Priority Queues on Parallel Machines [1]

Gerth Stølting Brodal [2]

BRICS, Department of Computer Science, University of Aarhus, 8000 Århus C, Denmark [3]

Abstract

We present time and work optimal priority queues for the CREW PRAM, supporting FindMin in constant time with one processor and MakeQueue, Insert, Meld, FindMin, ExtractMin, Delete and DecreaseKey in constant time with $O(\log n)$ processors. A priority queue can be build in time $O(\log n)$ with $O(n/\log n)$ processors. A pipelined version of the priority queues adopt to a processor array of size $O(\log n)$, supporting the operations MakeQueue, Insert, Meld, FindMin, ExtractMin, Delete and DecreaseKey in constant time. By applying the k-bandwidth technique we get a data structure for the CREW PRAM which supports MultiInsert$_k$ operations in $O(\log k)$ time and MultiExtractMin$_k$ in $O(\log \log k)$ time.

Key words: Parallel priority queues, constant time operations, binomial trees, pipelined operations.

1 Introduction

The construction of priority queues is a classical topic in data structures. Some references are [1,3,5,12–16,19,29,31–33]. A historical overview of implementations has been given by Mehlhorn and Tsakalidis [22]. Recently several papers have also considered how to implement priority queues on parallel machines [6,8–11,18,24–28]. In this paper we focus on how to achieve optimal

[1] A preliminary version of the paper was presented at the *5th Scandinavian Workshop on Algorithm Theory*, 1996 [2].

[2] Supported by Danish Natural Science Research Council (Grant No. 9400044). Work done while visiting the Max-Planck-Institut für Informatik, Saarbrücken, Germany. Email: `gerth@brics.dk`.

[3] BRICS (Basic Research in Computer Science), a Centre of the Danish National Research Foundation.

speedup for the individual priority queue operations known from the sequential setting. A similar approach has been taken by Pinotti and Pucci [25] and Ranade *et al.* [26]. The operations we support are all the commonly needed priority queue operations from the sequential setting, *e.g.*, see [22].

MakeQueue Create and return a new empty priority queue.

Insert(Q, e) Insert element e into priority queue Q.

Meld(Q_1, Q_2) Meld priority queues Q_1 and Q_2. The resulting priority queue is stored in Q_1.

FindMin(Q) Return the minimum element in priority queue Q.

ExtractMin(Q) Delete and return the minimum element in priority queue Q.

Delete(Q, e) Delete element e from priority queue Q provided a pointer to e is given.

DecreaseKey(Q, e, e') Replace element e by e' in priority queue Q provided $e' \leq e$ and a pointer to e is given.

Build$(e_1, \ldots, e_n)$ Create a new priority queue containing elements $e_1, \ldots, e_n$.

We assume that elements are taken from a totally ordered universe and that the only operation allowed on elements is the comparison of two elements that can be done in constant time. Throughout this paper n denotes the maximum allowed number of elements in a priority queue, and $|Q|$ the current size of priority queue Q.

Because of the $\Omega(n \log n)$ lower bound on the number of comparisons for comparison based sorting we immediately get an $\Omega(\log n)$ lower bound on the number of comparisons Insert or ExtractMin have to do, because sorting can be done by n Insert operations followed by n ExtractMin operations, implying that any parallel implementation has to do at least $\Omega(\log n)$ work for one of these operations.

Our main result is the following.

Theorem 1 *On a CREW PRAM priority queues exist supporting* FindMin *in constant time with one processor, and* MakeQueue, Insert, Meld, ExtractMin, Delete *and* DecreaseKey *in constant time with* $O(\log n)$ *processors.* Build *is supported in* $O(\log n)$ *time with* $O(n/\log n)$ *processors.*

Table 1 lists the performance of different implementations adopting parallelism to priority queues. Several papers consider how to build heaps [14,33] optimally in parallel [10,11,18,27]. On an EREW PRAM an optimal construction time of $O(\log n)$ has been achieved by Rao and Zhang [27] and on a CRCW PRAM

[4] The operations Delete and DecreaseKey require the CREW PRAM and require amortized $O(\log \log n)$ time.

Model	EREW	EREW [4]	Array	CREW
	Pinotti, Pucci [25]	Pinotti *et al.* [23]	Ranade *et al.* [26]	*New result*
FindMin	1	$\log\log n$	1	1
Insert	$\log\log n$	$\log\log n$	1	1
ExtractMin	$\log\log n$	$\log\log n$	1	1
Meld		$\log\log n$		1
Delete		$\log\log n$		1
DecreaseKey		$\log\log n$		1
Build	$\log n$			$\log n$

Table 1
Performance of different parallel implementations of priority queues.

an optimal construction time of $O(\log\log n)$ has been achieved by Dietz and Raman [11].

An immediate consequence of the CREW PRAM priority queues we present is that on an EREW PRAM we achieve the bounds stated in Corollary 2, because the only bottleneck in the construction requiring concurrent read is the broadcasting of information of constant size, that on an $O(\log n/\log\log n)$ processor EREW PRAM requires $O(\log\log n)$ time. Our time bounds are identical to those obtained by Pinotti *et al.* [23]. See Table 1.

Corollary 2 *On an EREW PRAM priority queues exist supporting* FindMin *in constant time with one processor, and* MakeQueue, Insert, Meld, ExtractMin, Delete *and* DecreaseKey *in* $O(\log\log n)$ *time with* $O(\log n/\log\log n)$ *processors. With* $O(n/\log n)$ *processors* Build *can be performed in* $O(\log n)$ *time.*

That a systolic processor array with $\Theta(n)$ processors can implement a priority queue supporting the operations Insert and ExtractMin in constant time is parallel computing folklore, see Exercise 1.119 in [21]. Ranade *et al.* [26] showed how to achieve the same bounds on a processor array with only $O(\log n)$ processors. In Section 5 we describe how our priority queues can be modified to allow operations to be performed via pipelining. As a result we get an implementation of priority queues on a processor array with $O(\log n)$ processors, supporting the operations MakeQueue, Insert, Meld, FindMin, ExtractMin, Delete and DecreaseKey in constant time. This extends the result of Ranade *et al.* [26].

A different approach to adopt parallelism to priority queues is by supporting the following two operations, where k is a fixed constant.

	Pinotti, Pucci [24]	Chen, Hu [6]	*New result*
Model	CREW	EREW	CREW
MELD	$\log\frac{n}{k} + \log\log k$	$\log\log\frac{n}{k} + \log k$	$\log\log k$
MULTIINSERT$_k$	$\log n$	$\log\log\frac{n}{k} + \log k$	$\log k$
MULTIEXTRACTMIN$_k$	$\log\frac{n}{k} + \log\log k$	$\log\log\frac{n}{k} + \log k$	$\log\log k$

Table 2
Performance of different parallel implementations of priority queues supporting multi-operations.

MULTIINSERT$_k(Q, e_1, \ldots, e_k)$ Insert elements $x_1, \ldots, x_k$ into priority queue Q.
MULTIEXTRACTMIN$_k(Q)$ Delete the k least elements from Q. The k elements are returned as a sorted list.

By applying the k-bandwidth technique, our data structure can be adapted to support the above multi-operations. A comparison with previous work is shown in Table 2. Our time bound are the first to be independent of n.

We throughout this paper assume that the arguments to the procedures initially only are known to processor zero, and that output is generated at processor zero too.

In Section 2 we present optimal priority queues for a CREW PRAM supporting the basic priority queue operations FINDMIN, MAKEQUEUE, INSERT, MELD and EXTRACTMIN. In Section 3 we extend the priority queues to support DELETE and DECREASEKEY. In Section 4 we consider how to build a priority queue. In Section 5 we present a pipelined version of our priority queues, and in Section 6 we describe how the k-bandwidth idea can be applied to our data structure. Finally some concluding remarks are given in Section 7.

2 Meldable priority queues

In this section we describe how to implement the priority queue operations MAKEQUEUE, FINDMIN, INSERT, MELD and EXTRACTMIN in constant time on a CREW PRAM with $\lfloor \log_2(n+1) \rfloor$ processors. In Section 3 we describe how to extend the repertoire of priority queue operations to include DELETE and DECREASEKEY too.

