A First Approach Towards Designing NUMA-Aware Concurrent Priority Queues

Authors

PANAGIOTA FATOUROU
FORTH & University of Crete
faturu@csd.uoc.gr

ELEFTHERIOS KOSMAS
University of Crete
ekosmas@csd.uoc.gr

KONSTANTINOS CHATZINIKOLAOU
University of Crete
csd4069@csd.uoc.gr

FORTH ICS TR 485 March 2023

Abstract

A priority queue is one of the most fundamental data structures (DS) with applications in many domains. Non-Uniform Memory Access (NUMA) has become a popular feature in modern architectures, adopted by many hardware vendors, such as Intel and AMD. Designing NUMA-aware concurrent data structures results in good performance and scalability in the number of threads, but it is a challenging task. In this work, we propose, study and analyze NUMA-aware concurrent priority queue implementations. The proposed algorithms employ software combining and an hierarchical design to address some of the challenges of developing NUMA-aware data structures.

March 2023

Table of Contents

1. Introduction

Non-Uniform Memory Access (NUMA) has become a popular feature in modern architectures, adopted by many hardware vendors, such as Intel and AMD. The NUMA architecture [Section 2.2] groups together the cores of the system into NUMA nodes/sockets. Using a number of different caches for every node (socket), the architecture makes the information exchange between the cores of the same group much faster than across groups. NUMA architectures, when programmed appropriately, have positive performance impact, due to the high scalability and low latency they achieve. Designing NUMA-aware concurrent data structures results in good scalability in the number of threads. In this work, we make a first step towards presenting NUMA-aware implementations of concurrent priority queues [Section 2.1]. A priority queue is one of the most fundamental data structures (DS) with applications in many domains, such as discrete event simulations, graph algorithms, and task scheduling.

Designing a concurrent data structure [Sections 2.2 and 3] is not an easy task, yet more so if the goal is to develop a NUMA-aware version of it. The main challenges when designing NUMA-aware concurrent priority queues are the following. 1) To exploit the locality benefits of a NUMA architecture, we have to use some local, to each socket (or even to each core) version of the priority queue. A thread executing at a specific core (or socket), inserts elements into its locally assigned queue [Section 4.1]. However, for DELETEMIN to be correct [Section 4.2], the algorithm's designer has to find a fast way to find the element of highest priority among all local queues, without destroying the locality-awareness benefits of employing multiple queues. We address this challenge by utilizing an hierarchical approach for accessing the queues. Our approach works by having a leader on each socket and a global coordinator thread. The rest of the threads are worker threads. Each worker thread has its own priority queue. Worker threads forward to their sockets' leaders, elements of highest priority from their local queues. Each leader stores the elements it receives from the workers of its socket to an intermediate-level data structure, and forwards those of highest priority to the coordinator. The coordinator serves all DELETEMIN requests and record their responses in appropriate arrays, so that they can be found by the threads which initiated them.

This simple scheme works well but have certain limitations. For instance, if a bunch of threads are not operating on the priority queue at some point in time [Section 2.2], there are no threads to forward upwards (to the appropriate leader) elements that reside on the local queue. Thus, some form of helping is required. This helping has to be carefully designed so that the benefits of the NUMA-aware technique is not outweighed by that cost. Finding and ensuring the right ratio of elements between the data structures of the different levels, is an additional challenge.

The proposed NUMA-aware algorithms employ both the technique of software combining [COMB] and an hierarchical design to address the above challenges. We have performed an initial performance evaluation which shows performance benefits of the proposed technique in the restricted cases where a) there exist only inserts in the queue, and b) all inserts occur before any DELETEMIN take place.

For the general case where INSERT and DELETEMINs happen concurrently, the performance is not as superior as we would like it to be. Thus, there is a need for extending and optimizing the proposed approach in order achieve the overall goal for the general case.

2. Definitions & Preliminaries

2.1. Priority Queue

Queues are one of the most fundamental data structures in computer science. FIFO queues operate based on the FIFO (First-In-First-Out) principle, meaning that, the first element inserted in the data structure should also be the first element to be removed. In a priority queue, on the other hand, each element additionally has a priority associated with it and the element to be removed from the priority queue should be the one with the highest priority.

Priority queues can be thought of as tree-based data structures that satisfy the *partial ordering property* (also referred as the *heap property* below): For every node N in the heap, N has higher priority than its children. If the tree is implemented using an array, then we call it heap. In a heap, we have assigned an id to each node of the tree. The root node has id 0. The left node of a node with id i has id [(2i+1)] and its right child has id [(2i+2)]. Thus, the parent of a node with id i has id [(i - 1) / 2]. The elements of the priority queue are placed at the positions of the heap array, as specified by their ids. Therefore, the element at position 0 represents the root of the tree, and the elements at positions 1 and 2 represent the root's left and right child. Then, for the element at position 1 (left child of the root), the element at position 3 represents its left child and the element at position 4 represents its right child, and so on.

Algorithms 1, 2 provide pseudocodes for two versions of INSERT (), and Algorithm 3 provides pseudocode for DELETEMIN, for a *sequential* implementation of a heap (i.e., a priority queue implemented using an array A).

Algorithm 1 provides pseudocode for the bottom-up version of INSERT. An INSERT operation inserts a new key as the last element of A (i.e., as the rightmost leaf of the bottommost level of the tree represented by the heap). Then, it traverses the tree upwards applying swaps among the key of the current node (which contains the newly inserted key) and that of its parent, if needed, to restore the heap property (i.e., the partial ordering of the heap). In more detail, array A contains the information stored in the nodes of the priority queue. Variable m stores an index to the element of the heap where the new key resides at each point in time. Initially, m stores the last position of A (line 4). Then, INSERT executes the loop of line 5 to restore the heap property (i.e., the partial ordering of the tree). If m is not the index of the root node (i.e., if the node indexed by m has a parent), and the key of the parent is larger than the variable key, then key must be moved upwards toward the root, until it reaches the correct position (line 9). The INSERT function takes as argument, the position of the node in the tree and returns the position of the parent node. If the node's position is i, then the position of the parent node is [(i - 1) / 2].

Algorithm 1 : Priority Queue: Insert

```
     Input: PQ_Heap *Heap, int key
     Output: Boolean result of insertion

1    if (Heap→size >= N) then //N is the heap capacity
2      return false;
3    end
4    int m = Heap→size;
5    while (m > 0 && key < Heap→A[PARENT(m)].Key) do
6      Heap→A[m].Key = Heap→A[PARENT(m)].Key;
7      m = PARENT(m);
8    End
9    Heap→A[m].Key = key;
10   Heap→size++;
```

There is also another implementation for the insert algorithm which starts the insertion from the root of the tree and follows a path downward until it reaches the correct node. This is the top-down version of INSERT, shown in Algorithm 2. Although the top-down version is not as efficient as the bottom-up version, and it will not be used to build our algorithms, it is presented because of its compatibility with the simultaneous synchronous (fine grain) DELETEMIN. The Bottom-Up INSERT function although is better overall, it cannot run with a simultaneous DELETEMIN function due to the opposite flow of the operations (Bottom-Up – Top-Down).
This algorithm first determines the path that should be followed to reach the correct position for the insertion of the new element (lines 9-17). After having stored the correct path using the "path" variable (line 6) the new element moves through that path until it reaches the desired node.

Algorithm 2 : Priority Queue: Insert [Top-Down]

```
     Input: PQ_Heap *Heap, int key
     Output: Boolean result of insertion

1    if (Heap→size >= N) then
2       return false;
3    end
4    int m = Heap->size;
5    int l = log2((m + 1));
6    int path[l];
7    int i = l - 1;
8    int p = m;
9    while (p > 0) do
```

```
10      if (p % 2) then
11          path[i] = LEFT;
12      else
13          path[i] = RIGHT;
14      i--;
15      p = PARENT(p);
16      end
17  end
18  i = 0;
19  int j = 0;
20  int t;
21  while (i < m) do
22      if (Heap→A[i].Key > key) then
23          t = key;
24          key = Heap→A[i].Key;
25          Heap→A[i].Key = t;
26      end
27      i = (path[j] == LEFT) ? LEFT_CHILD(i) : RIGHT_CHILD(i);
28      j++;
29  end
30  Heap→A[i].Key = key;
31  Heap→size++;
32  return true;
```

The DELETEMIN algorithm (Algorithm 3) deletes (and returns) the element with the highest priority, which is stored in the root of the tree. After deleting the root element, the algorithm restructures the tree to ensure that the heap priority is satisfied.

DELETEMIN first checks whether the heap is empty (line 1). If not, it extracts the highest priority element, i.e., the one that is stored at the root. If there are more nodes in the priority queue, the algorithm proceeds to examine whether the heap property is ensured. If this is not true, it performs a series of comparisons and swaps, moving the elements with the higher priority keys upwards (lines 11-23). The functions LEFT_CHILD and RIGHT_CHILD return the left and right children of the current node. Specifically, for node i, the LEFT_CHILD function returns (2 * i) + 1, whereas the RIGHT_CHILD function returns (2 * i) + 2.

