holds because any new node can be reached from $head_i$ following the path given by LOCATEPRED. The path may be altered by subsequent insertions. Since we do not unlink nodes from the data structure, the claim follows by noting that an

insertion adds a new node either between two contiguous nodes or as a leaf node. Future insertions either replace a node in p or add a new node. □

We now show that the nodes without a deletion mark and accessible through the deletion stack form a strictly well-ordered set that is equivalent to P. Invariant 3 states that the ordering property described by Definition 2 is kept at all times.

Invariant 3. $\forall n, n' \in L, n.child[d] = n' \implies n.key < n'.key \wedge \forall i \in [0, d)\ n.k[i] = n'.k[i] \wedge n.k[d] < n'.k[d]$

Proof. Initially the invariants trivially holds. The linkage among nodes is only changed by insertion, child adoption and purge. Insert preserves the invariants because the condition checks on line 6.38 and 6.41 guarantee that $\forall i \in [0, dp)\ pred.k[i] = node.k[i] \wedge pred.k[dp] < node.k[dp]$. Child adoption preserves the invariant because $\forall i \in [dp, dc)\ node.k[i] = curr.k[i] < curr.child[i].k[i]$. Similarly, purge preserves the ordering property because of the condition check on line 9.9. □

Invariant 4. $\forall d_1, d_2 \in [0, D), d_1 < d_2 \implies \forall i \in [0, d_1], stack.node[d_1].k[i] = stack.node[d_2].k[i] \wedge stack.node[d_1].k[d_2] < stack.node[d_2].k[d_2]$

Proof. Initially, the stack contains a dummy head node and the invariant holds. DELETEMIN sets the value of the stack's nodes in the range of $[d, D)$ to *child* (line 8.23). Following Invariant 3, *child* has a larger key than $s.node[d]$. INSERT updates the stack with the path recorded in LOCATEDPRED, according to Invariant 2 and 3 the path contains nodes with increasing keys. □

Invariant 5. $\forall d \in [0, D), stack.node[d].child[d].key > stack.node[D-1].key$

Proof. Following Invariant 3, $stack.node[d].child[d].k[d] > stack.node[d].k[d]$. Following Invariant 4, $stack.node[D-1].k[d] = stack.node[d].k[d]$. □

In summary, Invariant 5 states that the deletion stack splits the nodes in L_i into two groups: those that are reachable by the DELETEMIN algorithm and those that are not. We define the latter by $M_i = \{n | n \in L_i \wedge n.key \leq stack.node[D-1].key\}$, and $M = \bigcup_{i=0}^{m} M_i$. We also define the set of logically deleted nodes by $S = \{n | n \in L \wedge \text{ISMARKED}(n.prg, F_{del}) = \textbf{true}\}$. The abstract state can then be defined as $P \equiv L \setminus M \setminus S$.

5.2 Quiescent Consistency and Linearizability

We now sketch a proof that our algorithm is a quiescently consistent priority queue implementation that complies with the abstract semantics. Quiescent consistency states that method calls separated by a period of quiescence should appear to take effect according to their real time order [9].

Theorem 1. *A successful* INSERT$(\langle k, v\rangle)$ *operation that does not overlap with any* FINISHPURGING*, i.e. following a period of quiescence, takes effect atomically at one statement.*

Proof. If an INSERT operation returns on line 10.9, 10.22, or the second condition on line 10.23, the deletion stack is not updated. The decision point for such an operation to take effect is when the CAS operation on line 6.31 succeeds. The remaining CAS operations in the child adoption process will eventually succeed according to Lemma 2. If an INSERT operation needs to rewind the deletion stack, i.e. $\langle k, v\rangle \in M \implies \langle k, v\rangle \notin P$, the decision point for it to take effect is when the CAS operation on line 10.23 succeeds. The newly updated deletion stack ensures that $stack.node[D-1].key \leq key$, thus renders the new node reachable for DELETEMIN operations, i.e. $M = M \setminus \{\langle k, v\rangle, ...\}$. Equation 1 holds in both cases because $L = L \cup \langle k, v\rangle \wedge \langle k, v\rangle \notin M$. □

