Priority Queues on Parallel Machines

Gerth Stølting Brodal[*]

BRICS[**], Computer Science Department, Aarhus University,
Ny Munkegade, DK-8000 Århus C, Denmark.

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 and k elements can be inserted into a priority queue in time $O(\log k)$ with $O((\log n + k)/\log k)$ processors. With a slowdown of $O(\log\log n)$ in time the priority queues adopt to the EREW PRAM by only increasing the required work by a constant factor. 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.

1 Introduction

The construction of priority queues is a classical topic in data structures. Some references are [1, 2, 6, 7, 8, 9, 19, 20]. A historical overview of implementations can be found in [13]. Recently several papers have also considered how to implement priority queues on parallel machines [3, 4, 5, 11, 15, 16, 17, 18]. In this paper we focus on how to achieve optimal speedup for the individual priority queue operations known from the sequential setting [16, 17]. The operations we support are all the commonly needed priority queue operations from the sequential setting [13] and the parallel insertion of several elements at the same time [3, 15].

MakeQueue Creates and returns a new empty priority queue.

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

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

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

ExtractMin(Q) Deletes and returns the minimum element in priority queue Q.

[*] Supported by the Danish Natural Science Research Council (Grant No. 9400044). This research was done while visiting the Max-Planck Institut für Informatik, Saarbrücken, Germany. E-mail: `gerth@daimi.aau.dk`.

[**] Basic Research in Computer Science, a Centre of the Danish National Research Foundation.

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

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

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

MultiInsert$(Q,e_1,\ldots,e_k)$ Inserts elements $x_1,\ldots,x_k$ into priority queue Q.

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. We assume w.l.o.g. that n is of the form 2^k. This guarantees that $\log n$ is an integer.[1] Our main result is:

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 time* $O(\log n)$ *with* $O(n/\log n)$ *processors and* MultiInsert *in time* $O(\log k)$ *with* $O((\log n + k)/\log k)$ *processors.*

Table 1 lists the performance of different implementations adopting parallelism to priority queues. Several papers consider how to build heaps [7] optimally in parallel [4, 5, 11, 18]. On an EREW PRAM an optimal construction time of $O(\log n)$ is achieved in [18] and on a CRCW PRAM an optimal construction time of $O(\log\log n)$ is achieved in [5].

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 time $O(\log\log n)$. The bounds we achieve matches those of [3] for k equal one and those of [14]. See Table 1.

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

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 [12]. Recently Ranade *et al.* [17] showed how to achieve the same bounds on a processor array with only $O(\log n)$ processors. In Sect. 5 we describe how the 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 $O(\log n)$

[1] All logarithms in this paper are to the base two.

	[16]	[14]	[15]	[3]	[17]	This paper
Model	EREW	EREW[2]	CREW	EREW	Array	CREW
FindMin	1	$\log\log n$	1	1	1	1
Insert	$\log\log n$	$\log\log n$	–	–	1	1
ExtractMin	$\log\log n$	$\log\log n$	–	–	1	1
Meld	–	$\log\log n$	$\log\frac{n}{k}+\log\log k$	$\log\log\frac{n}{k}+\log k$	–	1
Delete	–	$\log\log n$	–	–	–	1
DecreaseKey	–	$\log\log n$	–	–	–	1
Build	$\log n$	–	$\frac{n}{k}\log k$	$\log\frac{n}{k}\log k$	–	$\log n$
MultiInsert	–	–	$\log\frac{n}{k}+\log k$	$\log\log\frac{n}{k}+\log k$	–	$\log k$
MultiDelete	–	–	$\log\frac{n}{k}+\log\log k$	$\log\log\frac{n}{k}+\log k$	–	–

Table 1. Performance of different parallel implementations of priority queues.

processors, supporting the operations MakeQueue, Insert, Meld, FindMin, ExtractMin, Delete and DecreaseKey in constant time. This extends the result of [17].

The priority queues we present in this paper do not support the operation MultiDelete, that deletes the k smallest elements from a priority queue (where k is fixed [3, 15]). However, a possible solution is to apply the k-bandwidth idea used in [3, 15], by letting each node contain k elements instead of one. If we apply the idea to the data structure in Sect. 2 we get the time bounds in Theorem 3, improving upon the bounds achieved in [15], see Table 1. We omit the details and refer the reader to [15].

Theorem 3. *On a CREW PRAM priority queues exist, supporting* MultiInsert *in time* $O(\log k)$, MultiDelete *and* Meld *in time* $O(\log\log k)$, *and* Build *in time* $O(\log k + \log\frac{n}{k}\log\log k)$.

