A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms

Orr Tamir[1], Adam Morrison[2], and Noam Rinetzky[3]

1 **ortamir@post.tau.ac.il Blavatnik School of Computer Science, Tel Aviv University**
2 **mad@cs.technion.ac.il Computer Science Department, Technion—Israel Institute of Technology**
3 **maon@cs.tau.ac.il Blavatnik School of Computer Science, Tel Aviv University**

Abstract

Existing concurrent priority queues do not allow to update the priority of an element after its insertion. As a result, algorithms that need this functionality, such as Dijkstra's single source shortest path algorithm, resort to cumbersome and inefficient workarounds. We report on a heap-based concurrent priority queue which allows to change the priority of an element after its insertion. We show that the enriched interface allows to express Dijkstra's algorithm in a more natural way, and that its implementation, using our concurrent priority queue, outperform existing algorithms.

1998 ACM Subject Classification D.1.3 [Programming Techniques]:Concurrent Programming

Keywords and phrases priority queues, concurrent data structures, `changeKey`(), Dijkstra's single-source shortest path algorithm

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.186

1 Introduction

A priority queue data structure maintains a collection (multiset) of items which are ordered according to a *priority* associated with each item. Priority queues are amongst the most useful data structures in practice, and can be found in a variety of applications ranging from graph algorithms [21, 6] to discrete event simulation [8] and modern SAT solvers [5]. The importance of priority queues has motivated many concurrent implementations [2, 3, 11, 15, 16, 17, 23, 24, 25, 26]. These works all focus on the performance of two basic priority queue operations, which consequently are the only operations provided by concurrent priority queues: `insert`(d, p), which adds a data item d with priority p, and `extractMin`(), which removes and returns the highest-priority data item.[1]

It turns out, however, that important applications of priority queues, such as Dijkstra's single-source shortest path (SSSP) algorithm [6, 4], need to update the priority of an item after its insertion, i.e., *mutable priorities.* Parallelizing these algorithms requires working around the lack of mutable priorities in today's concurrent priority queues by inserting new items instead of updating existing ones, and then identifying and ignoring extracted items with outdated priorities [2]—all of which impose overhead. Sequential heap-based priority queues support mutable priorities [4], but concurrent heaps have been abandoned in favor of skiplist-based designs [17] whose `extractMin`() and `insert`() are more efficient and scalable.

[1] In this paper, we consider lower p values to mean higher priorities.

Thus, the pursuit of performance for the basic priority queue operations can, ironically, end up leading to worse *overall* performance for the parallel *client application*.

The principle driving this work is that we should design concurrent data structures with the *overall performance* of the *client* as the goal, even if this entails compromising on the performance of the individual data structure operations. We apply this principle to priority queues by implementing CHAMP, a Concurrent Heap with Mutable Priorities, which provides a `changeKey`() operation to update priorities of existing items. We use CHAMP to implement a parallel version of Dijkstra's SSSP algorithm, and our experimental evaluation shows that, as parallelism increases, CHAMP's efficient `changeKey`() operation improves the client overall performance by saving it from doing *wasted work*—the overhead that arises when working around the lack of `changeKey`() support in prior designs. This occurs despite the fact that CHAMP's `extractMin`() and `insert`() operations do not scale as well as in prior designs.

Contributions: To summarize, we make the following technical contributions:

1. We present CHAMP, a linearizable lock-based concurrent priority queue that supports mutable priorities. CHAMP is an adaptation of the concurrent heap-based priority queue of [11] to support the `changeKey`() operation.
2. We convert an existing parallel SSSP algorithm to utilize the `changeKey`() operation.
3. We implement and evaluate our algorithms.

Arguably, the more important contribution of this paper is the conceptual one: A call to pay more attention in the design of data structures and interfaces to the overall performance and programmatic needs of the client applications than to the standalone scalability of the supported operations, as at the end, the client is always right.

2 Priority Queues with Mutable Priorities

A *priority queue with mutable priorities* (PQMP) is a data structure for maintaining a multiset $\boldsymbol{A}$ of *elements*, where each element is a pair comprised of a *data item* d and a value k called *priority*.[2] A PQMP supports the following operations:

- `extractMin`(): Removes and returns the element which has the highest priority in the queue.[1] In case multiple elements have the highest priority, one of them is chosen arbitrarily. If the queue is *empty*, a special value is returned.
- `peek`(): Acts similarly to `extractMin`(), except that the chosen element is not removed.
- `insert`(d, k): Inserts into the queue an element comprised of a given data item d and *priority* k, and returns a unique *tag* e identifying the element. If the queue has reached its *full capacity*, the element is not inserted, and a special value is returned.
- `changeKey`(e, k): Sets the priority of element e to k. If e is not in the queue, the operation has no effect. (The latter behavior was chosen for the sake of simplicity. An alternative, could have been, e.g., to return a special value or to raise an exception.)

The use of *tags* to identify elements in the queue, instead of their data items, as done, e.g., in [4, Ch. 6.5], allows to store in queue multiple elements with the same data item.

3 A Sequential Heap with Mutable Priorities

PQMPs can be implemented with a *binary heap* data structure. A binary heap is an almost complete binary tree that satisfies the *heap property*: for any node, the *key* of the node is less

[2] The term *key* is sometimes used instead of *priority*.

```
Element[0..Length] A
int Last

Lock[0..Length] L
class Element
  Priority key
  Data data
  int pos

  bool up
```

Figure 1 The data representation of the heap.

```
swap(i,j)
  temp = A[i]
  A[i] = A[j]
  A[j] = temp
  A[i].pos = i
  A[j].pos = j
```

```
leftChild(i)
  return 2*i
```

```
rightChild(i)
  return 2*i+1
```

```
parent(i)
  return ⌊i/2⌋
```

Figure 2 Auxiliary procedures.

than or equal to the keys of its children, if they exist [4]. Binary heaps are often represented as arrays: the root is located at position 1, and the left and right children of the node at location i are located at positions $2i$ and $2i+1$, respectively. Position 0 is not used. Heaps support `extractMin`(), `peek`(), `insert`(), and `changeKey`() operations that map naturally to respective priority queue operations, if we use elements' priorities as keys.

In the following, we describe a sequential implementation of an array-based heap. The sequential implementation is fairly standard. Thus, our description mainly focuses on certain design choices that we made in the concurrent algorithm which can be explained in the simpler sequential settings.