The priority queues we present in this section are based on *heap ordered binomial trees*. In the following we assume a one to one mapping between nodes of trees and priority queue elements, and for two nodes x and y, we let $x \leq y$ refer to the comparison between the two elements at the two nodes.

A tree where each node stores an element from a totally ordered universe is said to satisfy *heap order* if for all nodes v, the element stored at v is greater or equal to the element stored at the parent of v. An immediate consequence is that a heap ordered tree always has the minimum element at the root.

Binomial trees are heap ordered trees defined as follows. A binomial tree of *rank* zero is a single node. A binomial tree of rank $r \geq 1$ is obtained from two binomial trees of rank $r-1$ by making the root with the largest element the leftmost child of the root with smallest element, with draws broken arbitrarily. We let the rank of a node v denote the rank of the binomial tree rooted at v. It follows by induction that a binomial tree of rank r has exactly 2^r nodes and that a node of rank r has exactly one child of each of the ranks $0, \ldots, r-1$, the children appearing in decreasing rank order from left to right. Essential to our data structure is the fact that the link operation is reversible, *i.e.*, that we can *unlink* a binomial tree of rank $r \geq 1$ to two binomial trees of rank $r-1$. Binomial trees of rank zero to four are shown in Figure 1, and the linking of two binomial trees of rank three to a binomial tree of rank four is shown in Figure 2.

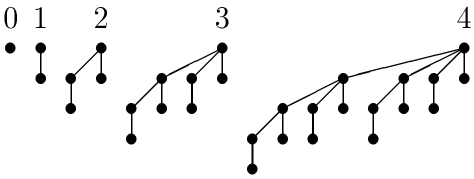

Fig. 1. The structure of binomial trees of rank zero to four.

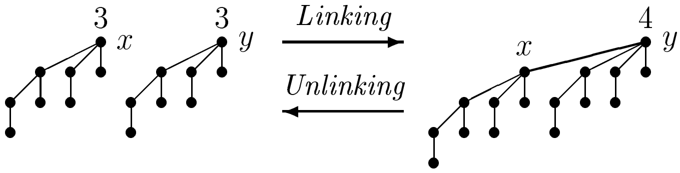

Fig. 2. Linking and unlinking binomial trees. It is assumed that the element stored at x is greater than or equal to the element stored at y.

The basic idea of the prominent binomial queues of Vuillemin [32] is to represent a priority queue by a collection of distinct ranked binomial trees. For implementation details on binomial queues refer to [32].

We also represent a priority queue Q by a forest of binomial trees, but we do not require all trees to have distinct ranks. In the following we let $r(Q)$ denote the largest rank of a tree in the representation of priority queue Q, we let $n_i(Q)$ denote the number of trees of rank i, and we let $n_{\max}(Q)$ denote the value $\max_{0 \leq i \leq r(Q)} n_i(Q)$. Throughout the rest of this section a tree denotes a binomial tree.

We require that a forest of trees representing a priority queue Q satisfies the two following constraints:

$\mathbf{A}_1$: $n_i(Q) \in \{1, 2, 3\}$ for all $i = 0, \dots, r(Q)$, and
$\mathbf{A}_2$: the minimum of the elements at roots of rank i is less than or equal to all the elements at roots of rank greater than i, for all $i = 0, \dots, r(Q)$.

The first constraint bounds the number of trees of each rank. We require that for each rank $i \leq r(Q)$ there is at least one tree of rank i present in the forest, and that at most three trees have equal rank. A bound on $r(Q)$ is given by Lemma 3 below. The second constraint forces an ordering upon the elements at the roots. Especially, we require that the minimum element is stored at a tree of rank zero. Figure 3 gives an example of a forest satisfying the two constraints. The nodes are roots. The labels inside the nodes are the elements, in the following integers, and the numbers below the nodes are the ranks.

Lemma 3 *$r(Q) \leq \lfloor \log_2(|Q| + 1) \rfloor - 1$.*

PROOF. By A_1,

$$|Q| \geq \sum_{i=0}^{r(Q)} 2^i = 2^{r(Q)+1} - 1,$$

and the lemma follows. $\square$

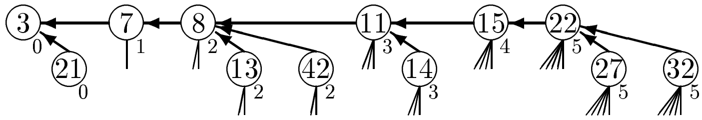

Fig. 3. A forest satisfying constraints A_1 and A_2. The arrows denote the ordering forced by A_2 upon the elements at the roots.

A priority queue is stored as follows. Each node v in a priority queue Q is represented by a record consisting of the following fields.

e : the element associated to v, and
L : a linked list of the children of v in decreasing rank order.

Notice that we do not store the rank of the nodes.

For a priority queue Q of size at most n we maintain an array $Q.L$ of size $\lfloor \log_2(n + 1) \rfloor$, such that $Q.L[i]$ is a pointer to a linked list of all roots of rank i. By A_1, $|Q.L[i]| \leq 3$ for all i. Notice that storing the children of a node in a linked list in decreasing rank order allows two nodes of equal rank to be linked in constant time by one processor.

Essential to our algorithms are the two procedures ParLink and ParUnlink. Pseudo code for the procedures is given in Figure 4. The procedure ParLink

Proc PARLINK(Q)
  for $p := 0$ **to** $\lfloor \log_2(n+1) \rfloor - 2$ **pardo**
    if $n_p(Q) \geq 3$ **then**
      Link two trees from $Q.L[p] \setminus \min(Q.L[p])$ and
      add the resulting tree to $Q.L[p+1]$
    fi
  od
end

Proc PARUNLINK(Q)
  for $p := 1$ **to** $\lfloor \log_2(n+1) \rfloor - 1$ **pardo**
    if $n_p(Q) \geq 1$ **then**
      Unlink $\min(Q.L[p])$ and add the resulting two trees to $Q.L[p-1]$
    fi
  od
end

Fig. 4. Parallel linking and unlinking binomial trees.

for each rank i in parallel links two trees of rank i to one tree of rank $i+1$, provided there are at least three trees of rank i. We require that the trees of rank i which are linked together are different from $\min(Q.L[i])$, *i.e.*, the rank i root with the smallest element remains a rank i root. The procedure PARUNLINK in parallel for each rank i unlinks $\min(Q.L[i])$. Figure 5 shows an application of the procedures PARUNLINK and PARLINK. For trees that are unlinked (linked) by PARUNLINK (PARLINK) the leftmost child of the root is shown too.

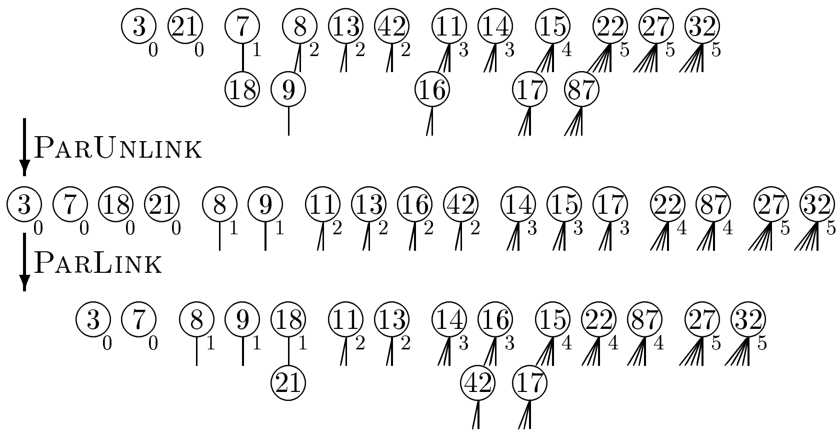

Fig. 5. Parallel linking and unlinking. Notice that parallel unlinking followed by parallel linking is not the identity function.

In procedures PARLINK(Q) and PARUNLINK(Q), processor p only accesses $Q.L$ at entries $p-1$, p and $p+1$, implying that the procedures can be implemented on an EREW PRAM with $\lfloor \log_2(n+1) \rfloor$ processors in constant time if processor p initially knows the address of $Q.L[p]$. Actually, it is sufficient for all processors to know the start address of the array $Q.L$.

The following lemmas capture the behavior of the procedures ParLink and ParUnlink with respect to the constraints A_1 and A_2.