2 Meldable priority queues

In this section we describe how to implement the priority queue operations MakeQueue, Insert, Meld, FindMin and ExtractMin in constant time on a CREW PRAM with $O(\log n)$ processors. In Sect. 3 we describe how to extend the repertoire of priority queue operations to include Delete and DecreaseKey.

The priority queues in this section are based on heap ordered binomial trees [19]. Throughout this paper we assume a one to one mapping between tree nodes and priority queue elements.

Binomial trees are defined as follows. A binomial tree of *rank* zero is a single node. A binomial tree of rank $r > 0$ is achieved from two binomial trees of rank $r-1$ by making one of the roots a son of the other root. It follows by induction that a binomial tree of rank r contains exactly 2^r nodes and that a node of rank

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

r has exactly one son of each of the ranks $0, \ldots, r-1$. Throughout this section a tree denotes a heap ordered binomial tree.

A priority queue is represented by a forest of binomial trees. In the following we let the largest ranked tree be of rank $r(Q)$, we let $n_i(Q)$ denote the number of trees of rank i and we let $n_{\max}(Q)$ denote the value $\max_{0 \le i \le r(Q)} n_i(Q)$. We require that a priority queue satisfies the constraints:

A$_1$: $n_i(Q) \in \{1, 2, 3\}$ for $i = 0, \ldots, r(Q)$, and
A$_2$: the minimum root of rank i is smaller than all roots of rank larger than i.

It follows from A$_2$ that the minimum root of rank zero is the minimum element.

A priority queue is stored as follows. Each node v in a priority queue is represented by a record consisting of:

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

For each priority queue Q an array $Q.L$ is maintained of size $1 + \log n$ of pointers to linked lists of roots of equal rank. By A$_1$, $|Q.L[i]| \le 3$ for all i. Notice that the chosen representation for storing the sons of a node allows two nodes of equal rank to be linked in constant time by one processor. The required space for a priority queue is $O(n)$.

Two essential procedures used by our algorithms are the procedures ParLink and ParUnlink in Fig. 1. In parallel ParLink for each rank i links two trees of rank i to one tree of rank $i+1$, if possible. By requiring that the trees of rank i that are linked together are different from $\min(Q.L[i])$, A$_2$ does not become violated. Let $n'_i(Q)$ denote the value of $n_i(Q)$ after performing ParLink. If $n_i(Q) \ge 3$ before performing ParLink then $n'_i(Q) \le n_i(Q) - 2 + 1$, because processor i removes two trees of rank i and processor $i-1$ adds at most one tree of rank i. Otherwise $n'_i(Q) \le n_i(Q) + 1$. This implies that $n'_{\max}(Q) \le \max\{3, n_{\max}(Q) - 1\}$. The equality states that if the maximum number of trees of equal rank is larger than three, then an application of ParLink decreases this value by at least one. The procedure ParUnlink unlinks the minima of all $Q.L[i]$. All $n_i(Q)$ at most increase by one except for $n_0(Q)$ that can increase by two. Notice that the new minimum of $Q.L[i]$ is less than or equal to the old minimum of $Q.L[i+1]$. This implies that if A$_2$ is satisfied before performing ParUnlink then A$_2$ is also satisfied after the unlinking. Notice that ParLink and ParUnlink can be performed on an EREW PRAM with $O(\log n)$ processors in constant time if all processors know Q.

The priority queue operations can now be implemented as:

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

Insert(Q, e) A new tree of rank zero containing e is created and added to $Q.L[0]$. To avoid $n_{\max}(Q) > 3$, ParLink(Q) is performed once.

```
Proc ParLink($Q$)
  for $p := 0$ to $\log n - 1$ 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 $Q.L[p+1]$

Proc ParUnlink($Q$)
  for $p := 1$ to $\log n$ pardo
    if $n_p(Q) \geq 1$ then
      Unlink $\min(Q.L[p])$ and add the resulting two trees to $Q.L[p-1]$
```

Fig. 1. Parallel linking and unlinking binomial trees.

```
Proc FindMin($Q$)
  return $\min(Q.L[0])$

Proc Insert($Q, e$)
  $Q.L[0] := Q.L[0] \cup \{new\text{-}node(e)\}$
  ParLink($Q$)

Proc Meld($Q_1, Q_2$)
  for $p := 0$ to $\log n$ pardo
    $Q_1.L[p] := Q_1.L[p] \cup Q_2.L[p]$
  do 3 times ParLink($Q_1$)

Proc MakeQueue
  $Q :=new\text{-}queue$
  for $p := 0$ to $\log n$ pardo $Q.L[p] := \emptyset$
  return $Q$

Proc ExtractMin($Q$)
  $e := \min(Q.L[0])$
  $Q.L[0] := Q.L[0] \setminus \{e\}$
  ParUnlink($Q$)
  ParLink($Q$)
  return $e$
```

Fig. 2. CREW PRAM priority queue operations.