Theorem 2. *A* DELETEMIN *that does not overlap with any* INSERT *operation takes effect atomically at one statement.*

Proof. A DELETEMIN operation updates the abstract state by growing S. The decision point for it to take effect is when the CAS operation on line 8.22 successfully marks a node for deletion, or on line 8.7 when it reaches the last node in the deletion stack without finding an unmarked node. In the first case, Equations 2 and 3 hold because $S' = S \cup \langle k, v\rangle \implies P' = P \setminus \{\langle k, v\rangle\}$. In the second case, $P' = P = \emptyset$. □

In our algorithm, the INSERT operation is linearizable with respect to other concurrent INSERT operations, and the DELETEMIN operation is linearizable with respect to other DELETEMIN operations. However, the DELETEMIN operation is not linearizable with the overlapping INSERT operations because only those nodes $\{n | n \notin M_i\}$ are reachable from $stack_i$. At the beginning of the DELETEMIN operation, the deletion stack is read once on line 8.6. The subsequent traversal will not be aware of any unmarked nodes removed from M. One method of designing a linearizable algorithm is the time-stamping mechanism adopted by Shavit and Lotan in [16], in which each deletion thread returns the minimal undeleted node among those inserted completely before it reads the deletion stack. After a node is inserted on line 6.31, it acquires a time-stamp. A deletion thread notes the time at which it reads the stack and only processes nodes with a smaller time-stamp. This time-stamping mechanism ensures that the DELETEMIN operations see a consistent abstract state throughout the search process.

5.3 Relaxation

Our algorithm is relaxed in the sense that it is quiescently consistent. We prefer quiescent consistency over linearizability because it allows for higher level of concurrency. The trade-off is that in concurrent executions without any quiescence the order in which method calls take effect may be undefined.

Lemma 1. *Purge does not change the abstract state P.*

Proof. Note that in Algorithm 9, L does not change because no new node is inserted; M does not change because the deletion stack is not modified; S does not change because no deletion flag is marked. □

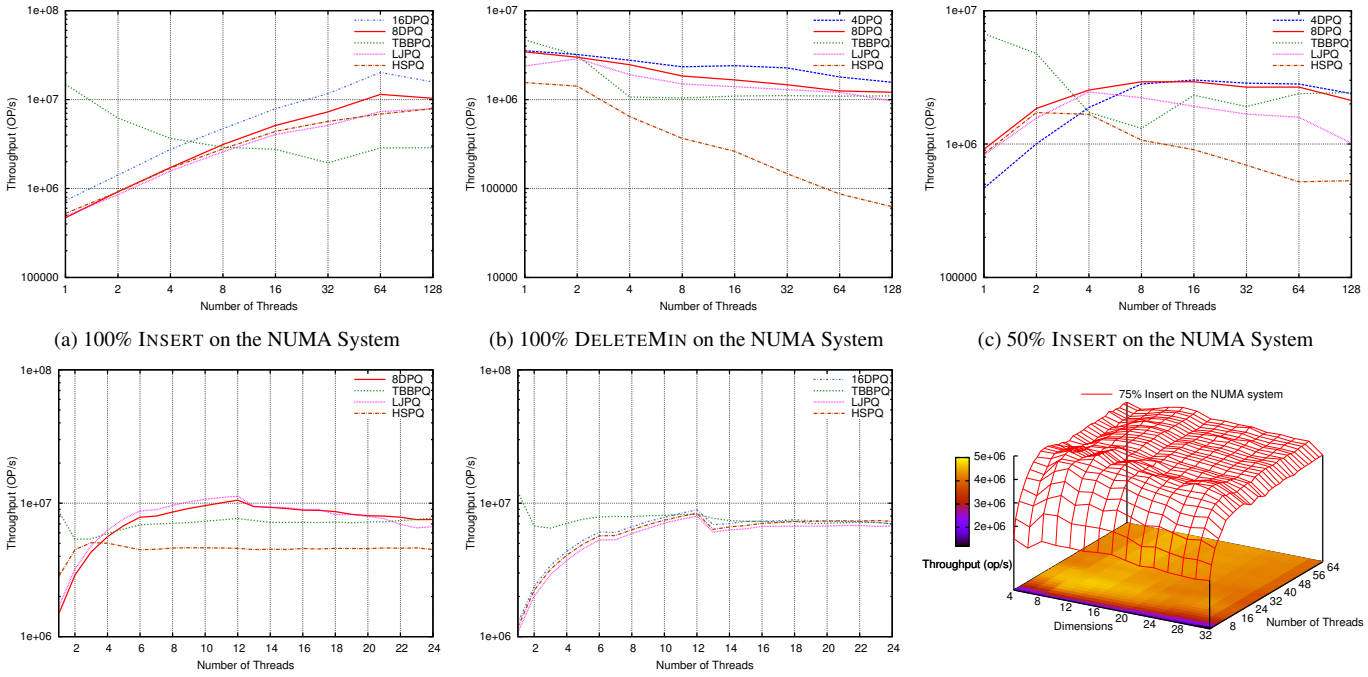
Theorem 3. *Let key_{prg} denote the key of the last prg node in Algorithm 9. Without quiescence, a successful* DELETEMIN *operation either returns $n \in P : \forall n' \in P,\ n.key < n'.key$ or $n \in P : n.key < key_{prg}$.*

