A HEAP-BASED CONCURRENT PRIORITY QUEUE WITH MUTABLE PRIORITIES FOR FASTER PARALLEL ALGORITHMS

by

Orr Tamir

under the supervision of Prof. Mooly Sagiv and Dr. Noam Rinetzky and the consultation of Dr. Adam Morrison and Prof. Yehuda Afek

A thesis submitted for the degree of Master of Science

Submitted to the Senate of Tel Aviv University
10 2015
To her and my family,
Abstract

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

Orr Tamir
School of Computer Science
Tel Aviv University

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 single source shortest path algorithm, resort to cumbersome and inefficient workarounds. 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 allows to update priorities of existing items. 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 algorithm in a more natural way, and that its implementation, using our concurrent priority queue, outperforms existing algorithms.
Acknowledgements

I would like to thank Prof. Uri Zwick and Prof. Haim Kaplan for their comments. I would also like to thank Prof. Yehuda Afek for hours of discussion of the unknown. Many thanks to Prof. Mooly Sagiv for kind guidance. Finally I would like to thank Dr. Noam Rinetzky that tolerated me for the longest time and motivated me to finish this work on the best side. Last but not least I would like to thank Dr. Adam Morrison for his help.
# Contents

1 Introduction 1  
2 Preliminaries 3  
  2.1 Notations 3  
  2.1.1 Simplifying assumptions 4  
  2.2 Shortest paths 4  
3 Priority Queues with Mutable Priorities 5  
  3.1 A Sequential Heap with Mutable Priorities 5  
  3.2 CHAMP: A Concurrent Heap with Mutable Priorities 8  
    3.2.1 Linearazability 12  
    3.2.2 Deadlock-Freedom 14  
    3.2.3 Comparison with Hunt’s Algorithm [10] 14  
4 Case Study: Parallelizing Dijkstra’s SSSP Algorithm 17  
  4.1 ParDijk: A Parallel version of Dijkstra’s SSSP Algorithm based on a Concurrent Priority Queue 18  
  4.2 ParDijk-MP: A Parallel version of Dijkstra’s SSSP Algorithm based on a Concurrent Priority Queue with Mutable Priorities 20  
  4.3 A note on Prim’s Minimal Spanning tree 21  
5 Experimental Evaluation 23  
6 Related Work 27  
7 Conclusions and Future Work 29  
Bibliography 32
List of Figures

3.1 The data representation of the heap. ......................................................... 6
3.2 Auxiliary procedures. ................................................................................ 6
3.3 Pseudo code of a sequential heap with mutable priorities ............................ 7
3.4 The pseudo code of CHAMP, a concurrent priority queue with updatable key based on a binary heap ................................................................. 9
3.5 Concurrent heapifying procedures. ............................................................. 10
3.6 A concurrent binary heap snapshot ............................................................ 12
4.1 Parallel versions of Dijkstra’s SSSP algorithm ............................................ 19
4.2 Parallel versions of Prim’s MST algorithm ................................................ 22
5.1 SSSP algorithms with different priority queues ........................................ 24
5.2 Basic priority queue operations performance ......................................... 26
Chapter 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 among the most useful data structures in practice, and can be found in a variety of applications ranging from graph algorithms [5, 20] to discrete event simulation [7] and modern SAT solvers [4]. The importance of priority queues has motivated many concurrent implementations [1, 2, 10, 14–16, 21–24]. 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 [3, 5], need to update the priority of an item after its insertion, i.e., *mutable priorities*. Parallelizing these algorithms requires working around the lack 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 [1]—all of which impose overhead. Sequential heap-based priority queues support mutable priorities [3], but concurrent heaps have been abandoned in favor of skiplist-based designs [16] whose extractMin() and insert() are more efficient and scalable. 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 linearizable [9] Concurrent Heap with Mutable Priorities, which provides a changePri() 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 effi-

\(^1\)In this thesis, we consider lower \(p\) values to mean higher priorities.
cient changePri() operation improves the client overall performance by saving it from doing wasted work—the overhead that arises when working around the lack of changePri() 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 [10] to support the changePri() operation.
2. We convert an existing parallel SSSP algorithm to utilize the changePri() 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.

**Outline** In chapter 2 we discuss the preliminaries for this thesis. Chapter 3 presents the sequential implementation and introduces the concurrent implementation. In chapter 4 we show how to use our new implementation in the single source shortest path problem. Chapter 5 evaluates the ideas presented in the previous chapter. Related work is discussed in 6 and we conclude in 7.
Chapter 2

Preliminaries

In this chapter we fix notations and briefly discuss certain properties of the single source shortest path problem in graphs. We also present our running example that is used throughout this thesis.