Algorithm 3 : Priority Queue: DeleteMin

```
     Input: PQ_Heap *Heap
     Output: Highest Priority Element

1    if (Heap→size == 0) then
2        return false;
3    End
4    int key = Heap→A[size - 1].Key;
5    Heap→size--;
6    if (Heap→size == 0) then
7        return key;
8    End
9    int m = 0;
10   int p;
11   while ((LEFT_CHILD(m) < Heap→size && key > Heap→A[LEFT_CHILD(m)].Key) ||
                (RIGHT_CHILD(m) < Heap→size && key > Heap→A[RIGHT_CHILD(m)].Key))
     do
12       if (RIGHT_CHILD(m) < Heap→size) then
13           if (Heap→A[LEFT_CHILD(m)].Key < Heap→A[RIGHT_CHILD(m)].Key) then
14               p = LEFT_CHILD(m);
15           else
16               p = RIGHT_CHILD(m);
17           end
18       else
19           p = Heap→size - 1;
20       end
21       Heap→A[m].Key = Heap→A[p].Key;
22       m = p;
23   end
24   Heap→A[m].Key = key;
```

2.2. Concurrent programming – Threads

We consider a system where a number of threads run concurrently and asynchronously. Threads communicate by accessing shared objects. An example of a shared object is Compare&Swap (CAS), which gets three parameters: a pointer to an integer (p), an integer (old), and a second integer (new). CAS compares the contents of the memory location provided with the old value and if they are the same it changes the value of the memory location with the new value and returns TRUE. This is done in an atomic manner. If the contents of the memory location had been changed by another thread in

the meantime, then no change occurs and CAS returns FALSE. Another shared object of interest is Fetch-and-Or (FAO). It supports the FAO operation which uses OR to update the state of the object and returns the value that was previously stored there.

Every thread may have its local variables. Such a variable is not visible to the other threads. It may be global (to the thread), meaning that it is visible to all routines executed by the thread, or local to some part of the code (e.g., a function).

2.3. NUMA-aware computing

A NUMA architecture groups together the cores of the system into NUMA nodes/sockets. In a NUMA (Non-uniform memory access) machine, the main memory is shared between the sockets of the system, but every individual socket possesses its own caches, which can be level 1 (L1) or level 2 (L2). There is also a level 3 (L3) cache, which is shared among the threads of all of the sockets. Consistency among the different threads for the L3 cache is achieved with a cache-coherence protocol. The architecture makes the information exchange between the cores of the same group much faster than across groups.

2.4. Correctness

Linearizability is a well-studied correctness condition for concurrent data structures. It gives the illusion that an execution is sequential, i.e., that every operation takes effect instantaneously at some point in time between its invocation and its response.

2.5. Blocking Algorithms

As mentioned above, when threads are accessing shared variables and objects that are shared with other threads, synchronization is needed. The parts of the algorithm that must be executed in mutual exclusion are called critical sections. A critical section should satisfy the condition that it is executed by at most one thread at each point in time. To ensure this property, applications often use locks. The lock interface supports two operations. The first operation is the “lock” to acquire the lock while the other is the “unlock” to release it. That means that no thread can access the lock when another thread has acquired or “locked” the lock until it is released by that same thread. Algorithms that use locks are called blocking because if a thread delays to release a lock the rest threads can block without making any progress.

3. Concurrent Priority Queue Implementations

This section presents concurrent priority queue implementations. which use locks, as well as the lock-free queue implementation presented in [LIND]. This section aims at providing a basic understanding of some existing concurrent priority queue implementations, which are used as competitors to the NUMA-aware priority queue implementation we present in Section 4.

3.1. Coarse-grain Locking

One of the most straightforward ways to implement a concurrent priority queue in a blocking way, is by employing the technique of coarse-grain locking (CGL), i.e., by using a single lock to achieve synchronization. The resulting implementation is called BH_CGL. Algorithms 4 and 5 provide pseudocodes for BH_CGL. These pseudocodes closely follow those in Algorithms 1 and 3 (Section 2). Note that to prevent more than one thread to apply changes on the heap, the lock protects all elements of the heap. Specifically, HLock is used to ensure that only a single thread uses the heap while executing an INSERT or DELETEMIN operation. A thread starts executing its operation by acquiring the lock and releases it after all changes of the operation have been applied. Thus, all operations on the heap are serialized. Moreover, the algorithm may pay high synchronization cost in cases of contention.

Algorithm 4 : Blocked Heap – Coarse Grain Lock – Bottom-Up insertion [BH_CGL_Insert]

```
     Input: PQ_Heap *Heap, int key, lock HLock
     Output: Boolean result of insertion

1    Lock(HLock);
2    if (Heap→size = N) then
3      Unlock(HLock);
4      return false;
5    end
6    int m = Heap→size;
7    while (m > 0 && key < Heap→A[PARENT(m)].Key) do
8      Heap A[m].Key = Heap→A[PARENT(m)].Key;
9      m = PARENT(m);
10   end
11   Heap→A[m].Key = key;
12   Heap→size++;
13   Unlock(HLock);
14   return true;
```

Algorithm 5 : Blocked Heap – Coarse Grain Lock – Deletion [BH_CGL_DMin]

```
Input: PQ_Heap *Heap, lock HLock
Output: Highest priority key

1  Lock(HLock);
2  if (Heap→size == 0) then
3     Unlock(HLock);
4     return false;
5  end
6  int key = Heap→A[size - 1].Key;
7  Heap→size--;
8  if (Heap→size == 0) then
9     Unlock(HLock);
10    return key;
11 end
12 int m = 0;
13 int p;
14 while ((LEFT_CHILD(m) < Heap→size && key > Heap→A[LEFT_CHILD(m)].Key) ||
             (RIGHT_CHILD(m) < Heap→size && key > Heap→A[RIGHT_CHILD(m)].Key))
   do
15    if (RIGHT_CHILD(m) < Heap→size) then
16       if (Heap→A[LEFT_CHILD(m)].Key < Heap→A[RIGHT_CHILD(m)].Key) then
17             p = LEFT_CHILD(m);
18       else
19             p = RIGHT_CHILD(m);
20       end
21    else
22       p = Heap→size - 1;
23    end
24    Heap→A[m].Key = Heap→A[p].Key;
25    m = p;
26 end
27 Heap→A[m].Key = key;
28 Unlock(HLock);
29 return key;
```

3.2. Fine-grain Locking

In contrast to the coarse-grain locking method, the fine-grain locking method [SHAV] requires more than a single lock. Specifically, it uses a lock for each individual position of the heap. There are different ways to achieve synchronization during INSERT and DELETEMIN in this case.

Algorithms 5 and 6 employ a technique known as “lock coupling” or “hand-over-hand locking”. In this kind of fine-grain synchronization, an operation starts by locking one heap elements. As the thread traverses a path in the priority queue and while holding the lock of the previously traversed node, it acquires the lock of the current node, and then releases the lock of the previous node.

Algorithm 6 : Blocked Heap – Fine Grain Lock – Insert [BH_FGL_Insert]

```
Input: PQ_Heap *Heap, int key, lock HLock
Output: Boolean result of insertion

1  Lock(HLock);
2  if (Heap→size >= N) then
3        Unlock(HLock);
4        return false;
5  end
6  int m = Heap→size;
7  lock A(m).lock;
8  Heap→size++;
9  Unlock(HLock);
10 int prev;
11 while (m > 0) do
12       Lock(Heap→A(PARENT(m)).lock);
13       if (key < Heap→A[PARENT(m)].Key) then
14            Heap→A[m].Key = Heap→A[PARENT(m)].Key;
15            prev = m;
16            m = PARENT(m);
17            Unlock(Heap→A(prev).lock);
18       else
19            Unlock(Heap→A(PARENT(m)).lock);
20            break;
21       end
22 end
23 Heap→A[m].Key = key;
24 Unlock(Heap→A(m).lock);
25 return true;
```

Algorithm 7 : Blocked Heap – Fine Grain Lock – Deletion [BH_FGL_DMin]

```
Input: PQ_Heap *Heap, lock HLock
Output: Boolean result of insertion

1   Lock(HLock);
2   if (Heap→size == 0) then
3       Unlock(HLock);
4       return false;
5   end
6   int key = Heap→A[size - 1].Key;
7   Heap→size--;
8   if (Heap→size == 0) then
9       Unlock(HLock);
10      return key;
11  end
12  int m = 0;
13  Lock(Heap→A(m).lock);
14  Unlock(HLock);
15  Lock(Heap→A(LEFT_CHILD(m)).lock);
16  Lock(Heap→A(RIGHT_CHILD (m)).lock);
17  int p, c;
18  while ((LEFT_CHILD(m) < Heap→ size && key > Heap→ A[LEFT_CHILD(m)].Key) ||
    (RIGHT_CHILD(m) < Heap→size && key > Heap→A[RIGHT_CHILD(m)].Key)) do
19      if (RIGHT_CHILD(m) < Heap→ size) then
20          if (Heap→A[LEFT_CHILD(m)].Key < Heap→A[RIGHT_CHILD(m)].Key) then
21              p = LEFT_CHILD(m);
22              c = RIGHT_CHILD(m);
23          else
24              p = RIGHT_CHILD(m);
25              c = LEFT_CHILD(m);
26          end
27      else
28          p = Heap→size - 1;
29          c = RIGHT_CHILD(m);
30      Unlock(Heap→ A(c).lock);
31      end
32      Heap→A[m].Key = Heap→A[p].Key;
```

```
33      Unlock(Heap→A(m).lock);
34      m = p;
35      Lock(Heap→A(LEFT_CHILD(m)).lock);
36      Lock(Heap→A(RIGHT_CHILD (m)).lock);
37  end
38  Heap→A[m].Key = key;
39  Unlock(Heap→A(RIGHT_CHILD(m)).lock);
40  Unlock(Heap→A(LEFT_CHILD (m)).lock);
41  Unlock(Heap→A(m).lock);
42  return key;
```

Both algorithms use locks in a similar manner. The only difference is that the INSERT function follows a single, fixed path to the root since it starts from the bottom of the tree. The DELETEMIN function starts from the root, so it must choose between the children of each node to proceed. To examine both children in order to make this choice, DELETEMIN has to lock both children of the currently locked node as it traverses down a path of the tree.

Roughly speaking, these functions follow the same pattern for achieving synchronization: As each thread traverses the tree, it locks the first node it encounters. Then, it locks its parent or its children (INSERT or DELETE, respectively). While simultaneously holding these locks, we can perform the swaps as we go.