Fig. 1 defines the type `Element`, and shows the data representation of a heap using two global variables: An array `A` and an integer `Last`. (Array `L` and the `up` field in elements are used only by the concurrent algorithm.) A heap with maximal capacity `Length` is comprised of an array `A` of pointers to `Element`s with `Length+1` entries and a counter `Last` which records the number of elements in the heap. We say that an element is *in* the heap if some entry in `A` points to it. We refer to the element pointed to by `A[1]` as the *root element*.

An element is comprised of three fields: `key` keeps the element's priority, `data` stores an application-specific data item, and `pos` records the *position* of the element in the heap: Given an element e, the position of an element e is the index of an entry in `A` which points to e, or -1 if e is not in the heap, i.e., if $e.\texttt{pos} \neq -1$ then $\texttt{A}[e.\texttt{pos}] = e$.

Figure 3 shows the pseudocode of a sequential heap. The operations use the auxiliary functions defined in Fig. 2. Each heap operation consists of two parts. First, it inspects, adds, removes, or changes an element. Then, because this change may violate the heap property, it *heapifies* the heap in order to restore the heap property. In the following, we describe how heap operations are implemented and then how heapifying is done. We use the $\cdot_{\mathsf{seq}}$ subscript to distinguish between the sequential operations and the concurrent ones.

- $\texttt{peek}_{\mathsf{seq}}()$: Returns the root element or *null* if the heap is empty.
- $\texttt{insert}_{\mathsf{seq}}(d, k)$: Returns *null* if the heap is *full*. Otherwise, it allocates and inserts a new element into the heap.[3] The element is placed at the $\texttt{Last}+1$ entry of `A`, which is at the lowest level of the heap, right after the last occupied position in the array. After the operation completes its second phase (heapify), it returns a pointer to the new element as its tag.

[3] In this paper, we sidestep the difficult problem of concurrent safe memory reclamation [19, 18, 9], and assume that memory is recycled either by the client or by an underlying garbage collector [12].

```
peek_seq()
 return A[1]
```

```
extractMin_seq()
 min = A[1]
 ls = Last
 if (ls = 0)
   return null
 min.pos = -1
 if (ls = 1)
   A[1] = null
 else
   A[1] = A[ls]
   A[1].pos = 1
   A[ls] = null
   if (ls = 2)
     Last = 1
   else
     Last = ls - 1
     bubbleDown_seq(A[1])
 return min
```

```
insert_seq(key, data)
 if (Last = Length)
   return null
 e = new Element(
  key, data, Last+1)
 if(Last = 0)
   A[1] = e
   Last = 1
   unlock(L[1])
 else
   lock(L[Last + 1])
   e.up = true
   A[Last + 1] = elm
   Last = Last + 1
   unlock(L[1])
   bubbleUp(e)
 return e
```

```
changeKey_seq(e, k)
 if (e.key∉{1..Last})
   return false
 if (k < e.key)
   e.key = k
   bubbleDown_seq(e)
 else if (k > e.key)
   e.key = k
   bubbleUp_seq(e)
 return true
```

```
bubbleDown_seq(e)
 min = e.pos
 do
   i = min
   l = leftChild(i)
   r = rightChild(i)
   if (l ≤ Last)
     if (A[l].key < A[i].key)
       min = l
     if (A[r] ≠ null and
         A[r].key < A[min].key)
       min = r
     if (i ≠ min)
       swap(i, min)
 while(i ≠ min)
```

```
bubbleUp_seq(e)
 i = e.pos
 do
   par = parent(i)
   if(A[i].key < A[par].key)
     swap(i, par)
     i = par
 while (i = par)
```

Figure 3 Pseudo code of a *sequential* heap with mutable priorities. `Length-1` is the capacity of the heap. We assume that `changeKey`$_{\text{seq}}$`(e, k)` is invoked with `e≠null`.

- `extractMin`$_{\text{seq}}$(): Returns *null* if the heap is *empty*. Otherwise, it replaces the root element with the rightmost element in the tree, which is the last occupied position in the array. After the second part (heapify), the operation returns the previous root element.
- `changeKey`$_{\text{seq}}$(): Changes the key of the specified element e to k, if e is in the heap. Note that position field of an element is used to locate the entry in `A` pointing to it.

The second part of the operation restores the heap property by heapifying: In `extractMin`$_{\text{seq}}$(), we use `bubbleDown`$_{\text{seq}}$(), which shifts the root element whose key might become larger than its children down in the heap until the heap property is restored. In `insert`$_{\text{seq}}$(), we use `bubbleUp`$_{\text{seq}}$(), which carries an element up in the heap until its key is larger than that of its parent. Finally, `changeKey`$_{\text{seq}}$() uses `bubbleDown`$_{\text{seq}}$() or `bubbleUp`$_{\text{seq}}$() as appropriate. Note that when an element is being swapped, its position field is updated too and that when an element is removed from the heap, its position is set to -1.

4 Champ: A Concurrent Heap with Mutable Priorities

In this section, we present a concurrent PQMP data structure based on Champ, a concurrent heap with mutable priorities. At its core, Champ is an array-based binary heap, very much

```
extractMin()
 Lock(A[1])
 min = A[1]
 ls = Last
 if (ls = 0)
   unlock(L[1])
   return null
 A[1].pos = -1
 if (ls = 1)
   Last = 0
   A[1] = null
   unlock(L[1])
 else
   lock(L[ls])
   A[1] = A[ls]
   A[1].pos = 1
   A[ls] = null
   Last = ls - 1
   unlock(L[ls])
   if (ls = 2)
     unlock(L[1])
   else
     bubbleDown(A[1])
 return min
```

```
insert(key, data)
 lock(L[1])
 if (Last = Length)
   unlock(L[1])
   return null
 e = new Element(
    key, data, Last+1)
 if(Last = 0)
   e.up = false
   A[1] = e
   Last = 1
   unlock(L[1])
 else
   lock(L[Last + 1])
   e.up = true
   A[Last + 1] = e
   Last = Last + 1
   unlock(L[1])
   bubbleUp(e)
 return e
```

```
changeKey(e, k)
 while (lockElement(e))
   if (e.up)
     unlock(L[e.pos])
   else
     if (k < e.key)
       e.up = true
       e.key = k
       bubbleUp(e)
     else if (k > e.key)
       e.key = k
       bubbleDown(e)
     else
       unlock(L[e.pos])
     return
```

```
peek()
 lock(L[1])
 ret = A[1]
 unlock(L[1])
 return ret
```

Figure 4 The pseudo code of CHAMP, a *concurrent* priority queue with updatable key based on a binary heap. The concurrent heapifying procedures are presented in Fig. 5. Auxiliary procedures, e.g., `swap()`, are defined in Fig. 2.

like the sequential algorithm described in the previous section. Synchronization is achieved using a fine-grained locking protocol, derived from the one used in [11] (see Sec. 4.3.)