2.1 Notations

In the rest of the thesis, we assume a fixed graph with positive weights $G = \langle V, E, w \rangle$ and a given root $s \in V = \{v_1, \ldots, v_k\}$. Given an edge $e = \langle p, n \rangle \in E$, we say that $e$ leaves $p$ and enters $n$, and refer to $p$, $n$, and $w(p, n)$ as $e$’s source, target and weight. We say that there is a path in $G$ from node $u$ to node $v$ if it is possible to construct a sequence of distinct edges $\pi = e_{i_1} \ldots e_{i_n}$ such that $e_{i_1}$ leaves $u$, $e_{i_n}$ enters $v$, and the target of $e_{i_j}$ is the source of $e_{i_{j+1}}$ for every $j = 1, \ldots, n - 1$. We refer to the source of $e_{i_1}$ as the source of $\pi$ and to the target of $e_{i_n}$ as its target. We say that $p$ precedes $n$ in $\pi$ and that $n$ follows $p$ in $\pi$ if $\pi$ contains the edge $\langle p, n \rangle$. We denote the set of paths in $G$ by $\text{paths}(G)$, the set of paths in $G$ going from $u$ to $v$ by $\text{paths}(G, u, v)$, and the set of nodes that appear in the edges of $\pi$ by $\text{nodes}(\pi)$. We denote by $\pi \pi'$ the path obtained by extending $\pi$ with $\pi'$ and by $\pi \langle p, n \rangle$ the path obtained by extending $\pi$ with the edge $\langle p, n \rangle$.

The length of a path $\pi$, denoted by $|\pi|$, is the sum of the weights of its edges. A path in $G$, from $u$ to $v$ is minimal if its length is equal or lower to that of any other path in $G$ from $u$ to $v$. The set of shortest paths from $u$ to $v$ is denoted by $\text{sp}(u, v)$. The shortest distance from $u$ to $v$, denoted by $d(u, v)$ is the length of a minimal path between from $u$ to $v$ or $\infty$ if no such path exists. The distance of $v$, denoted by $d(v)$, is its shortest distance from $s$. We say that $v$ is reachable if $d(v) < \infty$. A path is a minimal path (or shortest path) to $v$ if it is a minimal path from $s_0$ to $v$. If $d(v) < d(u)$ we state that $v$ is closer than $u$. 

3
2.1.1 Simplifying assumptions

To simplify the presentation, we add to $G$ a sentinel node $s_0 \notin V$ and a 0-weighted edge $(s_0, s)$. In the following, we use $V_0 = V \uplus \{s_0\}$, $E_0 = E \uplus \{(s_0, s)\}$ and $w_0 = w[<s_0, s> \rightarrow 0]$ as shorthands. We also use _ to denote an irrelevant values. In the following, we also make the implicit assumption that all the nodes in the graph are reachable from $s_0$. This assumption can be lifted by changing the termination condition of the algorithm. We note that in our experiments all nodes are reachable.

2.2 Shortest paths

The correctness of our algorithms stems from two general closure properties of shortest paths. The first property, stated formally below, is rather immediate: any prefix of a shortest path is also a shortest path.

$$P_{2.1} \overset{\text{def}}{=} \forall \tau, u, n. \tau \langle u, n \rangle \in sp(s_0, n) \implies \tau \in sp(s_0, u). \quad (2.1)$$

The second property highlights a certain relation between shortest paths of increasing distances.

$$P_{2.2} \overset{\text{def}}{=} \forall U \subset V_0, u \in U, v \in V \setminus U, \ (u, v) \in E_0, \pi \in sp(s_0, u).$$

$$\left(\max(d(U)) \leq \min(d(V \setminus U)) \land d(u) + w_0(u, v) = \min(\{d(u') + w_0(u', v') \mid u' \in U, v' \in V \setminus U\})\right)$$

$$\implies \pi \langle u, v \rangle \in sp(v). \quad (2.2)$$

Here, $d(W) = \{d(w) \mid w \in W\}$ denotes the set of distances to the nodes in $W$ and $w_0(u, v)$ is undefined if $(u, v) \notin E_0$. Informally, $P_{2.2}$ says that given a set of nodes $U$ such that the distance of any node in $U$ is equal to or smaller than the distance of any node not in $U$, one can determine the shortest paths to the closet nodes to $s_0$ which are not in $U$ by only considering the shortest paths leading to nodes in $U$ and the edges going from $U$ to $V$. More specifically, any path $\pi$ from $s_0$ to a node $v \notin U$ which ends with an edge leaving a node $u \in U$ and whose length is equal to or lower than any other such path is a shortest path to $v$. Another direct conclusion from $P_{2.2}$ is that nodes shortest paths are discovered in monotone increasing order (this order is also unique if no two nodes have the same cost), this captures the serial notion of Dijkstra’s algorithm. The second part of $P_{2.2}$ determines when an edge is a valid extension of a shortest path. This part also drives us to attempt to compute this minimum as soon as possible. Informally, the main idea behind our parallel algorithms is to find such an extension.