In the INSERT function, we can see the described pattern at lines 6 and 7, where a thread locks the last node of the tree (starting point), and before proceeding to make the necessary change (line 14), it locks the next in line node at line 12.

The same is true for the DELETEMIN function. In lines 12 and 13, a thread locks the root of the tree and then both of the children nodes (lines 15 and 16). After finding out the correct path and releasing the child that is not in that path (lines 18-39), the thread can make the necessary changes. At that moment the thread is keeping two locks before releasing the first one and starting the loop again.

When INSERTS and DELETEMINs happen concurrently, it is crucial for a thread to acquire locks in the same order in order to avoid deadlocks (as we do in the above functions).

3.3. The Linden and Jonsson Priority Queue Implementation

The Linden and Jonsson (LJ) algorithm [LIND] is an existing lock-free implementation which implements the priority queue as a skip-list [PUGH]. In this section, we will use the pseudocodes for the LJ algorithm in a similar manner as those in the original paper referenced above.

LJ employs the Compare&Swap (CAS) atomic instruction, which is supported by all modern architectures, to avoid using locking. LJ exhibits good performance. So, we use it, together with the

other two algorithms presented above, as the main competitor in the experimental analysis we present in Section 5.

A node to be deleted, is first logically deleted and then the physical deletion takes place. The physical deletion disconnects the node from the data structure. On the contrary, after a logical deletion, the deleted node is still linked in the skip list. It is however marked for deletion (i.e., a bit is set that indicates that the node is deleted). Physical deletions are made in batches, i.e., more than one node is deleted with a single update of the skip-list pointers. This approach counterbalances the latency created by the larger number of read operations that will be needed when traversing a skip-list containing logically deleted nodes. Logical deletions have been introduced in [LHAR] and have been used extensively to implement deletes in many concurrent data structures.
The nodes that are initialized with the AllocNode function are structures that contain all the information needed. They contain the value of the node, a key, and an array of next pointers (a pointer for every level of the skip list). The node also contains a boolean d value that describes the logical delete state of that node. There is also the boolean value inserting that is set to false when the insertion of the node is completed.

The insertion is performed in a way similar to an ordinary skip-list insertion. However, it takes into consideration the logically deleted nodes (thus, the delete flags of nodes). A node is initialized in lines 1 and 2. Then there is a search for the predecessor nodes at each level, to decide where the node will be inserted, using the function LocatePreds, whose pseudocode is shown below. Once these candidates have been found the new node is inserted in the lowest level of the list. This is done with the CAS of line 6. Then the new node is moving upwards level by level. First, the next pointer of the node is set (line 9) and from there it starts advancing to the next level. Then it checks if the new node has been deleted in the meantime (lines 10 and 17). If the node has been deleted then the algorithm returns, if not it continues to move to higher levels until it is inserted (line 20). After the insertion of that node, the batch deletion is allowed.

	Algorithm 8 : Linden - Insertion
	Input : skiplist_t q, key_t k, value_t v
1	height = Random(1, q.nlevels), new = AllocNode(height);
2	new.key = k, new.value = v, new.d = 0, new.inserting = 1;
3	repeat
4	(preds, succs, del) == LocatePreds(q, k);
5	new.next[0] == succs[0];
6	until (Compare&Swap(&<preds[0].next[0], preds[0].d>, <succs[0], 0>, <new , 0>)
7	i = 1;
8	while (i < height) do
9	new.next[i] == succs[i];
10	if (new.d == del || succs[i].d == del || succs[i] == del) then

```
11          break;
12       end
13       if  (Compare&Swap (&preds[i].next[i], succs[i], new)) then
14          i = i + 1;
15       else
16          (preds, succs,del) = LocatePreds(q, k);
17          if (succs[0] != new) then break;
18       end
19    end
20    new.inserting = 0;
```

The LocatePreds function is responsible for finding the successors and predecessors of a node with key k inside a skiplist q.

The algorithm traverses the skiplist from the higher level and searches for a key greater than k in every list from there. That node either hasn't set its delete flag and belongs to a level greater than zero, or it is a node that hasn't been deleted yet and it's on the lowest level (lines 5-10). From that node, the successor and predecessor are stored in the return values. With these two values, we also return the del variable, if there is one, which is the last deleted node of the deleted prefix as found in line 6.

Algorithm 9: Linden - LocatePreds

```
   Input : skiplist_t q, key_t k, value_t k

1  i = q.nlevels–1,pred = q.head;
2  del = NULL;
3  while (i ≥ 0) do
4       <cur , d> = <pred .next[i], pred .d>;
5       while ((cur.key < k) || cur.d || (d && i == 0)) do
6         if (d && i == 0) then
7           del = cur;
8         end
9         pred = cur;
10        <cur , d> = <pred .next[i], pred .d>;
11      end
12      preds[i] = pred;
13      succs[i] = cur;
```

```
14        i = i - 1;
15    end
16    return (preds, succs, del);
```

In lines 2-13, we see the DELETEMIN function traverses the skiplist from the lowest level by moving through the next pointers of the node in line 3, and in line 4 it checks if the node is pointing to the tail (dummy) node and if so, it returns the function. Then it continues until it reaches the first node that does not have its delete flag set (skipping all the nodes that have been logically deleted but are still in the system). If it is not the tail node, then we set the flag as seen in line 10, using the atomic Fetch-and-Or (FAO) instruction.

After that traversal, the DELETEMIN examines if the logically deleted nodes have surpassed a specific upper bound threshold (line 14, BoundOffset), before physically deleting all the nodes that were previously logically deleted. If the deletion is successful then the operation must proceed to update the higher-level pointers, as we see in line 17 using the Restructure function that is referenced in Algorithm12. In the remaining lines, after the restructure function has finished, the DELETEMIN operation proceeds to mark the nodes between the obshead node (observed first node), and the newhead node, to be prepared for recycling (lines 20-24).

```
      Algorithm 10 : Linden_DeleteMin
      Input : skiplist_t L
      Output : Deleted value v
      
1     x = L.head, offset = 0, newhead = Null, obshead = x.next[0]
2     repeat
3         <nxt, d> = <x.next[0], x.d>;
4         if (nxt == q.tail) then
5             return Empty;
6         end
7         if (x.inserting && newhead = Null) then
8             newhead = x;
9         end
10        <nxt, d> = FAO(&<x.next[0], x.d>, 1);
11        offset = offset +1;
12        x = nxt;
13    until not d
14    v = x.value;
```

```
15  if (offset < BoundOffset) then return v;
16  if (newhead == Null) then newhead = x;
17  if (Compare&Swap(&<q.head.next[0], q.head.d>, <obshead , 1>, <newhead , 1>))
    then
18      Restructure(q);
19      cur = obshead;
20      while (cur != newhead) do
21          nxt = cur.next[0];
22          MarkRecycle(cur);
23          cur = nxt;
24      end
25  return v;
```

Starting from the highest level and omitting level 0, the RESTRUCTURE operation updates the lists appropriately after a successful physical DELETE operation. At every level, we record the state of the head's pointer, as seen in line 4, and it is updated only if the node pointed to has been deleted. After that, we traverse that level until running into the first node that does not have a successor that has it's delete flag set, stored in variable pred, as we can see in line 11.

Algorithm 11: Linden - Restructure

```
    Input : Skiplist_t q
1   i = q.nlevels –1, pred = q.head;
2   while (i > 0) do
3        h = q.head.next[i];
4        cur = pred .next[i];
5        if (!h.d) then
6          i = i – 1;
7          continue;
8        end
9        while (cur.d) do
10         pred = cur;
11         cur = pred.next[i];
12       end
13       if  (Compare&Swap (&q.head.next[i], h, pred.next[i])) then
14         i = i – 1;
15       end
16  end
```

4. NUMA-aware Concurrent Priority Queue Implementations

The algorithms we have seen so far give us a glimpse of how priority queues are implemented and how we can easily extend them into a concurrent setting. In this section we will focus on NUMA-aware approaches and how to apply locality awareness techniques to get more efficient algorithms. We start by first explaining our NUMA-aware techniques on simplified scenarios, where threads may perform only inserts or only deletes. Such scenarios are describing cases where all the information must be stored first in the structure without the need for any deletion operations in the meantime. After the insert operations have occurred, the delete operations can be executed without any intervention from inserts.

4.1. Inserts Only

Instead of using a single, global heap for all threads, every thread should have its own, local heap. In that way INSERTS become embarrassingly parallel: every thread inserts at its heap, independently from the other threads. If all threads are active and operate on the priority queues, we expect an almost linear throughput as we increase the number of threads, due to the locality nature of the algorithm. The algorithm is simple· every thread inserts its element to their local heap using the INSERT function of Algorithm 1.

4.2. Deletes Only

As a consequence of the INSERT's simplicity, we need a more carefully designed solution for the DELETEMIN function in order to achieve linearizability. This tends to be a challenging task for the following reason: the DELETEMIN operation must return with the greatest key in the system. However, the thread that has issued the DELETEMIN operation does not know in which of the local queues this key resides. A thread can identify the key with the greatest priority in its heap but there must be a NUMA-sensitive way to compare all the greatest keys from all the local heaps of the threads. The algorithms that we use for the DELETEMIN function follow an hierarchical pattern which look like a pyramid. At the bottom of that pyramid are the workers, whose job is to promote the elements with the highest priority from their local queues to the middle level threads. At the middle level, we have as many leader threads as the number of sockets in the system, one leader for each socket. Finally, on the top of the pyramid, we have a single thread which plays the role of the coordinator. The coordinator performs the actual deletion and returns the results to the workers. The coordinator performs DELETEMIN operations in batches. Algorithm 12 presents the pseudocode for this version of the algorithm while the roles of the threads are explained in detail in sections 4.2.1, 4.2.2 and 4.2.3.