Lemma 4 *Let n_i and $n_{\max}$ denote the values of $n_i(Q)$ and $n_{\max}(Q)$, and let n_i' and $n_{\max}'$ denote the corresponding values after applying* ParLink(Q). *Then $n_{\max}' \leq \max\{3, n_{\max} - 1\}$, $n_0' \leq \max\{2, n_0 - 2\}$, and $n_i' \leq \max\{3, n_i - 1\}$ for $i \geq 1$.*

PROOF. If $n_0 \leq 2$, then $n_0' = n_0 \leq 2$. Otherwise we link two trees of rank zero and have $n_0' = n_0 - 2$. If $n_i \leq 2$ for $i \geq 1$, then we do not link any trees of rank i. At most one new tree of rank i can be created due to the linking of two trees of rank $i - 1$, and we have $n_i' \leq 2 + 1 = 3$. Otherwise we link two trees of rank i and have $n_i' \leq n_i - 2 + 1 = n_i - 1$, and the lemma follows. $\square$

Lemma 5 *If A_2 is satisfied for priority queue Q, then A_2 is also satisfied after applying* ParLink(Q).

PROOF. Assume A_2 is satisfied for rank $i \leq r(Q)$ before applying ParLink(Q). Let x denote $\min(Q.L[i])$. Because x is not linked, and all new roots with rank $> i$ is the result of linking two roots with rank $\geq i$ it follows that after applying ParLink(Q), x is still less than or equal to all the elements at roots of rank $> i$. It follows that the resulting $\min(Q.L[i]) \leq x$ satisfies A_2, and the lemma follows. $\square$

Lemmas 4 and 5 state that if the maximum number of trees of equal rank is greater than three, then an application of ParLink(Q) decreases this value by at least one without violating A_2.

Lemma 6 *Let n_i denote the value $n_i(Q)$, and n_i' the corresponding value after applying* ParUnlink(Q). *If $n_i \geq 1$ for $i = 1, \ldots, r(Q)$, then $n_0' \leq n_0 + 2$, and $n_i' \leq n_i + 1$ for $i \geq 1$.*

PROOF. At most two new trees of rank zero are created because of unlinking a tree of rank one. For rank $i \geq 1$, two new trees of rank i are only created if a tree of rank $i + 1$ is unlinked, in which case we also unlink a rank i tree, and $n_i' \leq n_i + 2 - 1 = n_i + 1$. $\square$

Lemma 7 *If for priority queue Q, A_2 is satisfied for all $i \geq 1$, then after applying* ParUnlink(Q), *A_2 is satisfied (for all $i \geq 0$).*

PROOF. Notice that for $i = 0, \ldots, r(Q)$, after applying ParUnlink(Q), $\min(Q.L[i])$ is less than or equal to the previous $\min(Q.L[i+1])$, because ParUnlink(Q) unlinks $\min(Q.L[i+1])$, implying that the new element $\min(Q.L[i])$ is less than or equal to all elements at the resulting roots of rank $> i$. □

Lemmas 4 and 6 guarantee that if A_1 is satisfied for priority queue Q, then A_1 is also satisfied after applying ParUnlink(Q) followed by ParLink(Q). Lemma 7 guarantees that if we make A_2 violated for priority queue Q because we remove $\min(Q.L[0])$, *i.e.*, we extract the minimum element from Q, we can reestablish A_2 by applying ParUnlink(Q).

We can now implement the priority queue operations as follows. We assume that the priority queues before performing the operations satisfy A_1 and A_2.

MakeQueue The array $Q.L$ is allocated and in parallel all $Q.L[i]$ are assigned the empty set.

FindMin(Q) Constraint A_2 guarantees that the minimum element in priority queue Q is $\min(Q.L[0])$. Processor zero returns $\min(Q.L[0])$.

Insert(Q, e) To insert element e into priority queue Q, a new tree of rank zero containing e is created and added to $Q.L[0]$ by processor zero. Constraint A_2 remains satisfied, and constraint A_1 can only become violated for rank zero if $|Q.L[0]| = 4$. By applying ParLink(Q) once it follows from Lemma 4 that A_1 is reestablished.

Meld(Q_1, Q_2) To merge priority queue Q_2 into priority queue Q_1 we merge the two forest by letting processor p set $Q_1.L[p]$ to $Q_1.L[p] \cup Q_2.L[p]$. Because $\min(Q_1.L[i])$ and $\min(Q_2.L[i])$ by A_2 were monotonically nondecreasing sequences in i, it follows that $\min(Q_1.L[i] \cup Q_2.L[i])$ is a monotonically nondecreasing sequence in i, and A_2 is therefore satisfied after having merged the two forests. The resulting forest satisfies $n_{\max}(Q_1) \leq 6$. By Lemma 4 we can reestablish A_1 by applying ParLink(Q_1) three times.

ExtractMin(Q) First processor zero finds and removes the minimum element from Q, which by A_2 is $\min(Q.L[0])$. By Lemma 7 it is sufficient to apply ParUnlink once to guarantee that A_2 is reestablished. After deleting a tree of rank zero and applying ParUnlink(Q), it follows by Lemma 6 that $n_{\max} \leq 4$. By Lemma 4 it is sufficient to apply ParLink once to reestablish A_1.

Pseudo code for the priority queue operations based on the previous discussion is shown in Figure 6. The procedures *new-queue* and *new-node*(e) allocate a new array $Q.L$ and a new node record in memory. Notice that the only part of the code requiring concurrent read is to "broadcast" the names of Q, Q_1 and Q_2 to all the processors, *i.e.*, the address of $Q.L$, $Q_1.L$, and $Q_2.L$. Otherwise the code only requires an EREW PRAM. From the fact that ParLink and ParUnlink can be performed in constant time with $\lfloor \log_2(n+1) \rfloor$ processors

```
Proc MakeQueue
   Q := new-queue
   for p := 0 to ⌊log_2(n + 1)⌋ − 1 pardo
      Q.L[p] := ∅
   od
   return Q
end

Proc FindMin(Q)
   return min(Q.L[0])
end

Proc Insert(Q, e)
   Q.L[0] := Q.L[0] ∪ {new-node(e)}
   ParLink(Q)
end

Proc Meld(Q_1, Q_2)
   for p := 0 to ⌊log_2(n + 1)⌋ − 1 pardo
      Q_1.L[p] := Q_1.L[p] ∪ Q_2.L[p]
   od
   do 3 times ParLink(Q_1)
end

Proc ExtractMin(Q)
   e := min(Q.L[0])
   Q.L[0] := Q.L[0] \ min(Q.L[0])
   ParUnlink(Q)
   ParLink(Q)
   return e
end
```

Fig. 6. CREW PRAM priority queue operations.

we have:

Theorem 8 *On a CREW PRAM priority queues exist supporting* FindMin *in constant time with one processor, and* MakeQueue, Insert, Meld *and* ExtractMin *in constant time with* $\lfloor \log_2(n+1) \rfloor$ *processors. If the processors know the addresses of the* $Q.L$ *arrays of the involved priority queues, then an EREW PRAM is sufficient.*

3 Priority queues with deletions

In this section we extend the repertoire of supported priority queue operations to include Delete and DecreaseKey. Notice that it is sufficient to give an implementation that supports Delete(Q, e), because DecreaseKey(Q, e, e') can be implemented as Delete(Q, e) followed by Insert(Q, e').

The priority queues in this section are not based on binomial trees, but on heap ordered trees defined as follows. To each node we assign a nonnegative integer rank, and the rank of a tree is the rank of the root of the tree. A tree of rank zero is a single node. A tree of rank r is a tree where the root has exactly five children of each of the ranks $0, 1, \ldots, r-1$, the children appearing in decreasing rank order from left to right. A tree of rank r can be created by linking six trees of rank $r-1$, by making five of the roots the leftmost children of the root with the smallest element. Notice that this is a straightforward generalization of binomial queues, and that a tree of rank $r \geq 1$ can be unlinked into six trees of rank $r - 1$.

Essential to the operations DELETE and DECREASEKEY is the additional concept of *holes* in trees. Each hole has a rank. A hole of rank r in a tree is a location in the tree where a child of rank r is missing. Figure 7 shows a tree of rank two with two holes of rank zero and two holes of rank one. Notice that if the tree of rank two is unlinked into trees of rank one, then the result is not six trees of rank one, but only four trees of rank one plus two holes of rank one which disappear.