Together, these properties indicate that one can iteratively compute the shortest paths to every node in a non-decreasing order of distances. They also highlight the inherently sequential nature of this kind of algorithms as the distance to a node $v$ cannot be computed before the distances to all nodes that are closer to $s_0$ than $v$ are determined.
Priority Queues with Mutable Priorities

A priority queue with mutable priorities (PQMP) is a data structure for maintaining a multiset $A$ of elements, where each element is a pair comprised of a data item $d$ and a value $k$ called priority.\footnote{The term key is sometimes used instead of priority.} A PQMP supports the following operations:

- **extractMin()**: Removes and returns the element which has the highest priority in the queue.\footnote{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.
- **changePri($e$, $k$)**: 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 [3, Ch. 6.5], allows to store in queue multiple elements with the same data item.

### 3.1 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 than or equal to the keys of its children, if they exist [3]. 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 \( \text{extractMin}() \), \( \text{peek}() \), \( \text{insert}() \), and \( \text{changePri}() \) 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. 3.1 defines the type \( \text{Element} \), and shows the data representation of a heap using two global variables: An array \( A \) and an integer \( \text{Last} \). (Array \( L \) and the \( \text{up} \) field in elements are used only by the concurrent algorithm and are described in section 3.2) A heap with maximal capacity \( \text{Length} \) is comprised of an array \( A \) of pointers to \( \text{Element}s \) with \( \text{Length}+1 \) entries and a counter \( \text{Last} \) 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: \( \text{key} \) keeps the element’s priority, \( \text{data} \) stores an application-specific data item, and \( \text{pos} \) 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 \( e.\text{pos} \neq -1 \) then \( A[e.\text{pos}] = e \).

Figure 3.3 shows the pseudocode of a sequential heap. The operations use the auxiliary functions defined in Fig. 3.2. 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 \( \_\text{seq} \) subscript to distinguish between the sequential operations and the concurrent ones.

- \( \text{peek}_\text{seq}() \): Returns the root element or \( \text{null} \) if the heap is empty.

- \( \text{insert}_\text{seq}(d,k) \): Returns \( \text{null} \) if the heap is \( \text{full} \). Otherwise, it allocates and inserts a new

![Figure 3.1: The data representation of the heap.](image)

![Figure 3.2: Auxiliary procedures.](image)
3.1. A Sequential Heap with Mutable Priorities

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

1. `extractMin_seq()`
2. `min = A[1]`
3. `ls = Last`
4. `if (ls = 0)`
5. `return null`
6. `min.pos = -1`
7. `if (ls = 1)`
9. `else`
11. `A[1].pos = 1`
13. `if (ls = 2)`
14. `Last = 1`
15. `else`
16. `Last = ls - 1`
17. `bubbleDown_seq(A[1])`
18. `return min`

1. `insert_seq(key, data)`
2. `if (Last = Length)`
3. `return null`
4. `e = new Element(key, data, Last+1)`
5. `if (Last = 0)`
7. `Last = 1`
8. `else`
9. `lock(L[Last + 1])`
10. `e.up = true`
12. `Last = Last + 1`
13. `bubbleUp(e)`
14. `return e`

1. `changeKey_seq(e, k)`
2. `if (e.pos ≤ [1..Last])`
3. `return false`
4. `if (k < e.key)`
5. `e.key = k`
6. `bubbleUp_seq(e)`
7. `else if (k > e.key)`
8. `e.key = k`
9. `bubbleDown_seq(e)`
10. `return true`

1. `bubbleDown_seq(e)`
2. `min = e.pos`
3. `do`
4. `i = min`
5. `l = leftChild(i)`
6. `r = rightChild(i)`
7. `if (l ≤ Last)`
8. `if (A[l].key < A[i].key)`
9. `min = l`
11. `min = r`
12. `swap(i, min)`
13. `while (i ≠ min)`
14. `swap(i, par)`
15. `i = par`

Figure 3.3: Pseudo code of a sequential heap with mutable priorities. Length-1 is the capacity of the heap. We assume that `changeKey_seq(e, k)` is invoked with `e≠null`.
element into the heap. The element is placed at the \( \text{Last} + 1 \) 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, it returns a pointer to the new element as its tag.

- **extractMin\(_{\text{seq}}\)**: Returns \textit{null} if the heap is \textit{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, the operation returns the previous root element.

- **changePri\(_{\text{seq}}\)**: 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, **changePri\(_{\text{seq}}\)** 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 \(-1\).

### 3.2 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 like the sequential algorithm described in the previous section. Synchronization is achieved using a fine-grained locking protocol, derived from the one used in [10] (see Sec. 3.2.3.)