Algorithm 12: Hierarchical Delete

```
1  announce[id][0] = DELETEMIN_REQUEST;
2  while (true) do
3      long lock_value = *leader_lock;
4      if (lock_value % 2 == 0) then   // attempt to promote worker to leader
5          if (Compare&Swap(leader_locks[group], lock_value, lock_value + 1)) then
6              break;
7          end
8      else
9          while (*leader_locks[group] == lock_value) do   // remain worker
10             Worker(group, id);   // see Algorithm 13
11         end
12         if (announce[id][0] == APPLIED) then   // my request has been applied
13             int returned_value = announce[id][1];
14             return;
15         end
16     end
17 end
18 while (true) do   // already leader, attempt to promote to coordinator
19     long lock_value = *coordinator_lock;
20     if (lock_value % 2 == 0) then   // attempt to be promoted to coordinator
21         if (Compare&Swap(coordinator_lock, lock_value, lock_value + 1)) then
22             break;
23         end
24     else
25         while (*coordinator_lock == lock_value) do
26             Leader(group); // see Algorithm 14
27         end
28     end
29 end
30 Coordinator();   // see Algorithm 15
31 (*coordinator_lock) ++;
32 (*leader_locks[group]) ++;
33 Memory_barrier;
```

By running the code in Algorithm 12, every thread becomes either a worker, or a leader, or the coordinator (as seen in lines 10, 26 and 30 respectively). The code uses as many locks as the number of sockets to choose one leader for each socket. It also uses an additional lock to choose a coordinator among the leaders. The code also uses two instances of the locking mechanism (lines 2-17 and 18-29). Each lock is implemented using an integer. A thread acquires a lock by increasing the value of the lock by one and does the same to release the lock. An odd value in the lock variable means that the lock is acquired, whereas an even value implies that the lock is free. A thread that acquires the lock increases the value of the integer twice, once to acquire the lock (lines 5 and 21) and then to release it (lines 31 and 32). If we take a closer look at the locking code, we will notice that the threads are not blocked by the lock. They try to acquire the lock, and if they fail to do so, they move on to execute the else part of the code (lines 8-16 and 24-28), where they are assigned to their appropriate roles (worker or leader).

Algorithm 12 uses several data structures. The announce array is a two-dimensional array with thread_number rows of two columns each, thus each thread has two elements in this array, one for announcing information to the other threads and one for receiving information from them. Specifically, if a thread has an identifier id, then announce[id][0] is used by the thread with identifier id to announce its active DELETEMIN operation and announce[id][1] is used to store the response to this operation.

Recall that threads that belong to the same socket can communicate faster than threads which execute in different sockets. We take advantage of this knowledge to separate the threads into groups. The groups are created so that threads of the same socket belong to the same group. There are as many instances of Algorithm 12 running in the system as the number of groups. Most of the variables used in Algorithm 12, are accessed by the threads of a single group. However, there exists a single coordinator lock used by the leader threads of all groups. The algorithms that are presented in later sections refer to a single group. The codes for the worker, the leader and the coordinator routines, are presented and discussed in sections 4.2.1, 4.2.2 and 4.2.3, respectively.

Let N_G be the number of groups in the system. There are N_G number of leader_lock instances and also N_G number of announce_array instances in the system, one for each group. The coordinator_lock is unique for all groups because it is related to the coordinator role, which is unique. From now on, all the codes and the algorithms that will be presented in later secions, will describe what happens to each group independently, except when explicitly mentioned otherwise.

4.2.1. Workers

In this section, we present and discuss the worker code which is illustrated in Algorithm13.

Every thread is by default a worker. However, recall that before a thread begins to act as a worker, it tries to acquire the appropriate leader lock, in an effort to become the leader of their group. If it fails

to do so, it serves as a worker until the leader lock is released or until it discovers that its request has been served.

Algorithm 13: Worker of socket group, with identifier id in this socket

Input: int group, int id, struct leader_array //stores highest priority elements from the heaps**, struct fill_next_array** //reminder to keep track of the next available position in the buffers

```
1   while (true) do
2       long lock_value = *(cluster_heaps[group, id]->lock);
3       if (lock_value % 2 == 0) then
4           if (Compare&Swap (cluster_heaps[group, id]->lock, lock_value, lock_value +
            1)) then //if the heap is not being used by another thread
5               leader_array[group][id].buffer[fill_next_array[group][id].value]
                =  deleteMin(cluster_heaps[group, id]); //move the highest key to leader
                array
6               (*cluster_heaps[group, id]->lock) ++;
7               fill_next_array[group][id].value = ((fill_next_array[group][id].value +
                1) % buffer_size);  //update position for the next available key there
8               return;
9           end
10      end
11      return;
12  end
```

The worker code (Algorithm 13) has two input parameters: group and id, identifying the group that the thread belongs to and the thread's identifier, respectively. Note that a thread may have to help another worker thread to complete its request which may not reside in the same socket. This is to ensure progress. That may happen in several cases. A thread may be inactive or busy doing another task, which means that its local heap will not be operated on, or it could be slower than the other threads. In those cases, a worker can operate on a different queue to ensure that the operation will make progress. Because of this helping, a lock mechanism (lines 2-4) is needed to ensure that only one thread is operating on the heap at each point in time. The cluster_heaps variable stores all the heaps of the system so that the threads can retrieve the one they need via the group and id variables. The pair [group, id] is often used as an identifier to refer to specific threads or the heaps they own.

In line 5 we see two arrays: the leader_array and the fill_next_array. The leader_array stores the elements with the priority from each heap. Specifically, each thread has a buffer in this array where

it stores the higher priority elements from its local queue. Every worker provides elements to the leader_array constantly from its local queue. These keys are derived by executing the DELETEMIN operation on the local heap. The next empty slot in this buffer is indicated by fill_next_array[group][id]. After inserting an element in the appropriate position of the leader array, the thread increases the value of fill_next_array[group][id] to indicate the next position in its buffer to store a newly coming element. fill_next_array[group][id] contains indices to the elements of the buffer that are used in a circular way (line 7).

In line 6 the thread releases the lock, and then in line 7, it updates the buffer of the fill_next_array for later use.

4.2.2. Leaders

In this section we present and discuss the leader code which is illustrated in Algorithm 14.

Algorithm 14: Leader

```
Input: int group, struct leader_array, struct coordinator_array //stores the highest priority elements from the heaps, fill_next_array, top_fill_next_array //reminder to keep track of the next available position in the coordinator_array

1  while (true) do
2      long lock_value = *leader_locks[group];
3      if (lock_value % 2 == 0) then
4          if (Compare&Swap (leader_locks[group], lock_value, lock_value + 1)) then //if the heap is not being used by another thread
5              if (coordinator_array[group].buffer[top_fill_next_array[group]] == EMPTY) then //if the leader hasn't provided any elements yet
6                  int i, pos = 0, min = MAXINT;
7                  for (i = 0; i < socket_size[group]; i++) do //examines all the threads in the socket
8                      while (leader_array[group][i].buffer[fill_next_array[group][i].value] == EMPTY) do //if the workers haven't provided any elements yet
9                          Worker(group, i); //become a worker for that socket
10                     end
11                     if (leader_array[group][i].buffer[fill_next_array[group][i].value] < min) then
12                         min = leader_array[group][i].buffer[fill_next_array[group][i].value];
13                         pos = i;
14                     end //if it has the key with the higher priority so far, keep it
```

```
15                                  end
16                                  leader_array[group][pos].buffer[fill_next_array[group][pos].value] = EMPTY;
17                                  fill_next_array[group][pos].value = ((fill_next_array[group][pos].value + 1) % buffer_size);
18                                  coordinator_array [group].buffer[*top_fill_next_array[group]] = min;
19                                  *top_fill_next_array[group] = ((*top_fill_next_array[group] + 1) % buffer_size_top);
20                              end //updated the positions on the arrays for future use
21                              (*leader_locks[group]) ++;
22                              return;
23                          end
24                      end
25                      return;
26                  end
```

It was discussed in the previous section that a worker could operate for another heap to help the operation advance. Here the leaders follow the same approach. This helping by the leaders is needed if e.g., none of the threads of a socket is active, so the socket does not have a leader. In that case a leader from another group can serve as the leader of the inactive socket. The input parameter group is used, so that the thread knows which group to assist as leader.

The leader code starts by attempting to obtain the leader lock (lines 2-4). The leader_locks array contains all the leader locks, and the leader can choose which to acquire by utilizing the group variable.

In line 6, the leader examines all the threads of the socket to make sure that every worker has already sent its highest priority element from its heap, then the leader compares this element with those sent by other threads of the socket in order to submit to the coordinator the key with the highest priority from the entire socket. If there is a worker that has not yet sent an element (line 8) then, instead of waiting and thus halting the program, the leader tries to become a worker for the thread that owns the heap with the missing element (line 9). After leader_array has been filled in with the missing element, the leader compares it with the rest elements to see if it has a greater priority (line 11). After the leader has examined all the elements, it sends to the coordinator the one with the highest priority and then updates the arrays and the buffers (lines 16-19).

During this phase, we see two new arrays: the coordinator_array and the top_fill_next_array. These two arrays are used similarly to the leader_array and the fill_next_array in the worker code. However, they are utilized for the communication of the leaders with the coordinator. The coordinator_array and the top_fill_next_array have as many indices as the total number of sockets in the system.

4.2.3. Coordinator

In this section we present the coordinator code shown in algorithms 15,16 and 17.

There exists a single coordinator in the entire system at each point in time. The coordinator palys one of the leader threads, specifically it is the leader thread that manages to acquire the coordinator lock. The coordinator employs a combining technique [COMB], to serve DELETEMIN operations and returns the elements with the highest (overall) priorities back to the initial workers who initiated the DELETEMIN operations. Specifically, the coordinator serves all the threads of its own group that have made a DELETEMIN request.