Proof. The purge algorithm creates a new head node while keeping the old head, thus the ordering relation between two set of nodes $Q = \{n | n \in L_i \wedge n.key \leq key_{prg}\}$ and $Q' = \{n' | n' \in L_{i+1} \wedge n'.key \leq key_{prg}\}$ is not defined. The relaxation takes place when INSERT operations overlap with FINISHPURGE because new nodes can be inserted into either set depending on when the insertion starts. If there are quiescent periods between insertions and purges, all new nodes are inserted into Q', which implies $Q \subseteq S$. Otherwise, $\exists n \in Q, n' \in Q' : n \notin S \wedge n' \notin S \wedge n.key > n'.key$. When the DELETEMIN operation switches to the new head on line 8.19 the strict ordering property is temporarily relaxed. □

5.4 Lock Freedom

Our algorithm is lock-free because it guarantee that for every possibly execution scenario, at least one thread makes progress. We prove this by examining unbounded loops in all possible execution paths, which can delay the termination of the operations.

Lemma 2. FINISHINSERTING *(Algorithm 7) and* FINISHPURGING *(Algorithm 9) complete in finite steps.*



(a) 100% INSERT on the NUMA System (b) 100% DELETEMIN on the NUMA System (c) 50% INSERT on the NUMA System

(d) 50% INSERT on the SMP System (e) 80% INSERT on the SMP System (f) Dimensionality and Performance

Figure 3: Throughput of the Priority Queues (MDPQs are named by their dimensionality, e.g. 8DPQ is an MDPQ with 8 dimensions)

Proof. We observe that the `repeat` loops on line 7.7 and 9.16 are subject to fail and retry when another insertion thread concurrently updates $curr.child[i]$. The number of retries is bounded by $\sqrt[D]{N}$, which is the maximum number of nodes in each dimension. For FINISHPURGING, the `for` loop (line 9.8) is also unbounded. According to Invariant 2, the purged node can always be reached from the head through a sequence of child pointers and the number of reties is bounded by $D\sqrt[D]{N}$. □

Theorem 4. INSERT *and* DELETEMIN *operations are lock-free.*

Proof. Note that all shared variables are concurrently modified by CAS operations, and the CAS-based unbounded loops (line 6.31, 8.22, and 10.23), only retry when a CAS operation fails. This means that for any subsequent retry, there must be one CAS that succeeded, which caused the termination of the loop. All reads of child pointer are preceded by FINISHINSERTING, which completes child adoption in finite steps to ensure consistency. Furthermore, our implementation does not contain cyclic dependencies between CAS-based loops, which means that the corresponding operation will progress. □