Meld(Q_1, Q_2) First $Q_2.L$ is merged into $Q_1.L$ by letting processor p set $Q_1.L[p]$ to $Q_1.L[p] \cup Q_2.L[p]$. The resulting forest satisfies $n_{\max}(Q_1) \leq 6$. Performing ParLink(Q_1) three times reestablishes A_1.

FindMin(Q) The minimum element in priority queue Q is $\min(Q.L[0])$.

ExtractMin(Q) First the minimum element $\min(Q.L[0])$ is removed. Performing ParUnlink once guarantees that A_2 is satisfied, especially that the new minimum element is contained in $Q.L[0]$, because the new minimum element was either already contained in $Q.L[0]$ or it was the minimum element in $Q.L[1]$. Finally ParLink performed once reestablishes A_1.

A pseudo code implementation for a CREW PRAM based on the previous discussion is shown in Fig. 2. Notice that the only part of the code requiring concurrent read is to "broadcast" the values of Q, Q_1 and Q_2 to all the processors. Otherwise the code only requires an EREW PRAM. From the fact that ParLink and ParUnlink can be performed in constant time with $O(\log n)$ processors we get:

Theorem 4. *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 $O(\log n)$ processors.*

3 Priority queues with deletions

In this section we extend the repertoire of supported priority queue operations to include DELETE and DECREASEKEY. Notice that DECREASEKEY(Q, e, e') can be implemented as DELETE(Q, e) followed by INSERT(Q, e').

The priority queues in this section are based on heap ordered trees defined as follows. A rank zero tree is a single node. A rank r tree is a tree where the root has exactly five sons of each of the ranks $0, 1, \ldots, r-1$. A tree of rank r can be created by linking six trees of rank $r-1$ by making the five larger roots sons of the smallest root.

The efficiency we achieve for DELETE and DECREASEKEY is due to the concept of *holes*. A hole of rank r in a tree is a location in the tree where a son of rank r is missing.

We represent a priority queue by a forest of trees with holes. Let $r(Q), n_i(Q)$ and $n_{\max}(Q)$ be defined as in Sect. 2. We require that:

B$_1$: $n_i(Q) \in \{1, 2, \ldots, 7\}$, for $i = 1, \ldots, r(Q)$,
B$_2$: the minimum root of rank i is smaller than all roots of rank larger than i,
B$_3$: at most two holes have equal rank.

Temporary while performing MELD we allow the number of holes of equal rank to be at most four. The requirement that a node of rank r has five sons of each of the ranks $0, \ldots, r-1$ implies that at least one son of each rank is not replaced by a hole. This implies that the subtree rooted at a node has at least size 2^r and therefore the largest possible rank is at most $\log n$.

A priority queue is stored 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 father of v, and
L : an array of size $\log n$ of pointers to linked lists of sons of equal rank.

For each priority queue Q two arrays $Q.L$ and $Q.H$ are maintained of size $1 + \log n$. $Q.L$ 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 son 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. Notice that the space required is $O(n \log n)$. By using worst case constant time extendible arrays to store the required arrays such that $|v.L| = v.r$, the space requirement can be reduced to $O(n)$. For simplicity we in the following assume that $|v.L| = \log n$ for all v.

The procedures PARLINK and PARUNLINK have to be modified such that linking and unlinking involves six trees and such that PARUNLINK catches holes to be removed from $Q.H$. PARLINK now satisfies $n'_{\max}(Q) \leq \max\{7, n_{\max}(Q) - 5\}$, and PARUNLINK $n'_i(Q) \leq n_i(Q) + 5$ for $i > 0$ and $n'_0(Q) \leq n_0(Q) + 6$.