Algorithm 15: Coordinator

```
1   int i;
2   for (i = 0; i < socket_size; i++) do   // find active requests from my socket
3      if (announce[i][0] == DELETEMIN_REQUEST)
4         int returned_value = get_min_element(); //find the element with the
          highest priority
5         announce[i][1] = returned_value;  //return value to worker
6         announce[i][0] = APPLIED;   //inform the worker of the completion of the
          request
7      end
8   end
```

We now present the main routine of the coordinator algorithm (Algorithm 15). In lines 2, the coordinator examines all the threads that belong to its group to figure out whether there exist active DELETEMIN operations initiated by them. Then for each one of them, it writes back the highest priority element to return (line 5). To find this element, the coordinator calls the get_min_element function and saves the value it returns into the announce array. Afterwards, it sets announce[i][0] to APPLIED, to indicate that the request has been served.

Algorithm 16: get_min_element

```
1   int i;
2   int pos = 0;
3   int min = MAXINT;
4   for (i = 0; i < socket_number; i++) do
5      while (coordinator_array[i].buffer[top_fill_next_array[i]] == EMPTY) do
6         Leader(i);
7      end
8      if (coordinator_array[i].buffer[top_fill_next_array[i]] < min) then
9         min = coordinator_array[i].buffer[top_fill_next_array[i]];
10        pos = i;
11     end
```

```
12  end
13  int value = coordinator_array[pos].buffer[top_fill_next_array[pos]];
14  coordinator_array[pos].buffer[top_fill_next_array[pos]] = EMPTY;
15  top_fill_next_array[pos] = ((top_fill_next_array[pos] + 1) % buffer_size);
16  return value;
```

In Algorithm 16, the coordinator must examine the coordinator_array to choose the highest priority element overall, among the highest priority elements of the groups. In line 4, the coordinator searches all the groups one by one. In case some groups have not yet provided their highest priority element (line 5), the coordinator cannot continue until it gets informed about this element by the appropriate leaders. Note that one of these leaders may be the coordinator itself. In this case, instead of waiting, the coordinator must help the leader by serving as the leader itself, as seen in line 6. Note that the coordinator may end up to act as a leader and even as a worker in its group in addition to its role as a coordinator.

In lines 13-15, the coordinator updates the values of the coordinator_array and top_fill_next_array. The top_fill_next_array works in a way similar as the fill_next_array, but for the top level of the hierarchy.

4.3. Simultaneous Insert-DeleteMin

In this section, we develop algorithms that support concurrent INSERT and DELETEMIN operations. In the implementations presented in the previous sections, arrays are used extensively because they are fast to traverse. The use of arrays works well in restricted cases were all INSERT operations are completed before executing any DELETEMIN operations. However, in the case of concurrent INSERTS and DELETEMINs, the arrays should be appropriately updated when new elements are inserted to ensure linearizability. Updating the arrays is often expensive because the arrays are sorted, and it would require array elements to be shifted in order to make room for the newly inserted elements. A good alternative is to use a dynamic data structure, such as a linked list.

The code for concurrent INSERT and DELETEMIN operations are listed in Algorithms 17 and 18.

Algorithm 17: HeapInsert - Simultaneous Insert-DeleteMin

***I**nput:* PQ_Heap *Heap

Output: *Boolean result of insertion*

```
1  int m = (Heap->Size);
2  if ((m && key < Heap->PQ[0].Key)) then
3      return false; //high priority elements will be saved on the higher levels,
       not here
4  End
5  while (m > 0 && key < (Heap->PQ)[PARENT(m)].Key) do
```

```
6       (Heap->PQ)[m].Key = (Heap->PQ)[PARENT(m)].Key;
7       m = PARENT(m);
8   End
9   (Heap->PQ)[m].Key = key;
10  (Heap->Size)++;
11  return true;
```

Algorithm 17 is the same as the sequential INSERT algorithm we saw in Algorithm 1, but with a small adjustment in lines 2-4. Specifically, if the element to be inserted has higher priority in comparison to all the other elements in its local queue, then the function returns false. This is desirable because in that case the element should be inserted on some higher-level data list instead.

```
Algorithm 18: HierInsertLocal
Input : int key, int group, int id
1   while (true) do
2       long lock_value = *(cluster_heaps[group, id]->lock);
3       if (lock_value % 2 == 0) then
4           if (Compare&Swap(cluster_heaps[group, id]->lock, lock_value, lock_value
            + 1)) then //aquire the lock to assure this thread is the only one
            operating here
5               Int key2 = key;
6               if (HeapInsert(cluster_heaps[group, id], key)) then //the element is
                stored locally but it might need to move upwards
7                   if (!Entries[group][id] || key < GPELEM[group, id]) then //if there
                    are no elements from that thread in the lists or the key has a very
                    high priority, then remove from local heap and place the element
                    upwards
8                       key2 = deleteMin(cluster_heaps[group, id]);   //key2 may be
                        different than key, in case the first condition of line 7 is
                        true
9                   else
10                      (*cluster_heaps[group, id]->lock) ++;
11                      return;
12                  end
13              end
14              MoveElementUp(key2, group, id); //this code is executed when the
                element hasn't been saved locally
15              (*cluster_heaps[group, id]->lock) ++;
```

```
16                 return;
17             end
18         end
19     end
```

Algorithm 18 provides the code that the threads invoke to perform an INSERT operation. A lock is used to synchronize access to a local queue as helpers may exist. In line 6 we see that the return value of the HeapInsert function is used to determine if this algorithm should continue or not. If the element is not saved locally, then the algorithm proceeds with the MoveElementUp algorithm to store the element to the list maintained at the middle level for the socket on which the thread that has initiated this HeapInsert belongs to. If the element is indeed saved in a local heap, then the algorithm checks if the specific thread has at least one element in the list above or if the element has the largest priority in that list (line 7). Since operations are faster when done locally, we prefer having more elements to the local heaps than on the lists of higher levels.

In line 7, Entries is an array that holds the information of how many elements each thread has in the lists and the GPELEM determines the highest priority element that is saved in all these lists. By checking these conditions (line 7), we try to balance the percentage of elements that are saved locally and on the higher level lists. If the conditions of line 7 are fulfilled, then the element to be inserted is deleted from the local heap where it was already stored and the algorithm proceeds by executing the MoveElementUp function, which will store this element at the higher-level list.

We proceed to present Algorithm 19. The leader_lists array contains as many elements as the groups in the system. Each of these elements is a pointer to the list corresponding to this group. In these lists the threads of the group can store their highest priority keys at each point in time. Similarly, the coordinator_lists array contains pointers to as many lists as the number of groups in the system. Note that in previous implementations the coordinator array was a one-dimensional array storing the higher priority keys from the local data structures from all the groups. Now the coordinator will have to choose the highest priority element from the lists of the coordinator_lists. Note that during INSERT, depending on the priority of the inserted elements, a thread can insert an element either locally or at the medium level, or even at the top level data structure. In line 2, the element's priority is compared to the highest priority element of the second level list and if the highest priority element has a smaller priority, then the thread proceeds to insert the new element at the coordinator level list as seen in line 3.

Algorithm 19: MoveElementUp

Input : int key, int group, int id

```
1  lock(leader_lists[group]->lock);
2  if(key < leader_lists[group]->key && leader_lists[group]->key != EMPTY) then
   //if the element has the highest priority key in the leader list, it moves up
   to the coordinator list
3      ListInsert(coordinator_lists[group], key, group, id); //coordinator has as
       many lists as the sockets of the system
4      unlock(leader_lists[group]->lock);
5  return;
6  ListInsert(leader_lists[group], key, group, id);  //basic case scenario, store
   to leader list
7  unlock(leader_lists[group]->lock);
```

The DELETEMIN follows the same hierarchical pattern as it did in the previous section. This is why Algorithm 12 (Hierarchical DeleteMin) is used as it is for the DELETEMIN operation here as well. It is necessary though to make changes to the worker, leader and coordinator code to take into consideration that simultaneous INSERT and DELETEMIN operations are now supported.

Algorithm 20: Worker - Simultaneous Insert-DeleteMin

Input : int group, int id

```
1  if(!Entries[group][id]) then //should fill the list with elements if empty
2      int key;
3      while(true) do
4          long lock_value = *(cluster_heaps[group, id]->lock);
5          if (lock_value % 2 == 0) then
6              if (Compare&Swap (cluster_heaps[group, id]->lock, lock_value,
               lock_value + 1)) then //the lock is needed to make sure only one thread
               operates here
7                  key = deleteMin(cluster_heaps[group, id]); //remove highest
                   priority key from heap to move upwards
8                  break;
9              end
10         end
11     end
```

```
12      MoveElementUp(key, group, id);
13      (*cluster_heaps[group, id]->lock) ++;
14  end
```

The worker function executes only when there are no entries of its local heap into the shared lists of higher levels (line 1). A worker deletes elements from its local heap and calls the MoveElementUp function that was discussed earlier. As mentioned above, inside its local MoveElementUp function a thread can insert in the medium-level list, or at the top one depending on the priority of the inserted element.

Algorithm 21: Leader - Simultaneous Insert-DeleteMin

Input : int group

```
1   int curr_t;
2   for(curr_t = 0; curr_t < socket_size[group]; curr_t++) then
3       Worker(group, curr_t); //the leaders act as assistans to the workers of their group
4   end
```

Now the leader of each group has the additional role to support their group by helping the workers within its group. It accomplishes this task by executing the worker code for all the other threads inside its socket, if needed.

The coordinator code will remain the same, as it was in the previous section (Algorithm 15). The only difference is that the get_min_element() routine (Algorithm 16) must change to adapt to the data structure and algorithmic changes that have occurred in the new implementation. So, Algorithm 15 will instead use the get_min_element() implementation shown below (Algorithm 22).