CHAMP is implemented using the global variables shown in Fig. 3.1. Variables \( A \) and \( \text{Last} \) play the same role as in the sequential setting (see Sec. 3.1.) 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 \textit{root lock}, is also used to protect the \( \text{Last} \) variable. A thread is allowed to modify a shared memory location only under the protection of the appropriate lock. Read accesses to the \textit{entries} of array \( A \) and to variable \( \text{Last} \) should also be protected by a lock. In contrast, fields of elements can be read without a lock.\(^2\)

Figures 3.4 and 3.5 show a 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

\(^2\)In this paper, we sidestep the difficult problem of concurrent safe memory reclamation [8,17,18], and assume that memory is recycled either by the client or by an underlying garbage collector [11].
3.2. **CHAMP: A CONCURRENT HEAP WITH MUTABLE PRIORITIES**

Figure 3.4: 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. 3.5. Auxiliary procedures, e.g., swap(), are defined in Fig. 3.2.
```
1 bubbleDown(e)
2 min = e.pos
3 do
4    i = min
5    l = LeftChild(i)
6    r = RightChild(i)
7    if (l \leq \text{Last})
8       lock(L[l])
9       lock(L[r])
10      if (A[l] != null)
11         if (A[l].key < A[i].key)
12            min = l
13      end
14      end
15      if (A[r] != null and
16          A[r].key < A[min].key)
17         min = r
18      end
19      if (i \neq min)
20         swap(i, min)
21     end
22     unlock(L[i])
23 while (i \neq min)
24 unlock(L[i])

1 bubbleUp(e)
2 i = e.pos
3 iLocked = true
4 parLocked = false
5 while (l < i)
6    par = Parent(i)
7    parLocked = tryLock(L[par])
8    if (parLocked)
9       if (!A[par].up)
10          if (A[i].key < A[par].key)
11             swap(i, par)
12          end
13          A[i].up = false
14          unlock(L[i])
15          unlock(L[par])
16          return
17      else
18         unlock(L[par])
19         parLocked = false
20      end
21 iLocked = false
22 if (parLocked)
23    i = par
24    iLocked = true
25 else
26    iLocked = lockElement(e)
27    i = e.pos
28    e.up = false
29 if (iLocked)
30    unlock(L[e.pos])
```

Figure 3.5: Concurrent heapifying procedures.
it requires and inspects, adds, removes, or changes the shared state. Then, it invokes \texttt{bubbleUp}(e) or \texttt{bubbleDown}(e) to \textit{locally} restore the heap property.

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

- \texttt{peek}(): Although \texttt{peek}() only reads a single memory location, it starts by taking the root lock. This is required because another thread might perform an \texttt{insert}() operation concurrently, which could lead to a state where the key of the root is \textit{not} lower than that of its children. Returning such a root element would violate linearizability (see Sec. 3.2.1).

- \texttt{insert}(), \texttt{extractMin}(), and \texttt{changePri}(): The first part of these operations is the same as that of their sequential counterparts, but with two exceptions:
  
  **Element locking.** The operations begin by taking locks. \texttt{changePri}_{\text{seq}}(e,k) takes the lock of the array entry pointing to \( e \). All other operations grab the root lock. Also, the operations avoid calling the heapifying procedures in cases where the \textit{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.** \texttt{insert}(e) and \texttt{changePri}(e) set the \textit{up} flag of \( e \) before invoking \texttt{bubbleUp}(e). This indicates that \( e \) is being propagated up the heap, which helps synchronize concurrent \texttt{bubbleUp}() operations, as we shortly explain.

The second part of every operation \textit{locally} restores the heap property using \texttt{bubbleUp}() and \texttt{bubbleDown}(). The two shift elements up, respectively, down the heap until the heap property is \textit{locally} restored: \texttt{bubbleUp}(e) stops when the key of \( e \) is bigger than that of its parent. \texttt{bubbleDown}(e) stops when the key of \( e \) is smaller than the keys of its children. Both operations won’t continue if \( e \) was extracted from the heap.

Both \texttt{bubbleDown}() and \texttt{bubbleUp}() employ the hand-over-hand locking protocol \cite{12} (also known as the \textit{tree-locking} protocols), but they acquire the locks in different orders: \texttt{bubbleDown}() takes the lock of the children of a node \( e \) only after it holds the lock of \( e \) while \texttt{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, \texttt{bubbleDown}() takes locks using \texttt{tryLock}() instead of \texttt{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 \texttt{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 \texttt{bubbleDown}() operation. In fact, the element might have been removed from the heap all together. The auxiliary procedure \texttt{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. \texttt{lockElement}(e) loops until it either obtained the lock protecting \( e \)'s position, or until it find that \( e \)
Figure 3.6: A concurrent binary heap snapshot

has been extracted from the heap, indicated by having value $-1$ in its position field.

Figure Fig. 3.6 depicts a state of a heap that might occur when it is manipulated by three thread, $t_4$, $t_2$, and $t_{10}$. The heap is depicted as a binary tree where the nodes, representing Elements, are depicted as are with the values of their keys. In this example, thread $t_2$ is bubbling-up element 2 (located at entry 2 of the heap) and thread $t_4$ is bubbling-up element 4, located at entry 4. Thread $t_{10}$ is bubbling-down element 10. The figure shows a tricky synchronization scenario that might occur when a bubbleUp(4) operation $t_4$ manages to lock an entry 2 (This might occur if the operation $t_2$ bubbling-up 2 has not managed to bring 2 to a position where its value is bigger than that of its parent, but had to release its locks to prevent a possible deadlock with $t_{10}$). 4 key is bigger than that 2, hence $t_4$ might come to the wrong conclusion that it managed to restore the heap-property and terminates. However, the property was restored with respect to an element, 2, 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. In our example, $t_4$ would give up its locks until $t_2$ or $t_{10}$ manage to swap 2 with 10.