CHAMP is implemented using the global variables shown in Fig. 1. Variables `A` and `Last` play the same role as in the sequential setting (see Sec. 3.) Variable `L` is an array of locks which contains one lock for every entry in `A`. Intuitively, lock `L[`i`]` is used to synchronize accesses to the i-th entry of `A` and to the element `A[`i`]` points to. Lock `L[1]`, which we refer to as the *root lock*, is also used to protect the `Last` variable. A thread is allowed to modify a shared memory location only under the protection of the appropriate lock. Read accesses to the entries of array `A` and to variable `Last` should also be protected by a lock. In contrast, fields of elements can be read without a lock.[3]

Figures 4 and 5 show the pseudocode of our concurrent heap. CHAMP implements the interface of a PQMP. As expected, the concurrent operations provide the same functionality as the corresponding sequential counterparts, and, like them, also consist of two stages: First, every operation grabs the locks it requires and inspects, adds, removes, or changes the shared state. Then, it invokes `bubbleUp`(e) or `bubbleDown`(e) to *locally* restore the heap property.

The more interesting aspects of the first part of the operations are summarized below:

- `peek`(): Although `peek`() only reads a single memory location, it starts by taking the root lock. This is required because another thread might perform an `insert`() operation concurrently, which could lead to a state where the key of the root is *not* lower than that of its children. Returning such a root element would violate linearizability (see Sec. 4.1).
- `insert`(), `extractMin`(), and `changeKey`(): The first part of these operations is the same

```
bubbleDown(e)
 min = e.pos
 do
  i = min
  l = LeftChild(i)
  r = RightChild(i)
  if (l ≤ Last)
    lock(L[l])
    lock(L[r])
    if (A[l] ≠ null)
      if (A[l].key < A[i].key)
        min = l
      if (A[r] != null and
          A[r].key < A[min].key)
        min = r
     if (i ≠ min)
       if(i == l)
         unlock(L[r])
       else
         unlock(L[l])
       swap(i, min)
       unlock(L(i))
 while(i ≠ min)
 unlock(L[i])
```

```
lockElement(e)
 while(true)
   i = e.pos
   if (i == -1)
     return false
   if (trylock(L[i]))
     if (i == e.pos)
       return true
     unlock(L[i])
```

```
bubbleUp(e)
 i = e.pos
 iLocked = true
 parLocked = false
 while (1 < i)
   par = Parent(i)
   parLocked = tryLock(L[par])
   if (parLocked)
     if (!A[par].up)
       if(A[i].key < A[par].key)
         swap(i, par)
       else
         A[i].up = false
         unlock(L[i])
         unlock(L[par])
         return
     else
       unlock(L[par])
       parLocked = false
   unlock(L[i])
   iLocked = false
   if (parLocked)
     i = par
     iLocked = true
   else
     iLocked = lockElement(e)
     i = e.pos
 e.up = false
 if (iLocked)
   unlock(L[e.pos])
```

Figure 5 Concurrent heapifying procedures.

as that of their sequential counterparts, but with two exceptions:

Element locking. The operations begin by taking locks. `changeKey`(e, k) takes the lock of the array entry pointing to e. (We explain the reason for using a loop later on.) All other operations grab the root lock. Also, the operations avoid calling the heapifying procedures in cases where the *global* heap property is guaranteed to hold after the change, e.g., when an element is inserted into an empty heap or when the last element in the heap is extracted.

Signaling upward propagation. `insert`(e) and `changeKey`(e) set the `up` flag of e before invoking `bubbleUp`(e). This indicates that e is being propagated up the heap, which help synchronize concurrent `bubbleUp`() operations, as we shortly explain.

The second part of every operation *locally* restores the heap property using `bubbleUp`() and `bubbleDown`(). The two shift elements up, respectively, down the heap until the heap property is *locally* restored: `bubbleUp`(e) stops when the key of e is bigger than that of its parent. `bubbleDown`(e) stops when the key of e is smaller than the keys of its children. Both operations stop if they detect that e was extracted from the heap.

Both `bubbleDown`() and `bubbleUp`() employ the hand-over-hand locking protocol [13] (also known as the *tree-locking* protocols), but they acquire the locks in different orders: `bubbleDown`() takes the lock of the children of a node e only after it holds the lock of e while `bubbleUp`() takes the lock of the parent of e only when it has e's lock. The hand-over-hand protocol ensures deadlock freedom when all the operations take their locks in the same order. However, if different orders are used, deadlock might occur. To prevent deadlocks, `bubbleDown`() takes locks using **tryLock**() instead of **lock**(), and in case the needed lock is not available it releases all its locks and then tries to grab them again.

An interesting case may happen when `bubbleUp`(e) attempts to get a hold of e's lock after its release: It is possible that the up-going element e have been pulled upwards by a concurrent down-going `bubbleDown`() operation. In fact, the element might have been removed from the heap all together. The auxiliary procedure `lockElement`(e) is thus used to locate a possibly relocated element. It repeatedly finds e's position in `A` using its position field and tries to lock the corresponding entry. `lockElement`(e) loops until it either obtained the lock protecting e's position, or until it finds that e has been extracted from the heap, indicated by having value -1 in its position field.

There is a tricky synchronization scenario that might occur when a `bubbleUp`(e) operation t_1 manages to lock an entry i pointing to an up-going element e'. (This might occur if the operation t_2 bubbling-up e' has not managed to bring e' to a position where its value is bigger than that of its parent, but had to release its locks to prevent a possible deadlock.) If e's key is bigger than that of e', t_1 might come to the wrong conclusion that it has managed to restore the heap-property and terminate. However, the property was restored with respect to an element, e, which is *not* in its “right” place. To ensure that such a scenario does not happen, `bubbleUp`(e) releases its lock when it detects that the parent of the element it is propagating is also being bubbled-up, indicated by its `up` flag.