Algorithm 22: get_min_element - Simultaneous Insert-DeleteMin

```
1   int group, min = INT_MAX, pos = EMPTY;
2   for(group = 0; group < socket_number; group++) then
3       Leader(group); //make sure the leader lists are filled with the elements needed
4       if(coordinator_lists[group]->key == EMPTY) then //fill the coordinator lists if empty
5           Node d_key = ListDeleteMin(leader_lists[group]);
6           ListInsert(coordinator_lists[group], d_key.key, d_key.group, d_key.id);
7       end
```

```
8        lock(coordinator_lists[group]->lock);
9        int key = coordinator_lists[group]->key;
10       if(key < min) then
11           min = key;
12           pos = group;
13       end
14   end
15   for(group = 0; group < socket_number; group++) do //unlock all coordinator lists
16       unlock(coordinator_lists[group]->lock);
17   end
18   ListDeleteMin(coordinator_lists[pos]);
19   return min;
```

The coordinator (Algorithm 22) must examine all the lists of the top level. To do so, the coordinator must make sure that the leader lists are filled with at least one element each (line 3). Thus, if there are lists from the coordinator_lists that are empty, the leader_list will have elements to provide (lines 4-7). After that the coordinator examines the element with the highest priority key from every socket and compares each of them with the others (lines 10-12). After finding the socket with the highest priority element in its list, the coordinator releases the locks from all the coordinator_lists (lines 15-17), and then returns the element.

Algorithm 23: ListInsert

Input : Node *head, int Key, int group, int id

```
1    Node *pred, *curr;
2    bool return_flag = 0;
3    while (true) do
4        pred = head; curr = pred->next;
5        while (curr->key < key && curr->next) do
6            pred = curr;
7            curr = curr->next;
8        end
9        lock(pred->lock); lock(curr->lock);
10       if (Validate(pred, curr) == true) then
```

```
11          Node *node = NEWCELL();
12          node->next = curr;
13          node->key = key;
14          node->group = group;
15          node->id = id;
16          pred->next = node;
17          return_flag = 1;
18      end
19      unlock(pred->lock); unlock(curr->lock);
20      if (return_flag) then return;
21  end
```

Algorithm 24: ListDeleteMin

Input : Node *head

Output : Deleted Node

```
1   Node *pred, *curr;
2   bool return_flag = 0;
3   while (true) do
4       pred = head; curr = pred->next;
5       lock(pred->lock); lock(curr->lock);
6       if (Validate(pred, curr)) then
7           Curr->marked = true;
8           pred->next = curr->next;
9           result.key = curr->key;
10          result.group = curr->group;
11          result.id = curr->id;
12          return_flag = 1;
13      end
14      unlock(pred->lock); unlock(curr->lock);
15      if (return_flag == 1) return result;
16  end
```

Algorithm 25: Validate	
Input : Node *pred, Node *curr **Output : Boolean value for success**	
1	`if (pred->marked == false && curr->marked === false && pred->next == curr) then`
2	`    return true;`
3	`else`
4	`    return false;`
5	`end`

The INSERT and DELETEMIN algorithms on the lists use lazy synchronization. Every node has an additional boolean field, called marked, which indicates if a node is deleted from the list but still exists in it physically. By using this technique, we can avoid heavy synchronization techniques, such as hand-over-hand locking, and we lock only the locks that are affected by the executed operation on the list. For instance, the INSERT function locks the target's predecessor and adds the new node. The DELETEMIN function is split into two parts. First, it marks the node to be deleted as deleted and then proceeds to delete the node physically. To ensure linearizability, the Validation function is needed to make sure that nothing has changed in the critical neighborhood of the list while acquiring the locks.

5. Experimental Analysis

5.1. Setup

We used a 40-core machine equipped with 4 Intel(R) Xeon(R) E5-4610 v3 1.7Ghz CPUs with 10 cores each, and 25MB L3 cache. The machine runs CentOS Linux 7.9.2009 with kernel version 3.10.0- 1160.45.1.el7.x86_64 and has 256GB of RAM. Code is written in C and compiled using the gcc compiler (version 11.2.1) with O2 optimizations.

5.2. Analyzed Algorithms

An experimental analysis, presented in LIND, showed that the Linden algorithm (which was discussed in Section 3.3), has better performance than many other priority queue implementations, especially when the number of threads is increasing. In that paper, the performance of LIND is compared to the performance of the following algorithms: Fraser's SkipList [SPRL], an implementation proposed by Sundell and Tsigas, called NOBLE [NOBL], and a heap implementation by Hunt et al. [HEAP]. For that reason, we will use the linden algorithm in our experiments as the most well-performed competitor.

The algorithms we have tested are the following:

- *BH-CGL*: This is the blocking heap implementation illustrated in Algorithms 4 and 5. It uses a coarse-grain lock to synchronize the threads in accessing the queue.
- *BH-FGL*: This is the blocking heap implementation illustrated in Algorithm 6 and 7. It uses the technique of fine-grain locking, which employs one lock for each node. Threads that access the queue use hand-over-hand locking [SHAV].
- *LIND*: This is the Linden algorithm that is described in detail in Section 3.3.

The benchmarks we have tested are the following:

- *Insert-Only*: This is a benchmark where threads execute only INSERT operations. Initially, the local heaps are empty. A total of 1.000.000 INSERT operations (1.000.000 total elements) are performed. They are equally distributed among the threads. Thus, every thread performs 1.000.000/Threads, INSERT operations, where Threads is the number of threads in the system.
- *DeleteMin-Only*: This is a benchmark where threads execute only DELETEMIN operations. Initially, the local heaps are filled with a total of 1.000.000 elements. Each heap stores 1.000.000 / Threads, elements. When the benchmark ends, all the heaps are empty.
- *Pairs*: Initially, the local heaps are filled with a few elements e.g., 512 each. Every thread runs 500.000 pairs of operations, where each pair is an INSERT operation, followed by a DELETEMIN operation, in that order.

The throughput is measured in MillOps/sec (1 million operations/second) and the cache misses are measured in millions, as seen in some of the experiments. The result of each experiment is the average of 10 repetitions of the same experiment. Furthermore, there is also random work after each operation, either it is INSERT or DELETEMIN. This random work is simulated, using an empty “for” loop with

the number of iterations varying between a “low” integer and a “high” integer, given by the user at the beginning of the experiment. The algorithms in every experiment used the same random work to produce the final results.

5.3. Existing Algorithms

Figures 1 and 2 compare the performance of BH-CGL with that of BH-FGL, showing that the implementation that uses a coarse grain lock is more well performed. Figures 3 and 4 show that the reason for this is that BH-FGL causes many more cache misses than BH-CGL in both Insert-Only and DeleteMin-Only experiments. This is because, in the fine-grain algorithms, the threads access multiple locks in every operation they execute. In what follows, we include only BH-CGL in our experiments.