Note. Our algorithm does not support concurrent priority changes of a given element $e$. (Changing the priority of different elements concurrently is of course allowed.) In the presented pseudocode, we do not use synchronization to enforce this requirement. Instead, we assume that the client threads are responsible to prevent this form of concurrent priority changes. We made this assumption because it holds in the realistic clients that we use and we wished to prevent the overhead of unnecessary synchronizations. An alternative, could have been, e.g., to serialize updates by having a per-node update lock.

### 3.2.1 Linearizability

Champ is a linearizable [9] priority queue with mutable key. Intuitively, linearizability means that every operation seems to take its effect instantaneously at some point between its invocation and response. In our case, these linearization points are as follows:

1. peek(), extractMin(), and insert(): The point in time when the operation obtained the
root lock.
2. `insert()` and `changePri(e)` which decreased the priority of `e`: The linearization point happens during the call to `bubbleUp(e)`. It is the end of the last swap of `e` before its `up` field was set to false.
3. `changePri(e)` which did 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 in the first line of `changePri(e)`.

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

(i) No element is stored in position 0.
(ii) The entries $A[1]..A[\text{Last}]$ contain distinct non-null values, except perhaps during the first part of `extractMin()`, when $A[1]$ and $A[\text{Last}]$ might be equal.
(iii) The value of every entry $A[\text{Last}+1]..A[\text{Length}]$ is null, except perhaps during the first part of `insert()` when $A[\text{Last}+1]$ might have the same non-null value as $A[1]$.
(iv) The position field $\text{pos}$ of an element agrees with its position in the heap, i.e., if $e.\text{pos} = i \land 0 < i$ then $A[e].\text{pos} = i$, 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, $A[i].\text{up} = false$ and $A[j].\text{up} = false$ then $A[j].\text{key} \leq A[i]$.

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. Assuming there are no two actions preforming `changePri(e)` on the same element, there are some pairs of non commutative operations: `extractMin()` or `peek()` and `insert()` or `changePri(e)`. If the first is `peek()` than we have a race on the lock of the root and the operation which wins the race is linearized before the other. The second case is that the first is `extractMin()`. Here we have several cases, the first one is that the second action is `insert()` or `changePri(e)` that reduces the element key thus moving it up. If the `extractMin()` action got the element of the second action then our definition of linearization points states that the `insert()` or `changePri(e)` happened before the `changePri(e)` action since the `insert()` or `changePri(e)` has no more swaps to pre-
form since its element is the root. The dual case is the extractMin() got element that is not the element of the insert() or changePri(e). In this case we can observe that our definitions states the that the insert() or changePri(e) has at least on swap to preform thus the element is not at the root position. The last case to verify is the extractMin() action with a changePri(e) that increases element key action. Here we have a race on the first action to lock e if its the root (if not then the actions are commutative), the winner of the race happened before the other as our definition of linearaziation points states.

3.2.2 Deadlock-Freedom

CHAMP is a 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.

3.2.3 Comparison with Hunt’s Algorithm [10]

Our priority queue is based on concurrent heap of Hunt et al. [10], which also uses fine-grained locking to synchronize bottom-up insert()s and top-down extractMin()s. The main difference is the addition of the changePri() operation. There are also some subtle differences between our algorithm and theirs.

- We use a different technique to prevent deadlocks between concurrent up-going bubbleUp()s and down-going bubbleDown()s: In [10], 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 changePri() take their locks in reverse order, thus possibly saving some redundant unlock() and relock() operations. Deadlock is prevented using tryLock()s operations as explained in Sec. 3.2.2.
- In both techniques, an element e bubbled up the heap might change its position due to a down-going operation. In [10], 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 changePri(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 [10], and once the reason for the non-linearizable behavior is known, changing the algorithm to be linearizable is rather easy.
CHAPTER 3. PRIORITY QUEUES WITH MUTABLE PRIORITIES
Chapter 4

Case Study: Parallelizing Dijkstra’s SSSP Algorithm

Important applications of priority queues, such as Dijkstra’s single-source shortest path (SSSP) algorithm [3, 5] and Prim’s minimal spanning tree (MST) algorithm [3, 20] need to update the priority of an item after its insertion, i.e., they require mutable priorities. In this work, we close the interface gap between sequential and concurrent priority queues by importing the changePri() 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 changePri().

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 \( s \leadsto u \) path as \( v \)'s distance from \( s \). The asymptotically fastest known sequential SSSP algorithm for arbitrary directed graphs with unbounded non-negative weights is Dijkstra’s algorithm [5, 6].