Note. Our algorithm supports concurrent priority changes of a given element e. These operations synchronize using the loop at the entry to `changeKey`(): A thread can enter the loop only if it manages to lock the position of element e. In case it detects an ongoing `bubbleUp`(e) operation, it releases the lock and retries. (Note that the lock cannot be obtained if there is an ongoing `bubbleDown`(e) operation). This ensures that there is only one thread that changes the key of an element. We note that in certain clients, e.g., Dijkstra's SSSP algorithm, the client threads prevent concurrent priority changes of a given element. In this cases, e.`up` is always false when we enter the loop.

4.1 Linearizability

Champ is a linearizable [10] priority queue with mutable keys. Intuitively, linearizability means that every operation seems to take effect instantaneously at some point between its invocation and response. In our case, these *linearization points* are as follows:

1. `peek`(), `extractMin`(), and `insert`() when invoked on a heap containing two or less elements: The point in time when the operation obtained the root lock.
2. `insert`() and `changeKey`(e) which decreased the priority of e: The linearization point happens during the call to `bubbleUp`(e). It is placed at the last configuration in which an entry in `A` pointed to e before its `up` field was set to false.
3. `changeKey`(e) which did not not find e in the heap, increased its priority or did not change it: The point in time in which the call to `lockElement`(e) returned.

Technically, the proof of linearization rests upon the following invariants:

(i) No element is stored in position 0.
(ii) The entries `A[1]`..`A[Last]` contain non-*null* values.
(iii) The value of every entry `A[Last+1]`$\cdots$`A[Length]` is *null*, except perhaps during the first part of `insert`() when `A[Last` $+ 1$`]` might have the same non-null value as `A[1]`.
(iv) The position field `pos` of an element agrees with its position in the heap, i.e., if e.`pos` $= i \wedge 0 < i$ then `A[`e`].pos` $= i$, except perhaps during a `swap`() involving `A[`i`]`.
(v) If the i-th entry in the heap and its parent $j = \lfloor i/2 \rfloor$ are not locked and, in addition, `A[`i`].up` $=$ *false* and `A[`j`].up` $=$ *false* then `A[`j`].key` $\leq$ `A[`i`]`.

Most of the invariants are quite simple and rather easy to verify, in particular, when we recall that the global variables and the elements can be modified only when the thread holds the lock which protects both the entry and the element it points to. Note that if an element is pointed to by two entries then the same operation holds the two locks protecting these entries. The only time a thread may modify a field of an object without holding the respective lock is when it sets off the `up` field of an element which was removed from the heap. The key invariant is (v). It provides a local analogue of the heap property. Intuitively, it says that if an element violates the heap property then there exists an ongoing operation which is “responsible” for rectifying the violation. Furthermore, any apparent inconsistency that might occur due to the violation can be mitigated by the placement of the linearization point of the responsible operation in the global linearization order. For example, we can justify an `extractMin`() operation which returns an element e although the heap contains a non-root element e' which has a lower key than e by placing its linearization point before that of the operation responsible for inserting e' or reducing its key. Invariant (v) ensures that such an operation is still active when `extractMin`() takes possession of the root lock.

4.2 DeadLock-Freedom

Champ is deadlock-free. All the operations except `bubbleUp`() capture their locks according to a predetermined order, thus preventing deadlock by construction. `bubbleUp`() uses **tryLock**(), and releases its locks if the latter fails, thus avoiding deadlocks all together.

4.3 Comparison with Hunt’s Algorithm [11]

Our priority queue is based on concurrent heap of Hunt et al. [11], as both use fine-grained locking to synchronize bottom-up `insert`()s and top-down `extractMin`()s. The main difference is the addition of the `changeKey`() operation. There are also some subtle differences in certain key aspects of the implementation.

- We use a different technique to prevent deadlocks between concurrent up-going `bubbleUp`()s and down-going `bubbleDown`()s: In [11], `insert`()s and `extractMin`()s takes locks in the same order. Specifically, they lock the parent before its child. Deadlock is prevented by having `insert`()s release their locks before they climb up the heap. In our algorithm, `insert`() and `changeKey`() take their locks in reverse order, thus possibly saving some redundant **unlock**() and re**lock**() operations. Deadlock is prevented using **tryLock**()s operations as explained in Sec. 4.2.
- In both techniques, an element e bubbled up the heap might change its position due to a down-going operation. In [11], the up-going operation propagating e finds it by climbing up the heap. In our case, we embed a position index inside the node which allows to locate it in a more direct fashion. The position index is particularly beneficial for `changeKey`(e) as it allows to efficiently check whether e is in the heap.

- Hunt reduces contention between `insert()` operations using a bit-reversal scheme to determine the index into which a new element is added. We use the standard scheme for insertions which maintains all the elements in a single contiguous part. We note that we can easily import their method into our algorithm.
- Finally, Hunt's priority queue is *not* linearizable, while ours is. The culprit is the `extractMin()` procedure which first removes the `Last` element from the heap and only then places it at the root. This allows for certain non-linearizable behaviors to occur. It is important to note, however, that there is no claim of linearizability in [11], and once the reason for the non-linearizable behavior is known, changing the algorithm to be linearizable is rather easy.

5 Case Study: Parallelizing Dijkstra's SSSP Algorithm

Important applications of priority queues, such as Dijkstra's single-source shortest path (SSSP) algorithm [6, 4] and Prim's minimal spanning tree (MST) algorithm [21, 4] need to update the priority of an item after its insertion. i.e., *mutable priorities*. In this work, we close the interface gap between sequential and concurrent priority queues by importing the `changeKey()` operation from the sequential setting to the concurrent one. To evaluate the benefits clients may gain by using the extended interface, we adapted a parallel version of Dijkstra's SSSP algorithm to use `changeKey()`.

The SSSP problem is to find, given a (possibly weighted) directed graph and a designated source node s, the weight of the shortest path from s to every other node in the graph. For every node v, we refer to the weight of a shortest $s \rightsquigarrow u$ path as v's *distance* from s. The asymptotically fastest known sequential SSSP algorithm for arbitrary directed graphs with unbounded non-negative weights is Dijsktra's algorithm [6, 7].