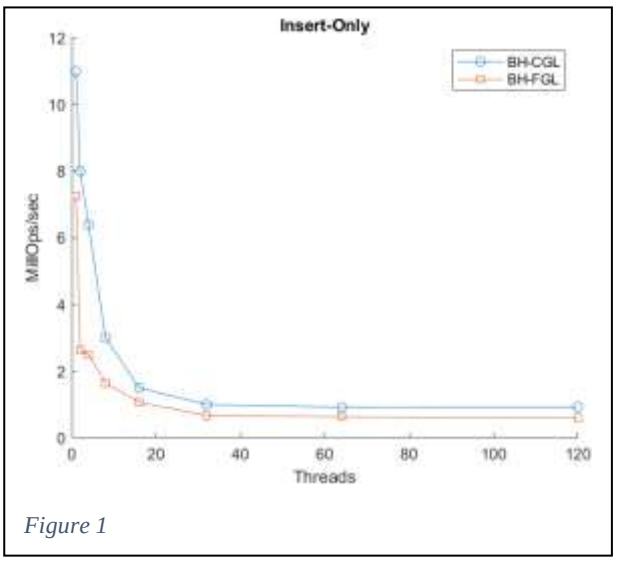

Figure 1

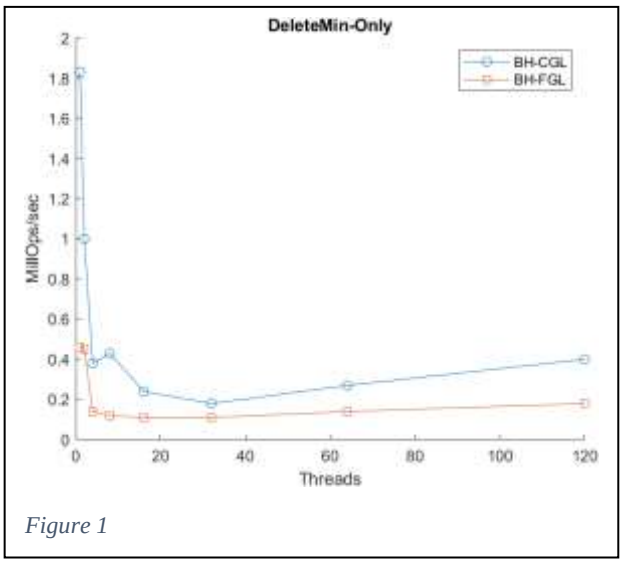

Figure 1

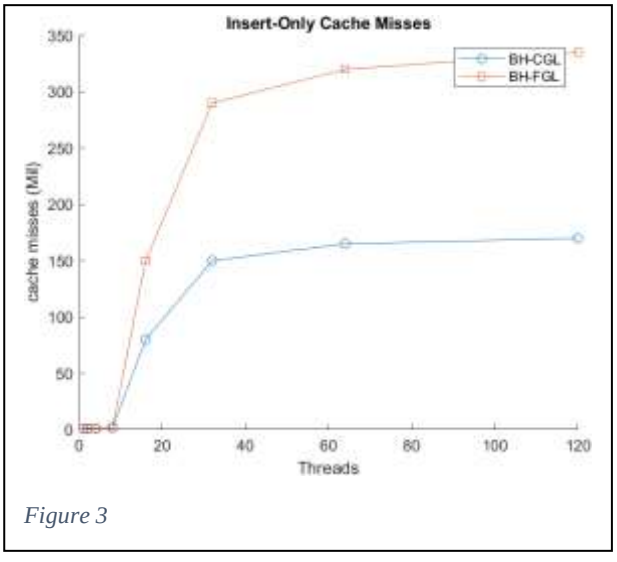

Figure 3

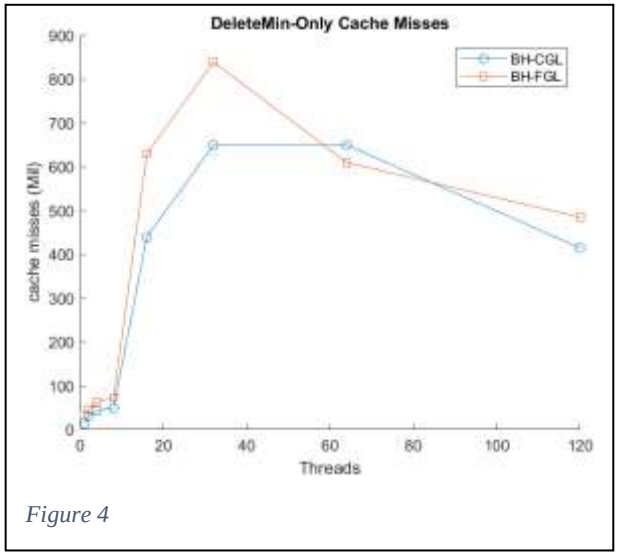

Figure 4

5.4. Hierarchical Algorithm – Individual Insert operations

Recall that INSERT operations are executed in an embarrassingly parallel way due to locality.

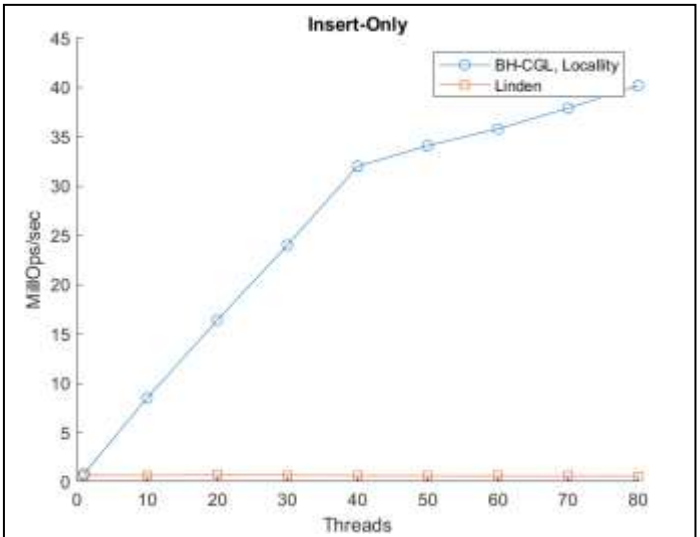

Figure 2

Figure 5 shows that the hierarchical algorithm achieves almost linear speedup (up to 30 threads) in executions that contain only insert operations.

5.5. Hierarchical Algorithm – Individual DeleteMin operations

Figure 6 focuses on the performance of a preliminary version of the DeleteMin algorithm, showing how this version behaves in the DeleteMin-Only case. This version, called SL-Del, employs only workers and one coordinator (so there are no leaders). Each worker moves upwards to the coordinator the highest priority elements of its local queue by storing them in a two-dimensional array (with as many rows as the number of threads). The worker has its own buffer in this array to store these elements (in the experiments below this buffer has 150 elements). Then, the coordinator reads through this array, the highest-priority element of each worker and decides which element is the one with the globally highest priority among all local heaps. In this solution, the coordinator has high overhead because it needs to read the buffers of all workers.

In Figure 6, the blue line shows the performance of an implementation of the above scheme which utilizes a dedicated coordinator (i.e., thread 0 plays the role of the coordinator). The red line illustrates the performance of the multiple coordinators version of the algorithm. In this version, instead of using

a predefined thread as the coordinator, we have implemented a locking mechanism, so that every worker can compete to decide which thread will play the role of the coordinator. This method is a preliminary version to the helping mechanism that will be used in the final version.

The addition of the locking mechanism had a negative impact in performance (red lines). The reason for that is shown in Figure 7. In that experiment, we measured the cache misses of the execution of the program for these two implementations and the difference is noticeable. The increase in the number of caches in the multiple-coordinators version is expected, because of the contention that is caused on the lock as the number of threads increases. When the system has more than 20 threads, the active requests come from different sockets, thus the coordinator had to serve requests from multiple different sockets. The negative effect that this has in the performance is obvious in Figures 6 and 7.

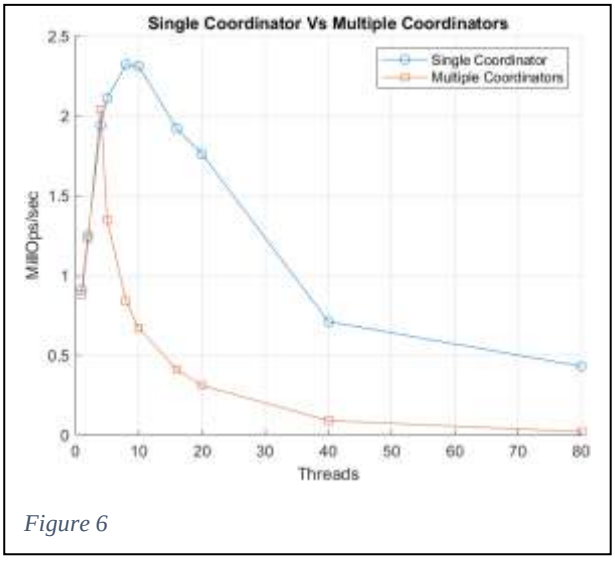

Figure 6

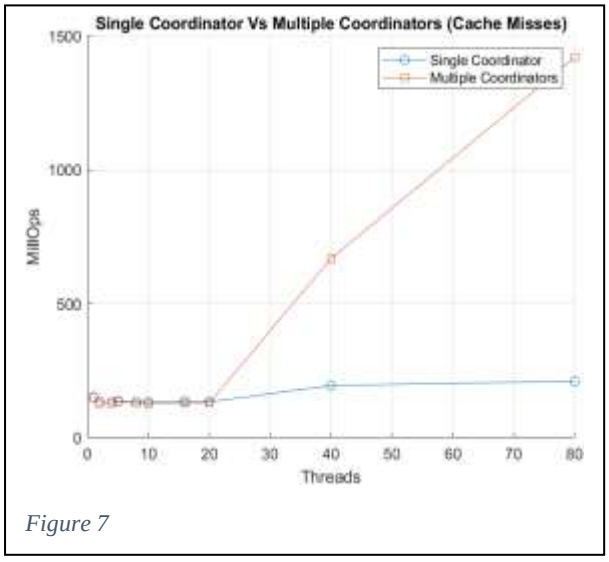

Figure 7

Figure 8 shows the performance of the hierarchical algorithm, described in Section 4.2, which we refer as Hierarchical. A thread can become either a worker, or a leader, or the coordinator. Only leaders communicate with the coordinator, and they do so by using a data structure, local to each group/socket. In that way, even if the number of threads gets larger, the algorithm performs well, due to locality.

Figure 8 includes two lines for the LIND algorithm and only one for the Hierarchical algorithm. The red line represents the throughput of LIND when one million DELETEMIN operations are performed, whereas the yellow, when ten million operations are performed. This increase on the number of operations slows the execution down and drops the overall throughput for the LIND algorithm. We do not see a second diagram for Hierarchical because experimenting in these two settings does not have any impact on the throughput of the algorithm. Thus, Figure 8 illustrates a nice feature of employing local heaps. Specifically, LIND (yellow line) starts from a data structure which contains

a large number of elements (to be able to perform the ten million DELETEMINs). In LIND, starting from a big data structure causes performance overheads due to the high cost of the DELETEMINs operations, which are executed at an early stage. On the contrary, this bad behavior is avoided in Hierarchical, because the initially inserted elements are split to many different local queues, and an hierarchical structure for accessing these queues is employed.

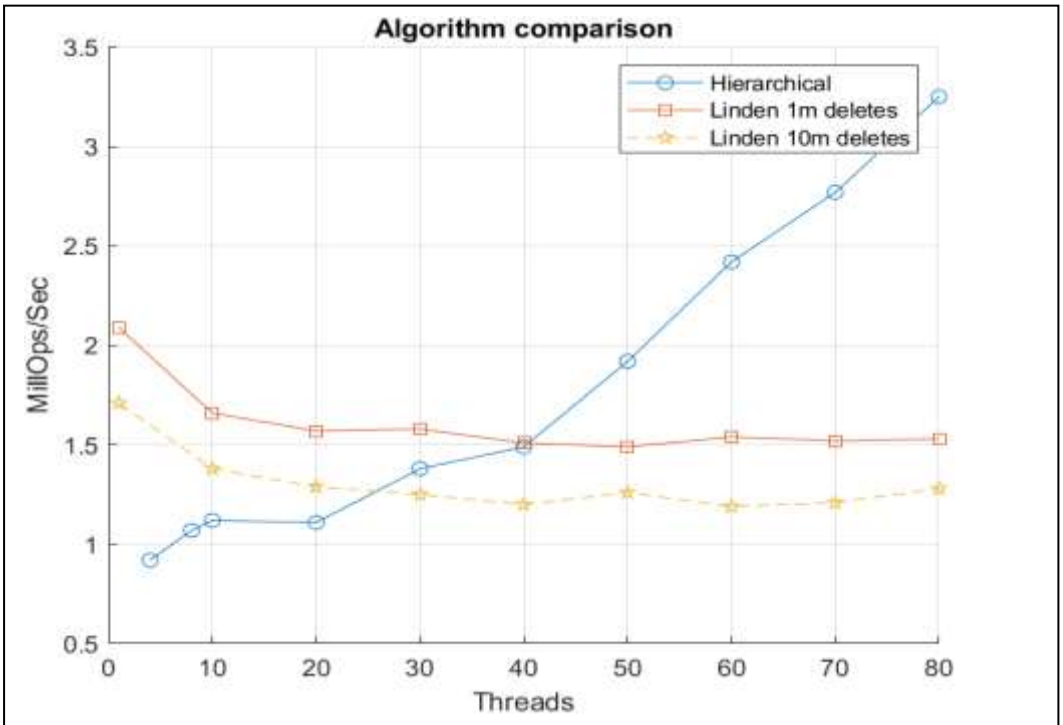

Figure 8

In the graphs presented above, at every point in time, all threads were active, operating on their local priority queues. However, in a general setting, not all threads will always be active. Figure 9 refers to such a setting. The x axis illustrates the number of threads that are inactive, thus not performing DELETEMIN operations on the priority queue. The different graphs in figure 9 are the following.