We now describe a procedure FixHoles that reduces the number of holes similar to how ParLink reduces the number of trees. 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 $|Q.H[p]| < 2$ the processor does nothing. Otherwise it considers two holes in $Q.H[p]$. Recall that all holes have at least one real tree node of rank p as a brother. If the two holes have different fathers, we swap one of the holes with a brother of the other hole. This makes both holes have the same father f. By choosing the largest node among the two holes' brothers as the swap node 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 brother b of rank $p+1$. Notice that b has at least three sons of rank p because we allowed at most four holes of rank p. We can now cut off b and all sons of b of rank p. By assigning b the rank p we only create one hole of rank $p+1$. We can now eliminate the two original holes by replacing them with two previous sons of b. At most four trees remain to be added to $Q.L[p]$. 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. This creates a new hole of rank $p+1$. We can now cut off all sons 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 where $f = \min(Q.L[p+1])$. By performing ParUnlink and ParLink once the two holes disappear. To compensate for the created new trees we finally perform ParLink once.

The priority queue operations can now be implemented as follows.

MakeQueue Allocate a new priority queue Q and assign the empty set to all $Q.L[i]$ and $Q.H[i]$.

Insert(Q, e) Create a tree of rank zero containing e and add this tree to $Q.L[0]$. 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. Performing ParLink(Q_1) twice followed by FixHoles(Q_2) twice reestablishes B_1 and B_3.

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

ExtractMin(Q) First perform FindMin and then perform Delete on the 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 $Q.H$. Merge $v.L$ into $Q.L$ and remove all appearances of v from $Q.H$. Notice that only for $i = v.r$, $\min(Q.L[i])$ can change and this only happens if e was $\min(Q.L[i])$. Unlinking $\min(Q.L[i])$ for $i = v.r+1, \ldots, r(Q)$ reestablishes B_2. Finally perform ParLink twice to reestablish B_1 and FixHoles once to reestablish B_3.

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 Fig. 3. Notice that the only part of the code that requires

```
**Proc** MakeQueue
  $Q :=$ *new-queue*
  **for** $p := 0$ **to** $\log n$ **pardo**
    $Q.L[p], Q.H[p] := \emptyset$
  **return** $Q$

**Proc** FindMin($Q$)
  **return** $\min(Q.L[0])$

**Proc** Insert($Q, e$)
  $Q.L[0] := Q.L[0] \cup \{new\text{-}node(e)\}$
  ParLink($Q$)

**Proc** Meld($Q_1, Q_2$)
  **for** $p := 0$ **to** $\log n$ **pardo**
    $Q_1.L[p] := Q_1.L[p] \cup Q_2.L[p]$
    $Q_1.H[p] := Q_1.H[p] \cup Q_2.H[p]$
  **do** 2 **times** ParLink($Q_1$)
  **do** 2 **times** FixHoles($Q_1$)

**Proc** DecreaseKey($Q, e, e'$)
  Delete($Q, e$)
  Insert($Q, e'$)
```

```
**Proc** ExtractMin($Q$)
  $e :=$ FindMin($Q$)
  Delete($Q, e$)
  **return** $e$

**Proc** Delete($Q, e$)
  $v :=$ the node containing $e$
  **if** $v.f \neq$ NIL **then**
    $Q.H[v.r] := Q.H[v.r] \cup \{v.f\}$
    $v.f.L[v.r] := v.f.L[v.r] \setminus \{v\}$
  **for** $p := 0$ **to** $\log n$ **pardo**
    **for** $u \in v.L[p]$ **do** $u.f :=$ NIL
    $Q.L[p] := Q.L[p] \cup v.L[p]$
    $Q.H[p] := Q.H[p] \setminus \{v\}$
  **for** $p := 0$ **to** $\log n$ **pardo**
    **if** $n_p(Q) \geq 1$ **and** $p > v.r$ **then**
      $Q.H[p-1] := Q.H[p-1] \setminus \min(Q.L[p])$
      Unlink $\min(Q.L[p])$ and
      add the resulting trees to $Q.L[p-1]$
  **do** 2 **times** ParLink($Q$)
  FixHoles($Q$)
```

Fig. 3. CREW PRAM priority queue operations.

concurrent read is the "broadcasting" of the parameters of the procedures and $v.r$ in Delete. The rest of the code does very local computing, in fact processor p only accesses entries p and $p \pm 1$ of arrays, and that these local computations can be done in constant time with $O(\log n)$ processors on an EREW PRAM.

Theorem 5. *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.*

4 Building priority queues

In this section we describe how to perform Build$(x_1, \ldots, x_n)$ for the priority queues in Sect. 3. 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 [10] we have an $\Omega(\log n)$ lower bound on the construction time on a CREW PRAM. We now give a matching upper bound on an EREW PRAM.