Dijkstra's algorithm partitions the graph into *explored* nodes, whose distance from s is known, and *unexplored* nodes, whose distance may be unknown. Each node v is associated with its distance, $dist(v)$, which is represented as a field in the node. The algorithm computes the distances by iteratively exploring the edges in the frontier between explored and unexplored nodes. The initial distances are $dist(s) = 0$ and $dist(v) = \infty$ for every $v \neq s$. In each iteration, the algorithm picks the unexplored node v with the smallest associated distance, marks it as explored, and then *relaxes* every edge (v, u) by checking whether $d = dist(v) + w(v, u) < dist(u)$, and if so, updating $dist(u)$ to d. Notice that once $dist(v) \neq \infty$, it always holds the length of some path from s to u, and hence $dist(v)$ is an upper bound on the weight of the shortest $s \rightsquigarrow v$ path.

Dijkstra's algorithm can be implemented efficiently using a priority queue with a `changeKey()` operation [7]. The idea is to maintain a queue of offers, where an *offer* $\langle v, d\rangle$ indicates that there is an $s \rightsquigarrow v$ path of weight d. An offer $\langle v, d\rangle$ is enqueued by inserting an element into the queue with key d and data v. In every iteration, the algorithm extracts a minimal offer $\langle v, d\rangle$ from the queue using `extractMin()`, and for each edge (v, u) it either `insert()`s a new offer (if $dist(u) = \infty$) or uses `changeKey()` to decrease the key of the existing offer $\langle u, d'\rangle$ if $dist(v) + w(v, u) < dist(u) = d'$.

Using `changeKey()` to parallelize Dijkstra's algorithm: Dijkstra's algorithm can be parallelized by using a concurrent priority queue from which multiple threads dequeue offers and process them in parallel. However, the existing parallel algorithm must work around the lack of `changeKey()` support in prior concurrent priority queues, with adverse performance consequences. Sec. 5.1 details this problem and describes the way existing parallel SSSP algorithms work. Sec. 5.2 describes the way our adaptation of the parallel algorithm addresses

```
Graph (E,V,w)
done[1..TNum] = [false,..,false]
D[1..|V|] = [∞,..,∞]
Element[1..|V|] Offer =
                  [null,..,null]
Lock [1.. |V|]  DLock
Lock [1.. |V|]  OfferLock
```

```
relax(v,vd)
 lock(OfferLock[v])
   if (vd < D[v])
     vo = Offer[v]
     if (vo = null)
       Offer[v] = insert(v,vd)
     else
       if (vd < vo.key)
         publishOfferMP(v,vd,vo)
 unlock(OfferLock[v])
```

```
publishOfferMP(v,vd,vo)
  updated = changeKey(vo, vd)
  if (!updated and vd < D[v])
    Offer[v] = insert(v,vd)
```

```
publishOfferNoMP(v,vd)
  Offer[v] = insert(v,vd)
```

```
parallelDijkstra()
 while (!done[tid])
   o = extractMin()
   if (o ≠ null)
     u = o.data
     d = o.key
     lock(DLock[u])
     if(dist < D[u])
       D[u] = d
       explore = true
     else
       explore = false
     unlock(DLock[u])
     if (explore)
       foreach ((u,v) ∈ E)
         vd = d + w(u,v)
         relax(v,vd)
   else
     done[tid] = true
     i = 0
     while (done[i] and i<TNum)
       i =  i + 1
     if(i == TNUM)
        return
     done[tid] = false
```

Figure 6 Parallel versions of Dijkstra's SSSP algorithm: `parallelDijkstra()` is a pseudocode implementation of **ParDijk-MP**. The pseudocode of **ParDijk** can be obtained by replacing the call to `publishOfferMP()` in `relax()` with a call to `publishOfferNoMP()`.

this problem by using `changeKey()`.

Concurrent *dist* updates: Both parallel algorithms described next must guarantee that when relaxing an edge (v, u), reading $dist(v)$ and the subsequent decreasing of $dist(v)$ happen atomically. Otherwise, an update to d might get interleaved between another thread's read of $dist(v)$ and subsequent update to $d' > d$, and thus be lost. This atomicity is typically realized by performing the update with a `compare-and-swap` operation [2, 14]. Our implementations, however, use per-node locking: if a thread decides to update $dist(v)$, it acquires v's lock, verifies that $dist(v)$ should still be updated, and then performs the update. This approach allows us to atomically update an additional $P(v)$ field, which holds the predecessor node on the shortest path to v [7], and thus computes the shortest paths in addition to the distances. We omit the details of this, which are standard.

5.1 ParDijk: A Parallel version of Dijkstra's SSSP Algorithm based on a Concurrent Priority Queue

A natural idea for parallelizing Dijkstra's algorithm is to use a concurrent priority queue and thereby allow multiple threads to dequeue and process offers in parallel. Because existing concurrent priority queues do not support `changeKey()`, doing this requires adapting the algorithm to use inserts instead of `changeKey`() when relaxing edges [2, 14].

Specifically, the `changeKey`() operation, which is required to update an existing offer $\langle u, d\rangle$ to have distance $d' < d$, is replaced by an `insert`() of a new offer $\langle u, d'\rangle$. As a result, in contrast to the original algorithm, multiple offers for the same node can exist in the queue. Consequently, the parallel algorithm might perform two types of *wasted work*: (1) *Empty work* occurs when a thread dequeues an offer $\langle v, d\rangle$ but then finds that $dist(v) < d$, i.e., that a better offer has already been processed. (2) *Bad work* occurs when a thread updates $dist(v)$ to d, but $dist(v)$ is later updated to some $d' < d$.

In both cases of wasted work, a thread performs an `extractMin`() that would not need to be performed had `changeKey`() been used to update offers in-place, as in the original algorithm. This is particularly detrimental to performance because `extractMin`() operations typically contend for the head of the queue, and the wasted work increases this contention and makes *every* `extractMin`()—wasted or not—more expensive.

Procedure `parallelDijkstra()` shown in Fig. 6 provides the pseudo code of the two parallel versions of Dijkstra's SSSP algorithm that we discuss. The **ParDijk** algorithm is obtained by replacing the call to `publishOfferMP()` in `relax()` with a call to `publishOfferNoMP()`.

The algorithm records its state in several global variables: A boolean array `done` maintains for every thread t a flag `done[`i`]` which records whether the thread found work in the priority queue; an array `D` which records the current estimation of the distance to every node; and an array `Offer` of pointers to offers (elements). Intuitively, `Offer[`u`]` points to the best offer ever made to estimate the distance to node u. The two lock arrays `DLock` and `OfferLock` are use to protect write accesses to arrays `D` and `Offers`, respectively. The locks in `OfferLock` are also used to prevent multiple threads from concurrently changing the priority (distance estimation) to the same node.