- **Missing threads from a single socket**: The missing threads all belong to the same socket. In this case, the value 4 of the x axis indicates that 4 threads are missing from one socket, thus the total number of missing threads in this case is 4.
- **Missing threads from 2 sockets**: Threads are removed from only 2 sockets. So, the value 4 of the x axis now indicates that 4 threads are missing from two sockets, resulting in a total of 8 missing threads.
- **Missing threads from a single socket with 3 active sockets**: In this setting, an entire socket is inactive, so the total number of missing threads is (Threads Missing + 20). In this case, the value 4 of the x axis indicates that 4 threads are missing from each of the three active sockets. Thus, the total number of missing threads in this case is 32.

- **Red dot:** Highest point of the throughput achieved by Hierarchical in Figure 8.
- **Blue dot:** Lowest point of the throughput achieved by Hierarchical in Figure 8.

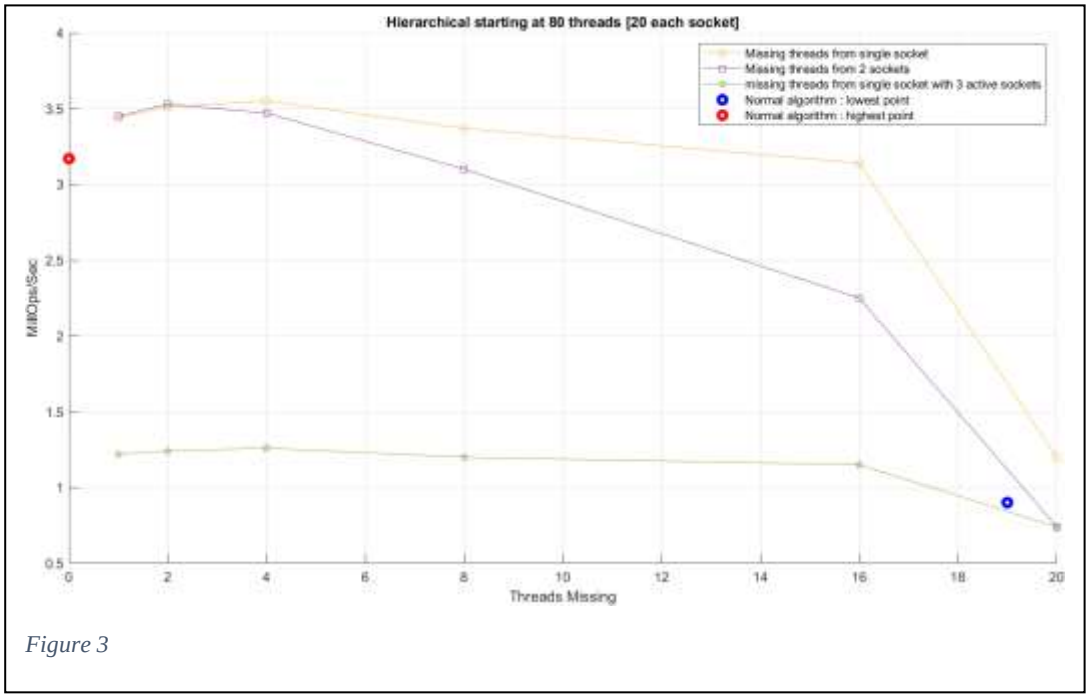

Figure 3

Note that the red dot (highest point of Hierarchical's throughput in Figure 8), is not the highest point among the graphs of Figure 9. In other words, the algorithm performs slightly better when there are inactive threads in specific sockets. This probably occurs because as the number of threads increases in every socket, the synchronization cost and the time spent on cache misses may be increased in comparison to the case where a few threads are missing from each socket.

5.6. Hierarchical Algorithm – Simultaneous Inserts and DeleteMins

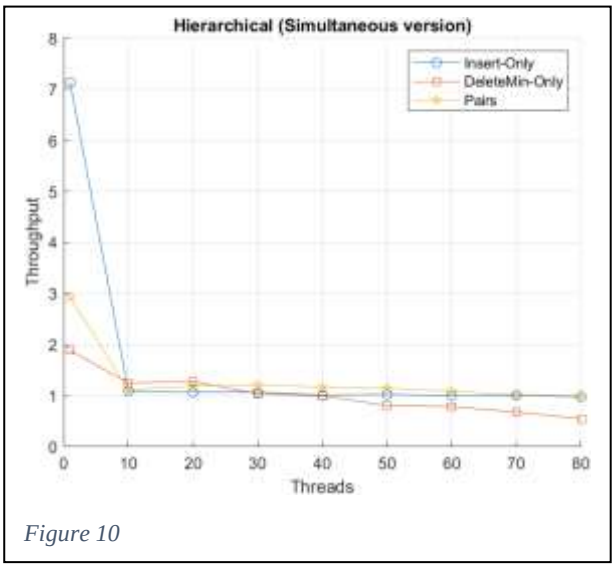

Figure 10

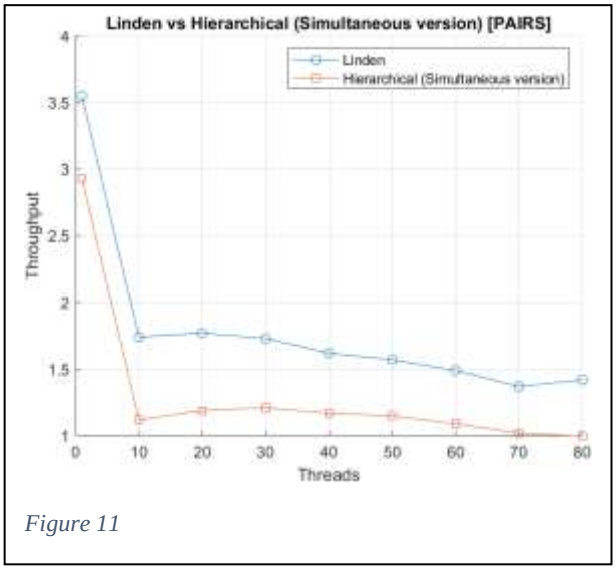

Figure 11

Since the Hierarchical algorithm has changed to support simultaneous INSERTS and DELETEMINS, the Insert-Only and Delete-Only performances have been affected as well, due to the different algorithm that is used.

The LIND algorithm has better results when we use the PAIR benchmark, compared to this version of the hierarchical algorithm. Figures 10 and 11 illustrate the impact the NUMA-aware techniques can have. In the previous version of the algorithm (Figure 8), the setting was so that the locality techniques could be used more extensively. On the contrary, in the version we study here, we made adjustments, which reduced the locality degree in order to support simultaneous INSERTS and DELETEMINs. This is why the algorithm, as seen in Figures 10 and 11, performs poorly in comparison to the previous version.

Summarizing, our experiments show that performance is highly affected by the locality degree of the methods we employ, and the locality degree allowed by the setting we study.

6. Conclusion

In this paper, we made a first effort to come up with a NUMA-aware priority queue implementation. We experimentally compared the newly-proposed implementation with standard lock-based implementations of priority queues and the Linden algorithm presented in [LIND]. Our experimental results showed that the proposed technique performs well in some restricted cases. However, the simultaneous INSERT and DELETEMIN versions of the algorithm did not perform the same well, so a bigger effort is required in coming up with a fully general approach that exhibits high performance. The greatest difference between these cases, comes from the necessity to reduce the degree of locality used due to helping (or absence of threads operating on the queues in some cases).

7. References

[SHAV] Maurice Herlihy and Nir Shavit. 2012. The Art of Multiprocessor Programming, Revised Reprint (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[LIND] Jonatan Linden and Bengt Jonsson. "A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention". In: Principles of Distributed Systems. Springer, 2013, pp. 206-220.

[PUGH] Pugh, W.: Concurrent maintenance of skip lists. Tech. Rep. CS-TR-2222, Dept. Of Computer Science, University of Maryland, College Park (1990).

[LHAR] Timothy L. Harris:A Pragmatic Implementation of Non-blocking Linked-Lists. *DISC 2001*: 300-314.

[COMB] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios Kosmas. 2022. 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). Association for Computing Machinery, New York, NY, USA, 337–352.

[SPRL] Dan Alistarh, Justin Kopinsky, Jerry Li, Nir Shavit. The SprayList: A Scalable Relaxed Priority Queue.

[NOBL] Haakan Sundell and Philippas Tsigas. “Fast and lock-free concurrent priority queues for multithread systems”. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS). IEEE. 2003, 11-pp.

[HEAP] Galen C Hunt et al. “An efficient algorithm for concurrent priority queue heaps”. In: Information Processing Letters 60.3 (1996), pp. 151-157.