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, \( \text{dist}(v) \), which we represent 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 \( \text{dist}(s) = 0 \) and \( \text{dist}(v) = \infty \) 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 \( d = \text{dist}(v) + w(v, u) < \text{dist}(u) \), and if so, updating \( \text{dist}(u) \) to \( d \). Notice that once \( \text{dist}(v) \neq \infty \), it always holds the length of some path from \( s \) to \( u \), and hence \( \text{dist}(v) \) is an upper bound on the weight of the shortest \( s \leadsto v \) path.

Dijkstra’s algorithm can be implemented efficiently using a priority queue with a changePri() operation [6]. The idea is to maintain a queue of offers, where an offer \( (v, d) \) indicates that there is an \( s \leadsto v \) path of weight \( d \). An offer \( (v, d) \) 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 changePri() to decrease the key of the existing offer $\langle u, d' \rangle$ if $dist(v) + w(v, u) < dist(u) = d'$.

**Using changePri() 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 changePri() support in prior concurrent priority queues, with adverse performance consequences. Sec. 4.1 details this problem and describes the way existing parallel SSSP algorithms work. Sec. 4.2 describes the way our adaptation of the parallel algorithm addresses this problem by using changePri().

**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 [1, 13]. 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 $v$ [6], and thus computes the shortest paths in addition to the distances. We omit the details of this, which are standard.

### 4.1 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 changePri(), doing this requires adapting the algorithm to use insert() instead of changePri() when relaxing edges [1, 13].

Specifically, the changePri() 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 changePri() 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
4.1. **ParDijk**: A parallel version of Dijkstra’s SSSP algorithm based on a Concurrent Priority Queue

ParDijk: A Parallel Version of Dijkstra’s SSSP Algorithm Based on a Concurrent Priority Queue

1. Graph \((E, V, w)\)
2. done \([1..TNum]\) = \([\text{false} \ldots \text{false}]\)
3. \(D[1..|V|] = [\infty \ldots \infty]\)
4. Element \([1..|V|]\) Offer = \([\text{null} \ldots \text{null}]\)
5. Lock \([1..|V|]\) DLock
6. Lock \([1..|V|]\) OfferLock
7. \(\text{parallelDijkstra}()\)
8. \(\text{while} (!\text{done}[\text{tid}])\)
9. \(o = \text{extractMin}()\)
10. \(\text{if} (o \neq \text{null})\)
11. \(u = o.\text{data}\)
12. \(d = o.\text{key}\)
13. \(\text{lock}(D\text{Lock}[u])\)
14. \(\text{if} (\text{dist} < D[u])\)
15. \(D[u] = d\)
16. \(\text{explore} = \text{true}\)
17. \(\text{else}\)
18. \(\text{explore} = \text{false}\)
19. \(\text{unlock}(D\text{Lock}[u])\)
20. \(\text{if} (\text{explore})\)
21. \(\text{foreach} ((u, v) \in E)\)
22. \(vd = d + w(u, v)\)
23. \(\text{relax}(v, vd)\)
24. \(\text{else}\)
25. \(\text{done}[\text{tid}] = \text{true}\)
26. \(\text{while} (\text{done}[\text{tid}] \text{ and } i < \text{TNum})\)
27. \(\text{if} (!\text{done}[i])\)
28. \(\text{done}[i] = \text{false}\)
29. \(\text{else}\)
30. \(i = i + 1\)

Figure 4.1: Parallel versions of Dijkstra’s SSSP algorithm: \text{parallelDijkstra}() is a pseudocode implementation of \text{ParDijk-MP}. The pseudocode of \text{ParDijk} can be obtained by replacing the call to \text{publishOfferMP}() in \text{relax()} with a call to \text{publishOfferNoMP}().
head of the queue, and the wasted work increases this contention and makes every \texttt{extractMin()}—wasted or not—more expensive. Note that we still use a simple bookkeeping of the best offer for each node in the queue as the original Dijkstra thus eliminating some of the wasted work.

Procedure \texttt{parallelDijkstra()} shown in Fig. 4.1 provides the pseudo code of the two parallel versions of Dijkstra’s SSSP algorithm that we discuss. The \texttt{ParDijk} algorithm is obtained by replacing the call to \texttt{publishOfferMP()} in \texttt{relax()} with a call to \texttt{publishOfferNoMP()}.

The algorithm records its state in several global variables: A boolean array \texttt{done} maintains for every thread \( t \) a flag \texttt{done}[,\texttt{flag}] which records whether the thread found work in the priority queue; an array \texttt{D} which records the current estimation of the distance to every node; and an array \texttt{Offer} of pointers to offers (elements). Intuitively, \texttt{Offer}[\texttt{u}] points to the best offer ever made to estimate the distance to node \( u \). The two lock arrays \texttt{DLock} and \texttt{OfferLock} are use to protect write accesses to arrays \texttt{D} and \texttt{Offers}, respectively. The locks in \texttt{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 \texttt{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. Here we use property 2.2, if the discovered distance is the shortest path, we need to look again on its neighbours. If the thread managed to improve the distance to \( u \), it explores each of its neighbours \( v \) by invoking \texttt{relax}(v, vd). The latter locks \( v \)'s entry in the \texttt{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 \texttt{Offer}[\texttt{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 \texttt{null}, the thread checks if all the other threads have not found work, and if so, terminates.