When a thread removes an offer $o = (u, d)$ from the queue, it first determines whether it can use it to improve the current distance to u. If this is the case, it updates `D` and turns to exploring u's neighbors, hoping to improve the estimation of their distances too. If the distance to u cannot be shorten, the thread goes back to the queue trying to get more work to do. If the thread managed to improve the distance to u, it explores each of its neighbors v by invoking `relax(v,vd)`. The latter locks `v`'s entry in the `Offer` array, and check whether the new estimation `vd`, is better than the current estimation `D[`v`]` and from the one suggested the best offer so far `Offer[`v`]`. If this is the case, it adds a new offer to the queue. Note that this might lead to node v having multiple offers in the queue.

If the thread does not find work in the queue, i.e., `o` turns out to be *null*, the thread checks if all the other threads have not found work, and if so, terminates.

5.2 ParDijk-MP: A Parallel version of Dijkstra's SSSP Algorithm based on a Concurrent Priority Queue with Mutable Priorities

Having a concurrent priority queue which supports a `changeKey`() operation enables updating an existing offer's distance in place, and thus allows parallelizing Dijkstra's algorithm without suffering from wasted work. The change is rather minor: The **ParDijk-MP** algorithm is obtained from procedure `parallelDijkstra()` by keeping the call to `publishOfferMP()` in `relax()`. Note that `publishOfferMP()` checks whether it can update an existing offer in the queue before it tries to insert a new one. This ensures that the queue never contains more than one offer for every node, although a new offer to the same node might be added after the previous offer has been removed.

6 Experimental Evaluation

Our evaluation of CHAMP focuses on the *overall* performance of the *client application* rather than on the performance of individual core operations. To this end, we used the parallel Dijkstra's algorithms (Section 5) as benchmarks: (1) **ParDijk**, the existing parallel algorithm that may create redundant offers, and (2) **ParDijk-MP**, the version that exploits mutable priorities to update offers in-place. Of these algorithms, only ParDijk can be run with prior priority queues without mutable priorities. We compare CHAMP to SKIPLIST, a linearizable concurrent priority queue based on a nonblocking skip list, as in the algorithm of Sundell and Tsigas [25].[4] As a performance yardstick, we additionally compare to the parallel SSSP implementation of the Galois [20] graph analytics system. Galois relaxes Dijkstra's algorithm by allowing for both empty work and bad work (see Sec. 5.1). It compensates for the incurred overheads by using a highly-tuned non-linearizable priority queue, which sacrifices exact priority order in exchange for reduced synchronization overhead. We thus use Galois as a representative of the family of *relaxed* non-linearizable priority queues, such as Lotan and Shavit's quiescently consistent algorithm [17] or the SprayList [2].

Experimental setup: We use a Fujitsu PRIMERGY RX600 S6 server with four Intel Xeon E7-4870 (Westmere EX) processors. Each processor has 10 2.40 GHz cores, each of which multiplexes 2 hardware threads, for a total of 80 hardware threads. Each core has private write-back L1 and L2 caches; the L3 cache is inclusive and shared by all cores on a processor. The parallel Dijkstra algorithms and priority queues are implemented in Java and run with the HotSpot Server JVM, version 1.8.0-25. Galois is implemented in C++; we use the latest version, 2.2.1. All results are averages of 10 runs on an idle machine.

SSSP run time: We measure the running time of each tested algorithm on several input graphs, as we increase the number of threads. Each input is a random graph over 8000 vertices, in which each edge occurs independently with some probability p and a random weight between 1 and 100.[5] Figures 7(a)– 7(e) depict the results. We observe an overall trend in which all algorithms obtain speedups up to at most 20 threads, but their run time plateaus or increases with more than 20 threads. This is consistent with prior SSSP experiments on identical hardware [2]. We therefore focus our attention on the concurrency levels in which speedups are obtained.

We find that while ParDijk-MP, which leverages CHAMP's `changeKey`() operation, performs worse than ParDijk/SKIPLIST with few threads, its run time improves as the number of threads increases and it eventually outperforms ParDijk/SKIPLIST. On the $p = 1\%$ and $p = 5\%$ graphs, the best run time of ParDijk/SKIPLIST is at 10 threads, and ParDijk-MPis 20% faster than it. Furthermore, the run time of ParDijk-MP plateaus up to 20 threads, whereas ParDijk/SKIPLIST starts deteriorating after 10 threads, making ParDijk-MP $\approx 2\times$ faster than ParDijk/SKIPLIST at 20 threads. On the $p = 10\%$ and $p = 20\%$ graphs, the best run time is at 20 threads, and ParDijk-MPis 60%–80% better than ParDijk/SKIPLIST. On the $p = 80\%$ graph ParDijk-MPoutperforms ParDijk/SKIPLIST only after 20 threads, obtaining a 20% better run time. Similarly, ParDijk-MP outperforms Galois given sufficient parallelism: On the $p = 1\%$ graph ParDijk-MP is consistently about $2\times$ faster, while on the other graphs it is $1.25\times$–$2\times$ slower up to 4 threads, but as more threads are added, its run time becomes $2\times$ better than Galois.

[4] Following the methodology of Lindén and Jonsson [15], we implement a singly-linked instead of doubly-linked skip list.
[5] We use the same random weight assignment as Alistarh et al. [2, 14].