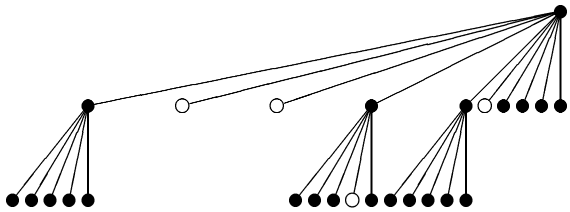

Fig. 7. A tree of rank two with two holes of rank zero and two holes of rank one.

We represent a priority queue Q by a forest of the above defined trees with holes. Let $r(Q), n_i(Q)$ and $n_{\max}(Q)$ be defined as in Section 2, and let $h_i(Q)$ denote the number of holes of rank i in the forest. We require the following constraints to be satisfied for a forest representing a priority queue Q.

B$_1$: $n_i(Q) \in \{1, 2, \ldots, 7\}$, for $i = 0, \ldots, r(Q)$,
B$_2$: the minimum of the elements at roots of rank i is less than or equal to all the elements at roots of rank greater than i, for all $i = 0, \ldots, r(Q)$, and
B$_3$: $h_i(Q) \leq 2$, for $i = 0, \ldots, r(Q) - 1$.

Constraint B_2 is identical to A_2, and B_1 bounds the number of trees of equal rank similarly to A_1. The new upper bound on the number of trees of equal rank is a consequence of that we in this section link six trees of equal rank instead of two trees. Constraint B_3 bounds the number of holes in the forest representing Q, *i.e.*, bounds the unbalancedness of the trees in the forest. An upper bound on $r(Q)$ is given by the following lemma.

Lemma 9 *If* $h_i(Q) \leq 4$ *for* $i = 0, \ldots, r(Q) - 1$, *then* $r(Q) \leq \lfloor \log_6 |Q| \rfloor + 1$.

PROOF. By straightforward induction, a tree of rank r without any holes has size 6^r. Because each hole of rank i removes exactly 6^i nodes from a tree, it follows that the number of nodes in a tree of rank r is at least

$$6^r - \sum_{i=0}^{r-1} 4 \cdot 6^i = 6^r - 4\frac{6^r - 1}{6 - 1} = \frac{1}{5}6^r + \frac{4}{5}.$$

Because $|Q| \geq \frac{1}{5}6^{r(Q)} + \frac{4}{5}$, we have $r(Q) \leq \log_6 |Q| + \log_6 5$. The lemma follows from the fact that $r(Q)$ is an integer. □

Temporary while performing MELD we later on allow the number of holes of equal rank to be at most four. The requirement that a node of rank r has five children of each of the ranks $0, \ldots, r-1$ implies that at least one child of each rank is not replaced by a hole at any point of time. This observation is crucial to the method of filling up holes that is described in this section.

We store a priority queue Q of size at most n as follows. Each node v of a tree is represented by a record consisting of:

e : the element associated to v,
r : the rank of v,
f : a pointer to the parent of v, and
L : an array of size $\lfloor \log_6 n \rfloor + 1$ of pointers to linked lists of children of equal rank.

Notice that we in this section need to store the parent pointers, the ranks of the nodes, and that $v.L$ are arrays instead of linked lists.

For each priority queue Q two arrays $Q.L$ and $Q.H$ are maintained of size $\lfloor \log_6 n \rfloor + 2$. The array $Q.L$ contains pointers to linked lists of trees of equal rank and $Q.H$ contains pointers to linked lists of "holes" of equal rank. More precisely $Q.H[i]$ is a linked list of nodes such that for each missing child of rank i of node v, v appears once in $Q.H[i]$. By B_1 and B_3, $|Q.L[i]| \leq 7$ and $|Q.H[i]| \leq 2$ for all i.

Similarly to the algorithms in Section 2 we have two procedures PARLINK and PARUNLINK. The procedures have to be modified such that linking and unlinking involves six trees instead of two trees. The procedure PARLINK(Q) for each rank p links six trees of rank p different from $\min(Q.L[p])$, if $n_p(Q) \geq 7$. The procedure PARUNLINK(Q, r) for each rank $p \geq r+1$ unlinks $\min(Q.L[p])$, if $n_p(Q) \geq 1$. The additional parameter r is required for the implementation of DELETE (for the case when the node to be deleted is $\min(Q.L[r])$). Because the holes below $\min(Q.L[p])$ of rank $p-1$ disappear when unlinking $\min(Q.L[p])$, we remove all appearances of $\min(Q.L[p])$ from $Q.H[p-1]$ before unlinking. The modified procedures are shown in Figure 8.

The behavior of the modified algorithms is captured by the following lemmas.

Lemma 10 *Let n_i, $n_{\max}$ and h_i denote the values of $n_i(Q)$, $n_{\max}(Q)$, and $h_i(Q)$, and let n'_i, $n'_{\max}$ and h'_i denote the corresponding values after applying* PARLINK(Q). *Then $n'_{\max} \leq \max\{7, n_{\max} - 5\}$, $n'_0 \leq \max\{6, n_0 - 6\}$, $n'_i \leq \max\{7, n_i - 5\}$ for $i \geq 1$, and $h'_i = h_i$ for all i.*

PROOF. The proof for n_i goes as for Lemma 4, except that 2 is replaced by 6. Because no holes are introduced or eliminated we have $h'_i = h_i$ for all i. □

```
Proc ParLink(Q)
   for p := 0 to ⌊log_6 n⌋ pardo
      if n_p(Q) ≥ 7 then
         Link six trees from Q.L[p] \ min(Q.L[p]) and
         add the resulting tree to Q.L[p + 1]
      fi
   od
end

Proc ParUnlink(Q, r)
   for p := 1 to ⌊log_6 n⌋ + 1 pardo
      if n_p(Q) ≥ 1 and p > r then
         Q.H[p − 1] := Q.H[p − 1] \ min(Q.L[p])
         Unlink min(Q.L[p]) and add the resulting trees to Q.L[p − 1]
      fi
   od
end
```

Fig. 8. Parallel linking and unlinking trees.

Lemma 11 *If B_2 is satisfied for priority queue Q, then B_2 is also satisfied after applying* ParLink(Q).

PROOF. Identical to the proof of Lemma 5. □

Lemma 12 *Let n_i and h_i denote the values $n_i(Q)$ and $h_i(Q)$, and n'_i and h'_i the corresponding values after applying* ParUnlink(Q, r). *If $n_i \geq 1$ for $i = r+1, \ldots, r(Q)$, then $n'_i = n_i$ for $i \leq r-1$, $n'_r \leq n_r + 6$, and $n'_i \leq n_i + 5$ for $i \geq r+1$. For all i, $h'_i \leq h_i$.*

PROOF. No trees of rank $\leq r-1$ are created or unlinked. At most 6 new trees of rank r result from unlinking $\min(Q.L[r+1])$. For rank $\geq r+1$, the argument goes as in the proof of Lemma 6 with 2 replaced by 6.

Because no new holes are introduced by ParUnlink and unlinking $\min(Q.L[i])$ can eliminate some holes with rank $i-1$, we have $h'_i \leq h_i$ for all i. □

Lemma 13 *If for priority queue Q, B_2 is satisfied for all $i \neq r$, then after applying* ParUnlink(Q, r), *B_2 is satisfied (for all i).*

PROOF. Because only trees of rank $\geq r+1$ are unlinked into trees of ranks $\geq r$, B$_2$ remains satisfied for all $i \leq r-1$. For rank $i \geq r$ the argument is identical to the argument in the proof of Lemma 7. □

We now describe a procedure FIXHOLES that in parallel for each rank reduces the number of holes similar to how the procedure PARLINK reduces the number of trees. When applying FIXHOLES(Q) we assume $h_i(Q) \leq 4$ for all i. The procedure is constructed such that processor p takes care of holes of rank p. The work done by processor p is the following. If $|Q.H[p]| < 2$ processor p does nothing. Otherwise processor p considers two holes in $Q.H[p]$. Recall that all holes have at least one real tree node of rank p as a sibling. If the two holes have different parents, one of the holes is swapped with a sibling node of the other hole. This makes both holes have the same parent f. By choosing the swap node as the node with the largest element among the two sibling nodes of the holes we are guaranteed to satisfy heap order after the swap.