6. Experimental Evaluation

We compare the performance of our algorithm (MDPQ) against Intel TBB's concurrent priority (TBBPQ) [15], Linden and Jonsson's linearizable priority queue (LJPQ) [12], and Harris and Shavit's quiescently consistent priority queue (HSPQ) [9]. Intel TBB is an established industry standard concurrent library. TBBPQ is based on an array-based heap and employs a dedicated aggregator thread to perform all operations. LJPQ is the best available linearizable priority queue that minimizes contention on the head node. HSPQ shares the same correctness guarantee as our algorithm. Both LJPQ and HSPQ are built on top of Fraser's [4] state of the art lock-free skiplist implementation. We employ a micro-benchmark to evaluate the performance of these approaches for uniformly distributed keys. This canonical evaluation method [5, 12, 16, 18] consists of a tight loop that randomly chooses to perform either an INSERT or a DELETEMIN operation. The tests are conducted on a 64-core NUMA system (4 AMD opteron CPUs with 16 cores per chip @2.1 GHz) and a 12-core SMP system (1 Intel Xeon 6-core CPU with hyper-threading @2.9GHz). Both the micro-benchmark and the priority queue implementations are compiled with GCC 4.7 with level three optimizations.

Figures 3a, 3b and 3c illustrate the throughput of the algorithms on the NUMA system. The y-axis represents the throughput measured by *operation per second*, and the x-axis represents the number of threads. Both axes are in logarithmic scale. In Figure 3a, threads perform solely INSERT operations. We observe that the skiplist-based and MDList-based approaches explore fine-grained parallelism and exhibit similar scalability trends. The throughput increases linearly until 16 threads, and continues to increase at a slower pace until 64 threads. Because executions beyond 16 threads span across multiple chips, the performance growth is slightly reduced due to the cost of remote memory accesses. The executions are no longer fully concurrent beyond 64 threads, thus the overall throughput is capped and may even reduce due to context switching overhead. The performance of LJPQ are on par with HSPQ because they employs identical skiplist insertion algorithms. Our priority queue based on an 8DList (8DPQ) achieves a 26% speedup over LJPQ on 16 threads and a further 55% speedup on 64 threads. Each insertion in MDPQ modifies at most two consecutive nodes, incurring less remote memory access than skiplist-based approaches. When we increase the dimensionality of MDPQ to 16, we obtain on average a 50% throughput gain over 8DPQ. A 16DPQ contains at most 4 nodes in each dimension. In order to reach the target position the search operation traverses a maximum of 64 nodes comparing to 128 nodes in a 8DPQ. Further increases in dimensionality can result in diminishing returns for concurrent INSERT. The growing number of child pointers causes synchronization overhead that cancels the benefit of having less nodes in each dimension. TBBPQ, on the other hand, exhibits reduced throughput when increasing the number of threads. Its heap-based structure achieves high throughput on a single thread, but the

aggregator effectively serializes all operations and limits the throughput in concurrent executions.

Figure 3b shows the results for DELETEMIN operations. We fill the data structures with one million elements and measure the time it takes to dequeue them. The overall throughput decreases as the number of threads increases, which matches the observation that DELETEMIN is the sequential bottleneck of a priority queue algorithm. Throughput of HSPQ drops significantly because it initiates physical deletion for every logically deleted node. The physical deletion constantly swings pointers which causes heavy content on the head node. 8DPQ and LJPQ achieve comparable performance because they both employ batch physical deletion, which reduces the total number of pointer updates. 4DPQ outperforms LJPQ by 50% starting from 16 threads because the number of pointers, which the deletion needs to update, reduces as dimensionality decreases. TBBPQ is able to keep a constant throughput starting from 4 threads. As the DELETEMIN operation is intrinsically sequential, having a dedicated aggregator helps relieve contention by allowing other threads to wait while a single thread performs the update.