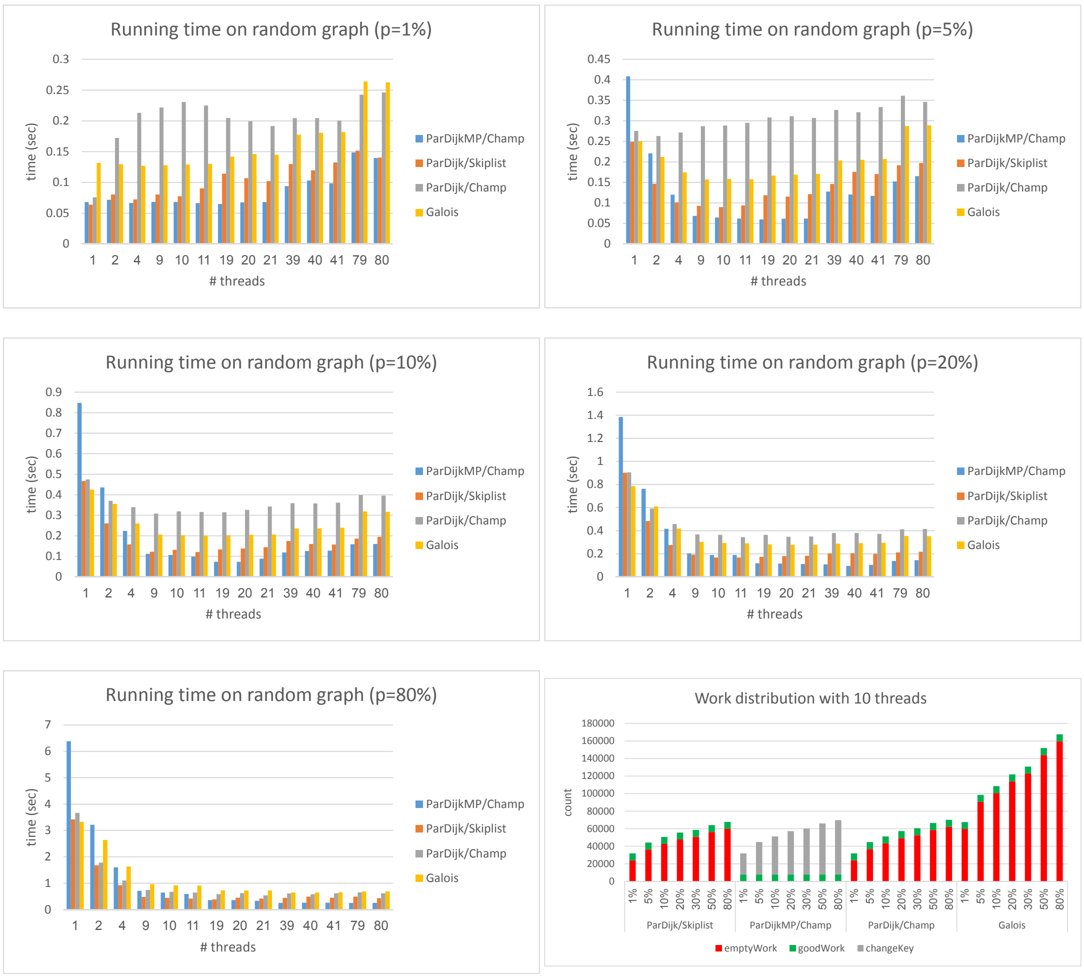

Figure 7 SSSP algorithms with different priority queues: Run time (lower is better) and work distribution.

Figure 7(f) demonstrates the reason for these results, using the 10-thread runs as an example. For each algorithm and input, we classify the work done in each iteration—i.e., for each `extractMin`()—into good work and useless empty work, in which a thread dequeues an outdated offer whose distance is greater than the current distance. (Bad work, in which a thread updated a distance not to its final value, is negligible in all experiments and therefore omitted.) For ParDijk-MP we additionally show the number of `changeKey`() operations performed. As Figure 7(f) shows, 75%–90% of the work in ParDijk and 90% of the work in Galois is useless. For ParDijk, this corresponds exactly to extraction of outdated offers that in ParDijk-MP are updated in-place using `changeKey`(). In eliminating this useless work, ParDijk-MP with Champ significantly reduces the amount of `extractMin`() operations, which—as we shortly discuss—are the least scalable operations. Note, however, that the gains ParDijk-MP obtains from eliminating the useless work are offset somewhat by Champ's inefficient core operations. We note that we got a similar work distribution when we ran the algorithm with a single thread. This indicates that the wasted work is due to the superfluous insertions and is not an artifact of concurrency.

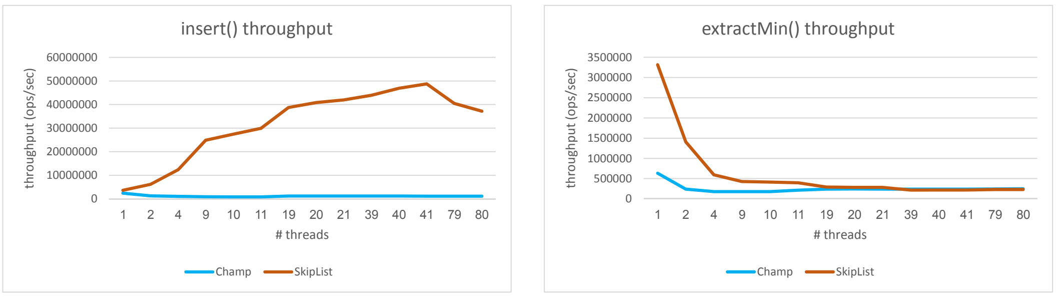

Figure 8 Basic priority queue operations performance: throughput (higher is better) of `insert()` and `extractMin()`.

Turning to ParDijk itself, we find that SKIPLIST outperforms CHAMP. This occurs because SKIPLIST's `insert()` and `extractMin()` are, respectively, more scalable and more efficient than CHAMP's. (We discuss this in detail next.) The performance gap between SKIPLIST and CHAMP shrinks as p increases and the graphs become denser. (For example, at 10 threads, ParDijk's SKIPLIST run time is $3\times$ better than with CHAMP for $p = 1\%$, $2.16\times$ better for $p = 20\%$ and $1.5\times$ better for $p = 80\%$.) The reason is that as the inputs become denser, threads perform more work—i.e., iterate over more edges—for each offer. Consequently, the priority queue's performance becomes a less significant factor in overall performance: it is accessed less frequently, and thus becomes less contended.