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

Having a concurrent priority queue with \texttt{changePri()} supports enables updating a existing offer’s distance in place, and thus allows parallelizing Dijkstra’s algorithm without suffering from wasted work. The change is rather minor: The \texttt{ParDijk-MP} algorithm is obtained from procedure \texttt{parallelDijkstra()} by keeping the call to \texttt{publishOfferMP()} in \texttt{relax()}. Note that \texttt{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.
4.3 A note on Prim’s Minimal Spanning tree

Prim’s minimal spanning tree algorithm is very similar to Dijkstra’s single source shortest path algorithm. The only difference is the weight inserted into the priority queue. Changing line 24 in parallelDijkstra() to be \( v_d = w(u,v) \) will result in a variant of Prim’s minimal spanning tree algorithm (code is presented in figure 4.2). The resulted variants of Prim algorithms have almost identical running time to the Dijkstra variants presented earlier. Therefore we will not present their evaluation.
1 Graph \((E, V, w)\)
2 done[l..TNum] = [false ... ,false]
3 \(D[1..|V|] = [\infty ... , \infty]\)
4 Element [1..|V|] Offer =
5 [null ... , null]
6 Lock [1..|V|] DLock
7 Lock [1..|V|] OfferLock

\[\text{parallelPrim()}\]

9 \(\text{while (!done[tid])}\)
10 o = \text{extractMin()}\)
11 if (o \neq \text{null})
12 u = o.data
13 d = o.key
14 lock(DLock[u])
15 if \((\text{dist} < D[u])\)
16 \(D[u] = d\)
17 \(\text{explore} = \text{true}\)
18 else
19 \(\text{explore} = \text{false}\)
20 unlock(DLock[u])
21 if (explore)
22 foreach \((u,v) \in E)\)
23 \(vd = w(u,v)\)
24 \(\text{relax}(v,vd)\)
25 else
26 done[tid] = true
27 while (done[tid] and i<TNum)
28 if (!done[i])
29 \(\text{done}[i] = \text{false}\)
30 else
31 \(i = i + 1\)

\[\text{relax}(v,vd)\]

1 \(\text{lock(OfferLock[v])}\)
2 \(\text{if (vd < D[v])}\)
3 \(\text{vo = Offer[v]}\)
4 \(\text{if (vo = null)}\)
5 \(\text{Offer[v] = insert(v,vd)}\)
6 \(\text{else}\)
7 \(\text{if (vd < vo.key)}\)
8 \(\text{publishOfferMP(v,vd,vo)}\)
9 \(\text{unlock(OfferLock[v])}\)

\[\text{publishOfferMP(v,vd,vo)}\]

1 \(\text{updated = changePri(vo, vd)}\)
2 \(\text{if (!updated and vd < D[v])}\)
3 \(\text{Offer[v] = insert(v,vd)}\)
4 \(\text{publishOfferNoMP(v,vd)}\)
5 \(\text{Offer[v] = insert(v,vd)}\)
6 \(\text{updated = changePri(vo, vd)}\)
7 \(\text{if (!updated and vd < D[v])}\)
8 \(\text{Offer[v] = insert(v,vd)}\)
9 \(\text{publishOfferNoMP(v,vd)}\)
10 \(\text{Offer[v] = insert(v,vd)}\)

Figure 4.2: Parallel versions of Prim’s MST algorithm: \text{parallelPrim()} is a pseudocode implementation of \text{ParDijk-MP}. The pseudocode of \text{ParDijk} can be obtained by replacing the call to \text{publishOfferMP()} in \text{relax()} with a call to \text{publishOfferNoMP()}.
Chapter 5

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 use the parallel Dijkstra’s algorithms (Section 4) 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 [23].¹ As a performance yardstick, we additionally compare to the parallel SSSP implementation of the Galois [19] graph analytics system. Galois relaxes Dijkstra’s algorithm by allowing for both empty work and bad work (see Sec. 4.1). 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 [16] or the SprayList [1].

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

¹Following the methodology of Lindén and Jonsson [14], we implement a singly-linked instead of doubly-linked skip list.
Figure 5.1: SSSP algorithms with different priority queues: Run time (lower is better) and work distribution.
occurs independently with some probability \( p \) and a random weight between 1 and 100.\(^2\) Figures 5.1a–5.1e 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 [1]. We therefore focus our attention on the concurrency levels in which speedups are obtained.

We find that while ParDijk-MP, which leverages CHAMP’s `changePri()` 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-MP is 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-MP is 60%–80% better than ParDijk/SKIPLIST. On the \( p = 80\% \) graph ParDijk-MP outperforms 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.