We show the results where threads randomly choose operations with a distribution of 50% INSERT and 50% DELETEMIN in Figure 3c. As the level of concurrency increases, the overall throughput peaks at some point where the executions strike a balance between the increasing throughput of INSERT and the decreasing throughput of DELETEMIN. 8DPQ exhibits the best overall throughput, outperforming LJPQ by 50% on average. Our algorithm allows insertion of new nodes into a position before a logically deleted node. The concurrent executions of INSERT and DELETEMIN guarantees quiescent consistency, which is relaxed compared to linearizability. This helps improve throughput over Linden's logical deletion scheme where deletion threads cannot proceed past an ongoing insertion [12]. 4DPQ achieves about 10% speedup over 8DPQ on 64 and 128 threads at the price of being slower when the number of threads is low.

Figure 3d and 3e show the throughput of the algorithms on the SMP system. The x-axis in these graphs is in linear scale. In Figure 3d, the test operations consist of an equal amount of INSERT and DELETEMIN operations. 8DPQ and LJPQ perform equally well, obtaining 30% speedup over TBBPQ and 50% over HSPQ with 12 threads. Executions beyond 12 threads are preemptive, and the overhead of context switching leads to a reduction of 8DPQ and LJPQ's throughput. In Figure 3e, we increase the ratio of INSERT operations to 80%. This is to simulate the typical access pattern of best-first search algorithms, in which a worker thread insert multiple new nodes (search state) upon dequeuing the node with highest priority [1]. In this case, all approaches except TBBPQ exhibit identical scalability trends. 16DPQ obtains 7% more throughput than HSPQ on 12 threads. TBBPQ obtains the same amount of throughput in both cases. Its compact heap structure is specifically optimized for Intel chips, which provides larger cache than the AMD CPU.

In Figure 3f, we sweep the dimension of MDPQ from 4 to 32 on the NUMA system, and show that the algorithm achieves maximum throughput with 12 dimensions on 16 threads. On all scale levels, we see that the throughput converges towards 12 dimensions. This means that the way the dimensionality of an MDPQ affects its performance is independent from the number of threads. The performance of MDPQ can be optimized if the access pattern of the user application is taken into account. The parametric sweep also reveals that the overall throughput is capped with 16 threads. This implies that the algorithms with inherent sequential semantics, such the DELETEMIN operation, pose scalability challenges for NUMA systems.

Overall, MDPQ excels at high levels of concurrency. The locality of its operations makes it suitable for NUMA architectures where remote memory access incurs considerable performance penalties. On an SMP system with low concurrency, MDPQ performs equally well and sometimes slightly better than the state of the art skiplist-based approaches.

7. Conclusion

In this paper, we introduced a lock-free priority queue design based on a novel multi-dimensional list. We exploited spatial locality to increase the throughput of the INSERT operation, and adopted quiescent consistency to address the sequential bottleneck of the DELETEMIN operation. When compared to the best available skiplist-based and heap-based algorithms, our algorithm achieved performance gains in all scenarios and an average of 50% performance improvement on a 64-core multiprocessor system. The proposed MDList guarantees logarithmic worst-case search time by mapping keys into high dimensional coordinates. The performance of an MDList-based data structure can be tailored to different access patterns by changing its dimensionality. Furthermore, an MDList provides a scalable alternative to skiplists and search trees, which opens up opportunities for implementing other multiprocessor data structures, such as dictionaries and sparse vectors.

References

[1] E. Burns, S. Lemons, W. Ruml, and R. Zhou. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, 39(1):689–743, 2010.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[3] F. Ellen, D. Hendler, and N. Shavit. On the inherent sequentiality of concurrent objects. *SIAM Journal on Computing*, 41(3):519–536, 2012.

[4] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5, 2007.

[5] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing*, pages 300–314. Springer, 2001.

[6] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 317–328. ACM, 2013.

[7] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[8] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.

[9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[10] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[11] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157, 1996.

[12] J. Lindén and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *Principles of Distributed Systems*, pages 206–220. Springer, 2013.

[13] W. Pugh. Concurrent maintenance of skip lists. 1990.

[14] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[15] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

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

[17] N. Shavit and A. Zemach. Scalable concurrent priority queue algorithms. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 113–122. ACM, 1999.

[18] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[19] M. Wimmer, D. Cederman, F. Versaci, J. L. Träff, and P. Tsigas. Data structures for task-based priority scheduling. *arXiv preprint arXiv:1312.2501*, 2013.