There are now two cases to consider. The first case is when the two holes have a sibling node b of rank $p+1$. Notice that b has at least three children of rank p because we assumed at most four holes of rank p and two of the holes are assumed to be siblings of b. We can now cut off b, and cut off all children of b of rank p by unlinking b. By assigning b the rank p we only create one new hole of rank $p+1$. We can now eliminate the two original holes of rank p by replacing them with two previous children of b. At most four trees remain to be added to $Q.L[p]$, depending on how many holes of rank p were below b. The second case is when f has rank $p+1$. Assume first that $f \neq \min(Q.L[p+1])$. In this case the subtree rooted at f can be cut off without violating B_2. If f is not a root this creates a new hole of rank $p+1$. We can now cut off all children of f that have rank p and assign f the rank p. This eliminates the two holes. At most four trees now need to be added to $Q.L[p]$. Finally there is the case when $f = \min(Q.L[p+1])$. By applying PARUNLINK$(Q, 0)$ and PARLINK(Q) once the two holes disappear. To compensate for the created new trees we finally perform PARLINK once more. Pseudo code for FIXHOLES is shown in Figure 9 and the two relinking cases are shown in Figure 10.

Lemma 14 *Let h_i denote $h_i(Q)$ and h'_i the corresponding values after applying* FIXHOLES(Q). *If $h_i(Q) \leq 4$ for all i, then $h'_i \leq \max\{2, h_i - 1\}$ for all i.*

PROOF. If $h_i < 2$, then at most one new hole of rank i can be created because two holes of rank $i-1$ were eliminated. Otherwise $h_i \geq 2$ and we eliminate two holes of rank i, and have $h'_i \leq h_i - 2 + 1 = h_i - 1$. □

The priority queue operations can now be implemented as follows.

MAKEQUEUE Allocate new arrays $Q.L$ and $Q.H$ and assign the empty set to $Q.L[i]$ and $Q.H[i]$ for all $i = 0, \ldots, \lfloor \log_6 n \rfloor + 1$.

INSERT(Q, e) Create a tree of rank zero containing e and add this tree to

Proc FixHoles(Q)
  for $p := 0$ **to** $\lfloor \log_6 n \rfloor$ **pardo**
    if $|Q.H[p]| \geq 2$ **then**
      Let $f_p, f'_p \in Q.H[p]$ be the parents of two holes of rank p
      if $f_p \neq f'_p$ **then**
        Let $b_p \in f_p.L[p]$ and $b'_p \in f'_p.L[p]$ be sibling nodes of the holes
        if $b_p.e \leq b'_p.e$ **then**
          Move b'_p from $f'_p.L[p]$ to $f_p.L[p]$
          Replace one occurrence of f_p by f'_p in $Q.H[p]$
          $f_p := f'_p$
        else
          Move b_p from $f_p.L[p]$ to $f'_p.L[p]$
          Replace one occurrence of f'_p by f_p in $Q.H[p]$
        fi
      fi
      if $f_p.L[p+1] \neq \emptyset$ **then**
        Let $b_p \in f_p.L[p+1]$ be a sibling node of rank $p+1$ of the holes
        Move two children of b_p from $b_p.L[p]$ to $f_p.L[p]$
        Move $b_p.L[p]$ and b_p to $Q.L[p]$
        Remove all occurrences of b_p and twice f_p from $Q.H[p]$
        Insert f_p into $Q.H[p+1]$
      else
        if $f_p \neq \min(Q.L[p+1])$ **then**
          Insert $f_p.f$ into $Q.H[p+1]$ if f_p is not a root
        Move $f_p.L[p]$ and f_p to $Q.L[p]$
        Remove all occurrences of f_p from $Q.H[p]$
        fi
      fi
    fi
  od
  ParUnlink($Q, 0$)
  do 2 **times** ParLink(Q)
end

Fig. 9. Parallel elimination of holes.

$Q.L[0]$. Only B_1 can become violated for rank zero, if $n_0(Q) = 7$. By Lemma 10 it is sufficient to perform ParLink(Q) once to reestablish B_1. Notice that Insert does not affect the number of holes in Q.

Meld(Q_1, Q_2) Merge $Q_2.L$ into $Q_1.L$, and $Q_2.H$ into $Q_1.H$. We now have $|Q_1.L| \leq 14$ and $|Q_1.H[i]| \leq 4$ for all i. That B_2 is satisfied follows from that Q_1 and Q_2 satisfied B_2 as in Section 2. By Lemma 10 we can reestablish B_1 by applying ParLink(Q_1) twice. By Lemma 14 we can reestablish B_3 by applying FixHoles(Q_2) twice.

FindMin(Q) Return $\min(Q.L[0])$.

ExtractMin(Q) First perform FindMin and then perform Delete on the

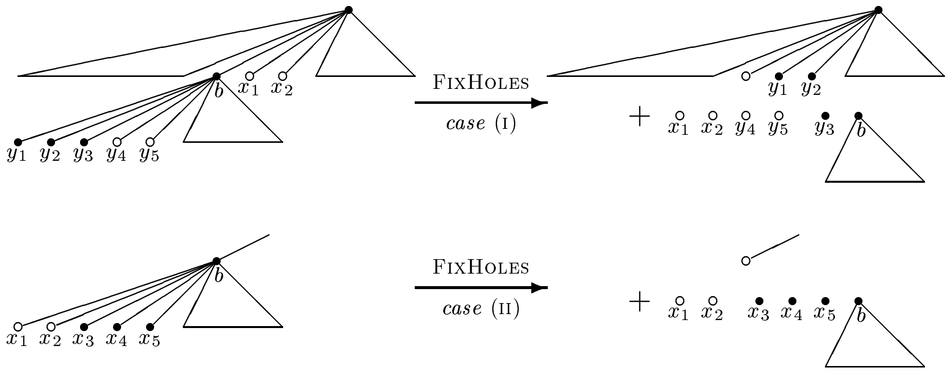

Fig. 10. Transformations to replace two holes of rank p (x_1 and x_2) by one hole of rank $p+1$.

found minimum.

DELETE(Q, e) Let v be the node containing e. Remove the subtree with root v. If this creates a hole then add the hole to $Q.H$ by adding $v.f$ to $Q.H[v.r]$. Merge $v.L$ into $Q.L$ and eliminate all holes below v by removing all appearances of v from $Q.H$. Notice that at most one new hole of rank $v.r$ is created, and that for $i = 0, \ldots, v.r - 1$ at most five new trees are added to $Q.L[i]$. Only for $i = v.r$, $\min(Q.L[i])$ can change and this only happens if v was $\min(Q.L[i])$, in which case B_2 can become violated for rank $v.r$. If $v = \min(Q.L[i])$, we can by Lemma 13 reestablish B_2 performing PARUNLINK($Q, v.r$). Because v is removed from $Q.L[v.r]$ in this case we have from Lemma 12 that for all $i \geq v.r$, $n_i(Q)$ at most increases by five. We now have that for all i, $n_i(Q)$ can at most increase by five. By Lemma 10 we can reestablish B_1 by performing PARLINK once.

Because at most one new hole of rank $v.r$ has been created, we can by Lemma 14 reestablish B_3 by performing FIXHOLES once.

DECREASEKEY(Q, e, e') Perform DELETE(Q, e) followed by INSERT(Q, e').

A pseudo code implementation for a CREW PRAM based on the previous discussion is shown in Figure 11. Notice that the only part of the code that requires concurrent read is the "broadcasting" of the parameters of the procedures. For DELETE all processors in addition have to know v (the address of $v.L$ for doing the parallel merge of $v.L$ and $Q.L$, and the rank $v.r$ and if $v = \min(Q.L[v.r])$ to decide if a parallel unlinking is necessary). In the rest of the code processor p only accesses entries $p-1$, p, and $p+1$ of arrays, and these computations can be done in constant time with $\lfloor \log_6 n \rfloor + 2$ processors on an EREW PRAM.