Core operations performance: We study the performance of the core queue operations with microbenchmarks. For `insert()`, we measure the time it takes N threads to concurrently `insert()` 10^6 items ($10^6/N$ each) into the priority queue. For `extractMin()`, we measure the time it takes N threads repeatedly calling `extractMin()` to empty a priority queue of size 10^6. Figure 8 shows the results, reported in terms of the throughput obtained (operations/second). We see that SKIPLIST `insert()` scale well, because insertions to different positions in a skiplist do not need to synchronize with each other. In contrast, every CHAMP `insert()` acquires the heap root lock, to increase the heap size and initiate a `bubbleDown`. As a result, CHAMP insertions suffer from a sequential bottleneck and do not scale. For `extractMin()`, both algorithms do not scale, since both have sequential bottlenecks in extractions. For CHAMP, it is the heap root lock again. For SKIPLIST, it is the atomic (via CAS) update of the pointer to the head of the skiplist.[6] The characteristics of the core operations explain the performance of ParDijk-MP vs. ParDijk: when updating an offer, ParDijk-MP performs a `changeKey()` where ParDijk performs an `insert()`. Both of these are scalable operations, although CHAMP's `changeKey()` may be heavier than a skiplist insertion, as it performs hand-over-hand locking. However, for an offer updated U times, ParDijk performs $U - 1$ extraneous `extractMin()`s that ParDijk-MP/CHAMP avoids. Because `extractMin()` is the most expensive and non-scalable operation, overall ParDijk-MPcomes out ahead.

Sparse vs dense graphs: In our experiments we used relatively dense graphs. When using sparse graphs like road networks, e.g., Rome99, USA-FLA, USA-NY, and USA-W [1], whose

[6] Note that SKIPLIST's `extractMin()` removes the head (minimum) skip list by first marking it logically deleted and then physically removing it from the list, and any thread that encounters a logically deleted node tries to complete its physical removal before proceeding. This causes further `extractMin()` serialization, on top of the memory contention causes by issuing CASes to the shared head pointer.

average degree is less than three, we noticed that CHAMP suffers from a slowdown of 2x-15x. We believe that the reason for this behavior is that in these scenarios there is much less wasted work (less than 11% in our experiments). Because there is so little wasted work, the competing algorithms outperform CHAMP due to their faster synchronization, which no longer gets outweighed by the execution on extraneous wasted work.

7 Related Work

Existing concurrent priority queues [2, 3, 11, 15, 16, 17, 23, 24, 25, 26] support `insert()` and `extractMin()` but not `changeKey`, and most of this prior work has focused on designing priority queues with ever more `insert()`/`extractMin()` throughput on synthetic microbenchmarks of random priority queue operations. Researchers have only recently [2, 15, 26] started evaluating new designs on priority queue client applications, such as Dijkstra's algorithm. We are, to the best of our knowledge, the first to step back and approach the question from the client application side, by considering how the `insert()`/`extractMin()` interface restricts the clients, and how to address this problem by extending the priority queue interface.

Our priority queue is based on concurrent heap of Hunt et al. [11], which we extend to support the `changeKey()` operation. We have also changed some of the design choices in [11], to better suit our applications. (See Sec. 4.3). Mound [16] also uses a heap-based structure. It minimizes swapping of heap nodes by employing randomization and storing multiple items in each heap node. It is thus not obvious how to implement `changeKey()` in Mound.

Several concurrent priority queues are based on skiplists [22]. Lotan and Shavit [17] initially proposed such a lock-based priority queue, and Sundell et al. [25] designed a nonblocking skiplist-based priority queue. In both algorithms, contention on the head of the skiplist limits the scalability of `extractMin()`. There are two approaches for addressing this bottleneck: One is to improve `extractMin()` synchronization, for example by batching node removals [15] or using combining [3]. Currently this approach does not lead to algorithms that scale beyond 20 threads [3, 15]. A second approach *relaxes* the priority queue correctness guarantees by allowing `extractMin()` to not remove the minimum priority item [2, 23, 26]. Using such algorithms requires reasoning about—and possibly modifying—the application, to make sure it can handle this relaxed behaviors. Note that all these algorithms—relaxed or not—still provide the client with only the limited set of `insert()`/`extractMin()` operations.

8 Conclusions and Future Work

We present and evaluate CHAMP, the first concurrent algorithm for a priority queue with mutable keys. CHAMP is implemented using an array-based binary heap, and consequently its core priority queue operations, `insert()` and `extractMin()`, do not scale as well as in prior designs. Despite this, we show that CHAMP's extended interface improves the performance of parallel versions of Dijkstra's SSSP algorithm, because it saves the client algorithm from *wasting work* when working around the lack of `changeKey()` support in other priority queues. This raises an interesting question for future work: can we efficiently implement mutable priorities in the more scalable skip list-based priority queues without compromising on the scalability of the core operations?

Acknowledgments. This work was funded by EU FP7 project ADVENT (308830), ERC grant agreement no. [321174-VSSC], by Broadcom Foundation and Tel Aviv University Authentication Initiative, by Israel Science Foundation (grants 652/11, 1227/10 and 1749/14), and by Yad HaNadiv foundation.

References

1 9th DIMACS implementation challenge. http://www.dis.uniroma1.it/challenge9/download.shtml.

2 Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A Scalable Relaxed Priority Queue. In *PPoPP*, 2015.

3 Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The Adaptive Priority Queue with Elimination and Combining. In *DISC*, 2014.

4 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

5 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

6 E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

7 Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *JACM*, 34(3):596–615, July 1987.

8 Richard M. Fujimoto. Parallel discrete event simulation. *CACM*, 33(10):30–53, 1990.

9 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

10 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12:463–492, July 1990.

11 Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. *IPL*, 60(3):151–157, 1996.

12 Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

13 Zvi M. Kedem and Abraham Silberschatz. A characterization of database graphs admitting a simple locking protocol. *Acta Informatica*, 16, 1981.

14 Justin Kopinsky. SprayList SSSP benchmark. `https://github.com/jkopinsky/SprayList/blob/master/sssp.c`, 2015.

15 Jonatan Lindén and Bengt Jonsson. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *OPODIS*, 2013.

16 Yujie Liu and Michael Spear. Mounds: Array-Based Concurrent Priority Queues. In *ICPP*, 2012.

17 Itay Lotan and Nir Shavit. Skiplist-Based Concurrent Priority Queues. In *IPDPS*, 2000.

18 Paul E. McKenney and John D. Slingwine. Read-copy update: using execution history to solve concurrency problems. In *PDCS*, 1998.

19 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 2004.

20 Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *SOSP*, 2013.

21 R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

22 William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *CACM*, 33(6):668–676, June 1990.

23 Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. In *SPAA*, 2015.

24 Nir Shavit and Asaph Zemach. Scalable Concurrent Priority Queue Algorithms. In *PODC*, 1999.

25 Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *JPDC*, 65(5):609–627, 2005.

26 Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The Lock-free k-LSM Relaxed Priority Queue. In *PPoPP*, 2015.