Figure 5.1f 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 `changePri()` operations performed. As Figure 5.1f 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 `changePri()`. 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.

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.

\(^2\)We use the same random weight assignment as Alistarh et al. [1, 13].
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 5.2 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.\(^3\) The characteristics of the core operations explain the performance of ParDijk-MP vs. ParDijk: when updating an offer, ParDijk-MP performs a changePri() where ParDijk performs an insert(). Both of these are scalable operations, although CHAMP’s changePri() may be more expensive 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-MP comes out ahead.

\(^3\)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 encounter 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.
Chapter 6

Related Work

In [1] a scalable priority queue with relaxed ordering semantics is presented. Starting from a nonblocking SkipList, the main innovation behind there design is that the DeleteMin operations avoid a sequential bottleneck by "spraying" themselves onto the head of the SkipList list in a coordinated fashion.

[2] presents a priority queue with a new elimination algorithm which further increases concurrency on balanced workloads with similar numbers of add() and removeMin() operations.

[10] describes a priority queue that supports top-down extractMin() and bottom-up insert() actions. The implementation uses fine grained locking.

[14] presents a linearizable, lock-free, concurrent priority queue algorithm, based on skiplists, which minimizes the contention for shared memory that is caused by the DeleteMin operation. The main idea is to minimize the number of global updates to shared memory that are performed in one DeleteMin.

[15] introduces a concurrent data structure called the mound. The mound is a rooted tree of sorted lists that relies on randomization for balance. It supports $O(\log(\log(N)))$ insert and $O(\log(N))$ extractMin operations, making it suitable for use as a priority queue.

[16] bases the design of concurrent priority queues on the probabilistic skiplist data structure, rather than on a heap. They show that a concurrent skiplist structure, following a simple set of modifications, provides a concurrent priority queue with a higher level of parallelism and significantly less contention than the fastest known heap-based algorithms.

[21] presents a simple, concurrent data structure that approximates the behaviour of a priority queue and that gives very good performance guarantees. They also discuss models for the semantics of relaxed priority queues and introduce a technique for "waitfree locking" that allows to convert sequential data structures to relaxed concurrent data structures.

[22] addresses the problem of designing bounded range priority queues that is queues that support a fixed range of priorities. They also present two simple new algorithms LinearFunnels and FunnelTree that provide true scalability throughout the concurrency range.
[23] presents an efficient and practical lock-free implementation of a concurrent priority queue. The algorithm is based on the randomized sequential list structure called Skiplist, and a real-time extension of the algorithm is also described.

[24] presents a new, lock-free priority queue that relaxes the delete-min operation so that it is allowed to delete any of the p+1 smallest keys, where p is a runtime configurable parameter. Additionally, the behaviour is identical to a non-relaxed priority queue for items added and removed by the same thread. The priority queue is built from a logarithmic number of sorted arrays in a way similar to log-structured merge-trees.

All the above support `insert()` and `extractMin()` but not `changePri`, 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 [1, 14, 24] 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. [10], which we extend to support the `changePri()` operation. We have also changed some of the design choices in [], to better suit our applications. (See Sec. 3.2.3).
Chapter 7

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 changePri() 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?
Bibliography


הכרות

תורי עדיפויות מקבילים לא מתטרםино שוני של מתוחכם של אייברימ לאחור. התוכנות, יכולות ממוקמים והיכולים החסומים הם קבוליים שטרים צירים את היכולים והם האלגוריתמים של דיקסטרה למסלולים קצרים בוור לייצורים 앙' נلزم לבצר وعن כפיים לא יעילים. המרחק אחור תור עדיפויות ייעל יותר באופי ארור גורם ליבצע עמים.

ה>()) (CHAMP) המבוסס על תורי עדיפויות

העיקרון המניע את העבודה זה הוא ההכון הבנתי תורי התוכנות בתוכנה הבציועים של תורי עדיפויות זה ביצועים פקיעים לתוך בתוכנה שאלגוריתמים איפול מתוכנו שטי תורי עדיפויות 

(יהי נוכםبوت החרושת) אחוזי. הפעולה עיקרו זה כלי לשורג תורי עדיפויות

ה) (CHAMP) המבוסס על תורי עדיפויות

הмפרט העניין של התוכן האלגוריתם שנולakens תורי עדיפויות

שידייקסטרה בצרור תניית יותר בעעודת התוכנה בשיצור. המימונים בצרור

התוכנה הוחזקה מימונים שעידועים בתרימ יוזם אחורי.
תור עדיפויות Mbps עירמה הכלל
אפשритор לשניים מתוחת עבורה
מקבול אלגוריתמים מהיר

היבור זה הוגש חורף מחדרישות ל牒לי התואר
"מוסמר האוניברסיטה" (M.Sc.)
על ידי
اور תמיר

עבודת המחבר בוצעה בהנחיית
פרופ' שמואל שגיב