Theorem 15 *On a CREW PRAM priority queues exist supporting* FINDMIN *in constant time with one processor, and* MAKEQUEUE, INSERT, MELD, EXTRACTMIN, DELETE *and* DECREASEKEY *in constant time with* $\lfloor \log_6 n \rfloor + 2$ *processors.*

```
Proc MakeQueue
  Q := new-queue
  for p := 0 to ⌊log_6 n⌋ + 1 pardo
    Q.L[p] := ∅
    Q.H[p] := ∅
  od
  return Q
end

Proc FindMin(Q)
  return min(Q.L[0])
end

Proc Insert(Q, e)
  Q.L[0] := Q.L[0] ∪ {new-node(e)}
  ParLink(Q)
end

Proc Meld(Q_1, Q_2)
  for p := 0 to ⌊log_6 n⌋ + 1 pardo
    Q_1.L[p] := Q_1.L[p] ∪ Q_2.L[p]
    Q_1.H[p] := Q_1.H[p] ∪ Q_2.H[p]
  od
  do 2 times ParLink(Q_1)
  do 2 times FixHoles(Q_1)
end

Proc DecreaseKey(Q, e, e')
  Delete(Q, e)
  Insert(Q, e')
end

Proc ExtractMin(Q)
  e := FindMin(Q)
  Delete(Q, e)
  return e
end

Proc Delete(Q, e)
  v := the node containing e
  if v.f ≠ nil then
    Q.H[v.r] := Q.H[v.r] ∪ {v.f}
    v.f.L[v.r] := v.f.L[v.r] \ {v}
  fi
  for p := 0 to ⌊log_6 n⌋ + 1 pardo
    for u ∈ v.L[p] do u.f := nil od
    Q.L[p] := Q.L[p] ∪ v.L[p]
    Q.H[p] := Q.H[p] \ {v}
  od
  if v.f = nil then
    if v = min(Q.L[v.r]) then
      ParUnlink(Q, v.r)
    fi
    Q.L[v.r] := Q.L[v.r] \ {v}
  fi
  ParLink(Q)
  FixHoles(Q)
end
```

Fig. 11. CREW PRAM priority queue operations.

4 Building priority queues

In this section we describe how to perform Build$(x_1, \ldots, x_n)$ for the priority queues in Section 3. The same approach applies to the priority queues described in Section 2. Because our priority queues can report a minimum element in constant time and that there is lower bound of $\Omega(\log n)$ for finding the minimum of a set of elements on a CREW PRAM [17] we have an $\Omega(\log n)$ lower bound on the construction time on a CREW PRAM. We now give a matching upper bound on the construction time on an EREW PRAM.

First a collection of trees is constructed satisfying B_1 and B_3 but not B_2. We assume the n elements are given as n rank zero trees. We partition the elements into $\lfloor (n-1)/6 \rfloor$ blocks of size six. In parallel we now construct a rank one tree from each block. The remaining 1–6 elements are stored in $Q.L[0]$. The same block partitioning and linking is now done for the rank one trees. The remaining rank one trees are stored in $Q.L[1]$. This process continues until no tree remains. Because the resulting forest has no holes, we have $r(Q) \leq \lfloor \log_6 n \rfloor$ and there are at most $\lfloor \log_6 n \rfloor + 1$ iterations because. The resulting forest satisfies B_1 and B_3. By standard techniques it follows that the above construction can be done in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM.

To establish B_2 we for $i = 1, \ldots, \lfloor \log_6 n \rfloor$ perform $\textsc{ParUnlink}(Q, \lfloor \log_6 n \rfloor - i)$ followed by $\textsc{ParLink}(Q)$. By induction it follows as in the proof of Lemma 13 that after the ith iteration B_2 is satisfied for all ranks $\geq \lfloor \log_6 n \rfloor - i$. This finishes the construction of the priority queue. The last step of the construction requires $O(\log n)$ time with $\lfloor \log_6 n \rfloor + 1$ processors. We conclude that:

Theorem 16 *On an EREW PRAM a priority queue containing n elements can be constructed optimally with $O(n/\log n)$ processors in $O(\log n)$ time.*

Because $\textsc{Meld}(Q, \textsc{Build}(x_1, \ldots, x_k))$ implements the priority queue operation $\textsc{MultiInsert}(Q, x_1, \ldots, x_k)$ we have the following corollary.

Corollary 17 *On a CREW PRAM the* MultiInsert *operation can be performed in $O(\log k)$ time with $O((\log n + k)/\log k)$ processors.*

Notice that k does not have to be fixed as in [6] and [24] (in [6] and [24], k needs to be a fixed constant due to the supported $\textsc{MultiExtractMin}_k$ operation).

A different approach to build a priority queue containing n elements would be to merge the n single element priority queues in a treewise fashion by performing $n-1$ Meld operations. The thereby achieved bounds would match those of Theorem 16, but the described approach illustrates that it is easy to convert a forest not satisfying B_2 into a forest that satisfies B_2 by using the procedures ParLink and ParUnlink.

5 Pipelined priority queue operations

The priority queues in Sections 2 and 3 require the CREW PRAM to achieve constant time per operation. In this section we address how to perform priority queue operations in a pipelined fashion. As a consequence we get an implementation of priority queues on a processor array of size $O(\log n)$ supporting priority queue operations in constant time. As in [26] we assume that on a processor array all requests are entered at processor zero and that output is

```
Proc MakeQueue
   Q := new-queue
   for p := 0 to ⌊log₂(n + 1)⌋ − 1 do
      Q.L[p] := ∅
   od
   return Q
end

Proc FindMin(Q)
   return min(Q.L[0])
end

Proc Insert(Q, e)
   Q.L[0] := Q.L[0] ∪ {new-node(e)}
   for i := 0 to ⌊log₂(n + 1)⌋ − 2 do
      Link(Q, i)
   od
end

Proc Meld(Q₁, Q₂)
   for i := 0 to ⌊log₂(n + 1)⌋ − 1 do
      Q₁.L[i] := Q₁.L[i] ∪ Q₂.L[i]
      do 2 times Link(Q₁, i)
   od
end

Proc DecreaseKey(Q, e, e′)
   Delete(Q, e)
   Insert(Q, e′)
end

Proc ExtractMin(Q)
   e := FindMin(Q)
   Delete(Q, e)
   return e
end

Proc Delete(Q, e)
   v := the node containing e
   r := 0
   if v.rightmost-child ≠ NIL do
      v := v.rightmost-child
      while v.f ≠ NIL do
         Move v to Q.L[r] and v := v.left
         Link(Q, r)
         r := r + 1
      od
      Move v to Q.L[r] and v := v.f
      Link(Q, r)
      r := r + 1
   fi
   while v.left ≠ NIL or v.f ≠ NIL do
      if v.f ≠ NIL then
         Unlink v.f
         add the tree of rank r to Q.L[r]
         let v replace f
      else
         Unlink v.left
         add one tree to Q.L[r]
         insert one tree to the right of v
      fi
      Link(Q, r)
      r := r + 1
   od
   Q.L[r] := Q.L[r] − {v}
   for i := r to ⌊log₂(n + 1)⌋ − 1 do
      Unlink(Q, i + 1)
      Link(Q, i)
   od
end
```

Fig. 12. A sequential implementation allowing pipelining.

generated at processor zero too.

The basic idea is to represent a priority queue by a forest of heap ordered binomial trees as in Section 2, and to perform the priority queue operations sequentially in a loop that does constant work for each rank in increasing rank order. This approach then allows the operations to be performed in a pipelined fashion. In this section we require that a forest of binomial trees representing a priority queue satisfies the constraints:

$\mathbf{C}_1$: $n_i(Q) \in \{1, 2\}$, for $i = 0, \ldots, r(Q)$, and

$\mathbf{C}_2$: the minimum of the elements at roots of rank i is less than or equal to all the elements at roots of rank greater than i, for all $i = 0, \ldots, r(Q)$.

Notice that C_1 is a stronger requirement than A_1 in Section 2. In fact the sequence $n_0(Q), n_1(Q), \ldots$ is uniquely determined by the size of Q, because $n_0(Q) = 1$ if and only if $|Q|$ is odd, and that $n_i(Q)$ is uniquely given for $i > 0$ follows by induction. The following lemma gives the exact relation between $r(Q)$ and $|Q|$.

Lemma 18 $r(Q) = \lfloor \log_2(|Q| + 1) \rfloor - 1$.

PROOF. Because $n_i(Q)$ is uniquely given by $|Q|$, and $r(Q)$ is a nondecreasing function of $|Q|$ the lemma follows from

$$\sum_{i=0}^{r(Q)} 2^i \leq |Q| < \sum_{i=0}^{r(Q)+1} 2^i. \qquad \square$$

We first give a sequential implementation of the priority queue operations. Later we discuss how pipelining can be adopted to this implementation.

We assume that a forest is represented as follows. Each node is represented by a record having the fields:

e : the element associated to v,
left, right : pointers to the left and right siblings of v,
leftmost-child : a pointer to the leftmost child of v,
rightmost-child : a pointer to the rightmost child of v, and
f : a pointer to the parent of v, if v is the leftmost child of a node. Otherwise f is NIL.

Furthermore we have an array $Q.L$ of pointers to the roots.

A sequential implementation of the priority queue operations is shown in Figure 12 (the not quite trivial implementation of DELETE is due to the fact that the code is written in such a way that pipelining the code should be straightforward). The implementation uses the following two procedures.

LINK(Q, i) Links two trees from $Q.L[i] \setminus \min(Q.L[i])$ to one tree of rank $i+1$ that is added to $Q.L[i+1]$, provided $i \geq 0$ and $|Q.L[i]| \geq 3$.

UNLINK(Q, i) Unlinks the tree $\min(Q.L[i])$ and adds the resulting two trees to $Q.L[i-1]$, provided $i \geq 1$ and $|Q.L[i]| \geq 1$.

The implementation of the priority queue operations MAKEQUEUE, FIND-MIN, DECREASEKEY and EXTRACTMIN is obvious. The remaining priority queue operations are implemented as follows.

INSERT(Q, e) First a new rank zero tree is created and added to $Q.L[0]$. To reestablish C_1 we for each rank i in increasing rank order perform LINK(Q, i) once. It is straightforward to verify that $n_j(Q) \leq 2$ for $j \neq i$ and $n_i(Q) \leq 3$ before the ith iteration of the loop.

MELD(Q_1, Q_2) We incrementally merge the forests and perform LINK(Q_1, i) twice for each rank i. Two times are sufficient because before the linking in the ith iteration $|Q_1.L[i]| \leq 6$, where at most two trees come from the original $Q_1.L[i]$, two from $Q_2.L[i]$ and two from the linking of the roots in $Q_1.L[i-1]$.

DELETE(Q, e) Procedure DELETE proceeds in three phases. First all children of the node v to be removed are cut off and moved to $Q.L$. The node v is now a *dead* node without any children. In the second phase v is moved up thru the tree by iteratively unlinking a sibling or the parent of v. Finally the third phase reestablishes C_2 in case phase two removed $\min(Q.L[i])$ for some i.

In the first phase we after moving the rank i child of v to $Q.L[i]$ perform LINK(Q, i) once, which is sufficient to guarantee $n_i(Q) \leq 2$ because at most two rank i trees can have been added to $Q.L[i]$, one child of v and one tree from performing LINK($Q, i-1$). In the second phase we similarly for each subsequent rank i add one tree to $Q.L[i]$ and perform LINK(Q, i) once. If the dead node v has rank i, and v has a rank $i+1$ sibling, this sibling is unlinked and one tree replaces v, one tree is inserted into $Q.L[i]$, and v replaces the unlinked rank $i+1$ subtree. Otherwise v is the leftmost child and the parent of v is unlinked and the rank i tree is inserted into $Q.L[i]$ and v replaces its previous parent. When v becomes a root of rank r, we remove v which can make C_2 violated. Because at most one tree has been added to $Q.L[r]$ by LINK($Q, r-1$) we have $n_r(Q) \leq 2$. In the last phase we reestablish C_1 and C_2 for the remaining ranks $i \geq r$ by performing UNLINK($Q, i+1$) followed by LINK(Q, i). This phase is similar to the PARUNLINK and PARLINK calls done in the implementation of DELETE in Figure 11.

This finishes our description of the sequential data structure. Notice that each of the priority queue operations can be viewed as running in steps $i = 0, \ldots, \lfloor \log_2(n+1) \rfloor - 1$. Step i takes constant time and only accesses, links and unlinks nodes of rank i and $i+1$.

To implement the priority queues on a processor array a representation is required that is distributed among the processors. We make one essential modification to the above described data structure. Instead of having *rightmost-child* fields in node records, we instead maintain an array *rightmost-child* that for each node v stores a pointer to the rank zero child of v or to the v itself if v has rank zero. This only implies minor changes to the code in Figure 12. Notice that this modified representation only has pointers between nodes with rank difference at most one. For a record of rank r, *left* and f point to records of rank $r+1$, and *right* and *leftmost-child* point to records of rank $r-1$.

The representation we distribute on a processor array is now the following. We let processor p store all nodes of rank p. If the rank of a node is increased or decreased by one, the node record is respectively send to processor $p+1$ or $p-1$. In addition processor p stores $Q.L[p]$ for all priority queues Q. Notice that $Q.L[p]$ only contains pointers to records which are on processor p too. The array *rightmost-child* is stored at processor zero. Recall that *rightmost-child* only contains pointers to records of rank zero. The pointers that Delete and DecreaseKey take as arguments should now not be pointers to the nodes but the corresponding entries in the array *rightmost-child*.

With the above described representation step i of an operation only involves information stored at processors $\{i-1, i, i+1, i+2\}$ (processor $i-1$ and $i+2$ because back pointers have to be updated when linking and unlinking trees) which can be accessed in constant time. This immediately allows us to pipeline the operations by standard techniques, such that we for each new operation perform exactly four steps of each of the previous initiated but not yet finished priority queue operations. Notice that no latency is involved in performing operations: The answer to a FindMin query is known immediately, and for ExtractMin the minimum element is returned instantaneously, whereas the updating of the priority queue is done over the subsequent $\lfloor \log_2(n+1) \rfloor / 4 - 1$ priority queue operations.

Theorem 19 *On a processor array of size $\lfloor \log_2(n+1) \rfloor$ each of the operations* MakeQueue, Insert, Meld, FindMin, ExtractMin, Delete *and* DecreaseKey *can be supported in constant time.*

6 Multi priority queue operations

The priority queues we presented in the previous sections do not support the priority queue operation MultiExtractMin_k, that deletes the k smallest elements from a priority queue where k is fixed constant [6,24]. However, a possible solution is to apply the k-bandwidth idea [6,24] to the data structure presented in Section 2, by letting each node store k elements in sorted order instead of one element. The elements stored in a binomial tree are now required to satisfy *extended heap order*, *i.e.*, the k elements stored at a node are all required to be less than or equal to all the elements stored in the subtree rooted at that node. In the following we describe how to modify the data structure presented in Section 2 to support the operations MultiInsert_k, MultiExtractMin_k and Meld.

We need the following two lemmas.

Lemma 20 (Kruskal [20]) *On a CREW PRAM two sorted lists containing k elements each can be merged in $O(k/p + \log\log k)$ time with p processors.*

Lemma 21 (Cole [7]) *On a CREW PRAM a list containing k elements can be sorted in $O((k \log k)/p + \log k)$ time with p processors.*

Because each node now stores k elements, we have from Lemma 3 that $r(Q) \leq \lfloor \log_2(|Q|/k + 1) \rfloor - 1$. The new interpretation of constraint A_2 is that for each rank i there exists a root (denoted $\min(Q.L[i])$) which is less than or equal to all the remaining roots of rank i with respect to extended order, and all roots of rank $> i$ are larger than or equal to $\min(Q.L[i])$ with respect to extended order. For the modified data structure we need the following procedure.

COMPARE&SWAP(v_1, v_2) Given two roots v_1 and v_2 of equal rank the elements at v_1 and v_2 are rearranged, eventually by swapping v_1 and v_2, such that v_1 becomes less than or equal to v_2 with respect extended order, without violating the extended heap order of the trees.

The procedure COMPARE&SWAP can be implemented as follows. If the maximum element stored at v_1 is larger than the maximum element stored at v_2, we swap the trees rooted at v_1 and v_2. Next the merging procedure of Kruskal is used to merge the k elements at v_1 with the k elements at v_2. Finally the k smallest elements are moved to v_1 and the largest k elements are moved to v_2. On a CREW PRAM COMPARE&SWAP takes $O(k/p + \log\log n)$ time with p processors.

We extend the definition such that COMPARE&SWAP$(v_1, v_2, \ldots, v_c)$ should rearrange the roots such that v_1 after applying COMPARE&SWAP is less than or equal to $v_2, \ldots, v_c$ with respect to extended order (if $c \leq 1$ nothing is done). One possible implementation is to first recursively apply COMPARE&SWAP$(v_2, \ldots, v_c)$ (for $c \geq 3$) and then to apply COMPARE&SWAP(v_1, v_2).

We now give an implementation of the multi priority queue operations. The linking of two roots v_1 and v_2, is done by first performing COMPARE&SWAP(v_1, v_2) and then making v_2 the leftmost child of v_1. The unlinking of a tree proceeds as before. The procedures PARLINK(Q) and PARUNLINK(Q) proceed as described in Section 2, except that after both operations we for each rank p apply COMPARE&SWAP$(Q.L[p])$ to make $\min(Q.L[p])$ well defined. Because at no time $|Q.L[p]| > 6$ for any p, we have that the modified procedures PARLINK and PARUNLINK take $O((k \log \frac{n}{k})/p + \log\log k)$ time with p processors.

MULTIINSERT$_k(Q, e_1, \ldots, e_k)$ First a new node v of rank zero is created containing the k new elements by applying the sorting algorithm of Cole. To make $\min(Q.L[0])$ well defined, we apply COMPARE&SWAP$(\min(Q.L[0]), v)$.

```
Proc MultiInsert_k(Q, e_1, ..., e_k)
   v := new-node(Sort(e_1, ..., e_k))
   Compare&Swap(min(Q.L[0]), v)
   Q.L[0] := Q.L[0] ∪ {v}
   ParLink(Q)
end

Proc MultiExtractMin_k(Q)
   (e_1, ..., e_k) := min(Q.L[0])
   Q.L[0] := Q.L[0] \ min(Q.L[0])
   ParUnlink(Q)
   ParLink(Q)
   return (e_1, ..., e_k)
end

Proc Meld(Q_1, Q_2)
   for p := 0 to ⌊log_2(n/k + 1)⌋ − 1 pardo
      Compare&Swap(min(Q_1.L[p]), min(Q_2.L[p]))
      Q_1.L[p] := Q_1.L[p] ∪ Q_2.L[p]
   od
   do 3 times ParLink(Q_1)
end
```

Fig. 13. CREW PRAM multi insertion and deletion operations.

Finally we apply ParLink once.

MultiExtractMin$_k(Q)$ The k elements to be returned are stored at $\min(Q.L[0])$. Otherwise the procedure is identical to the procedure described in Section 2.

Meld(Q_1, Q_2) To make $\min(Q_1.L[p])$ well defined after merging the two forest, we first apply Compare&Swap$(\min(Q_1.L[p]), \min(Q_2.L[p]))$. Thereafter the procedure proceeds as in Section 2.

The correctness of the above procedures follows as in Section 2. Pseudo code for the operations is given in Figure 13. From the previous discussion we have the following theorem.

Theorem 22 *On a CREW PRAM priority queues exist, supporting* MultiInsert$_k$ *in* $O((k\log k + k\log\frac{n}{k})/p + \log k)$ *time with* p *processors, and* MultiExtractMin$_k$ *and* Meld *in* $O((k\log\frac{n}{k})/p + \log\log k)$ *time with* p *processors.*

7 Conclusion

We have presented new implementations of parallel priority queues. Whereas the priority queues of [6,24–26] are based on traditional sequential heaps [14,33] and leftist heaps [30], our priority queues are designed specifically for the parallel setting. Our parallel priority queues are the first to support both MELD and DELETEMIN in constant time with $O(\log n)$ processors.

Our implementation of DECREASEKEY in Section 3 requires $\Theta(\log n)$ work. In the sequential setting it is known that DECREASEKEY can be supported in worst-case constant time [3,12]. It remains an open problem to support DECREASEKEY with constant work without increasing the asymptotic time and work bounds of the other operations, and it also remains an open problem to efficiently support multi DECREASEKEY operations. Initial research towards supporting multi DECREASEKEY operations has be done by Brodal *et al.* [4].

References

[1] Gerth Stølting Brodal. Fast meldable priority queues. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, volume 955 of *Lecture Notes in Computer Science*, pages 282–290. Springer Verlag, Berlin, 1995.

[2] Gerth Stølting Brodal. Priority queues on parallel machines. In *Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 1097 of *Lecture Notes in Computer Science*, pages 416–427. Springer Verlag, Berlin, 1996.

[3] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 52–58, 1996.

[4] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A parallel priority data structure with applications. In *Proc. 11th Int. Parallel Processing Symposium (IPPS)*, pages 689–693, 1997.

[5] Svante Carlsson, Patricio V. Poblete, and J. Ian Munro. An implicit binomial queue with constant insertion time. In *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer Verlag, Berlin, 1988.

[6] Danny Z. Chen and Xiaobo Hu. Fast and efficient operations on parallel priority queues (preliminary version). In *Algorithms and Computation: 5th International Symposium, ISAAC '93*, volume 834 of *Lecture Notes in Computer Science*, pages 279–287. Springer Verlag, Berlin, 1994.

[7] Richard Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):130–145, 1988.

[8] S. K. Das, M. C. Pinotti, and F. Sarkar. Optimal and Load Balanced Mapping of Parallel Priority Queues in Hypercubes. *IEEE Transcations on Parallel and Distributed Systems*, 7(6):555–564, 1996.

[9] S. K. Das and M. C. Pinotti. $O(\log \log N)$ Time Algorithms for Hamiltonian-Suffix and Min-Max-Pair Heap Operations on the Hypercube. *Journal of Parallel and Distributed Computing*, 48(2):200–211, 1998.

[10] Paul F. Dietz. Heap construction in the parallel comparison tree model. In *Proc. 3rd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 621 of *Lecture Notes in Computer Science*, pages 140–150. Springer Verlag, Berlin, 1992.

[11] Paul F. Dietz and Rajeev Raman. Very fast optimal parallel algorithms for heap construction. In *Proc. 6th Symposium on Parallel and Distributed Processing*, pages 514–521, 1994.

[12] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[13] Rolf Fagerberg. A generalization of binomial queues. *Information Processing Letters*, 57:109–114, 1996.

[14] Robert W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.

[15] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert Endre Tarjan. The pairing heap: A new form of self–adjusting heap. *Algorithmica*, 1:111–129, 1986.

[16] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proc. 25th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 338–346, 1984.

[17] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[18] Chan M. Khoong. Optimal parallel construction of heaps. *Information Processing Letters*, 48:159–161, 1993.

[19] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[20] Clyde P. Kruskal. Searching, merging and sorting in parallel computations. *IEEE Transactions on Computers*, C-32(10):942–946, 1983.

[21] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

[22] Kurt Mehlhorn and Athanasios K. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity.* MIT Press/Elsevier, 1990.

[23] Vincenzo A. Crupi, Sajal K. Das, and Maria C. Pinotti. Parallel and distributed meldable priority queues based on binomial heaps. In *Proceedings of the Int. Conference on Parallel Processing*, Bloomingdale, Illinois, 255–262, 1996.

[24] Maria C. Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40:33–40, 1991.

[25] Maria C. Pinotti and Geppino Pucci. Parallel algorithms for priority queue operations. *Theoretical Computer Science*, 148(1):171–180, 1995.

[26] Abhiram Ranade, Szu-Tsung Cheng, Etienne Deprit, Jeff Jones, and Sun-Inn Shih. Parallelism and locality in priority queues. In *Proc. 6th Symposium on Parallel and Distributed Processing*, pages 490–496, 1994.

[27] Nageswara S. V. Rao and Weixiong Zhang. Building heaps in parallel. *Information Processing Letters*, 37:355–358, 1991.

[28] V. Nageshwara Rao and Vipin Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37(12):1657–1665, 1988.

[29] Jörg-Rüdiger Sack and Thomas Strothotte. An algorithm for merging heaps. *Acta Informatica*, 22:171–186, 1985.

[30] Robert Endre Tarjan. *Data Structures and Network Algortihms.* Society for Industrial and Applied Matehmatics, Philadelphia, Pennsylvania, 1983.

[31] Jan van Leeuwen. The composition of fast priority queues. Technical Report RUU-CS-78-5, Department of Computer Science, University of Utrecht, 1978.

[32] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[33] John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.