First a collection of trees is constructed satisfying B_1 and B_3 but not B_2. 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. There are at most $O(\log n)$ iterations because each iteration reduces the number of trees by a factor six. The resulting forest satisfies B_1 and B_3. It is easy to see that the above construction can be done in time $O(\log n)$ with $O(n/\log n)$ processors on an EREW PRAM.

To establish B_2 we $\log n$ times perform ParUnlink followed by ParLink. By induction it follows that in the ith iteration all $Q.L[j]$ where $j \geq \log n - i$ satisfy B_2. This finishes the construction of the priority queue. The last step of the construction requires time $O(\log n)$ with $O(\log n)$ processors. We conclude that:

Theorem 6. *On an EREW PRAM a priority queue containing n elements can be constructed optimally with $O(n/\log n)$ processors in time $O(\log n)$.*

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 corollary below. Notice that k does not have to be fixed as in [3, 15].

Corollary 7. *On a CREW PRAM* MultiInsert *can be performed in time $O(\log k)$ with $O((\log n + k)/\log k)$ processors.*

5 Pipelined priority queue operations

The priority queues in Sect. 2, 3 and 4 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. On a processor array we assume that all requests are entered at processor zero and that output is generated at processor zero too [17].

The basic idea is to represent a priority queue by a forest of heap ordered binomial trees as in Sect. 2, and to perform the operations sequentially in a loop that does constant work for each rank in increasing rank order. This approach then allows us to pipeline the operations. We require that a forest of binomial trees representing a priority queue satisfies:

C_1 : $n_i(Q) \in \{1, 2\}$, for $i = 1, \ldots, r(Q)$,
C_2 : the minimum root of rank i is smaller than all roots of rank larger than i.

Notice that C_1 is stronger than A_1 in Sect. 2. Sequential implementations of the priority queue operations are shown in Fig. 4. We assume a similar representation as in Sect. 3. The pseudo code uses the following two procedures similar to those used in Sect. 2.

$\textsc{Link}(Q, i)$ 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 $|Q.L[i]| \geq 3$.

```
Proc MakeQueue
  $Q$ :=new-queue
  for $p := 0$ to $\log n$ do $Q.L[p] := \emptyset$
  return $Q$

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

Proc Insert($Q, e$)
  $Q.L[0] := Q.L[0] \cup \{new\text{-}node(e)\}$
  for $i := 0$ to $\log n$ do Link($Q, i$)

Proc Meld($Q_1, Q_2$)
  for $i := 0$ to $\log n$ do
    $Q_1.L[i] := Q_1.L[i] \cup Q_2.L[i]$
    do 2 times Link($Q_1, i$)

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

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

Proc Delete($Q, e$)
  $v$ := the node containing $e$
  for $i := 0$ to $v.r - 1$ do
    Move $v.L[i]$ to $Q.L[i]$
    Link($Q, i$)
  $r, f := v.r, v.f$
  Remove node $v$
  while $f \neq$ NIL do
    if $f.r = r + 1$ then
      $f.r := f.r - 1$
      Move $f$ to $Q.L[r]$ and
      $f := f.f$
    else
      Unlink $f.L[r + 1]$ and add
      one tree to $f.L[r]$ and
      one tree to $Q.L[r]$
    Link($Q, i$)
    $r := r + 1$
  for $i := r$ to $\log n$ do
    Unlink($Q, i + 1$)
    do 2 times Link($Q, i$)
```

Fig. 4. A sequential implementation allowing pipelining.

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 $|Q.L[i]| \geq 1$.

Each of the priority queue operations can be viewed as running in steps $i = 0, \ldots, \log n$. Step i only accesses, creates and destroys nodes of rank i and $i + 1$. Notice that requirement C_1 implies that Meld only has to perform Link two times for each rank, whereas the implementation of Meld in Fig. 2 has to do the corresponding linking three times. Otherwise the only interesting procedure is Delete. Procedure Delete proceeds in three phases. First all sons of the node to be removed are cut off and moved to $Q.L$. In the second phase the hole created is eliminated by moving it up thru the tree by unlinking the brother node of the hole's current position or unlinking the father node of the hole. Finally the third phase reestablishes C_2 in case phase two removed $\min(Q.L[i])$ for some i. This phase is similar to the last for loop in the implementation of Delete in Fig. 3.

The pseudo code given in Fig. 4 assumes the same representation for nodes as in Sect. 3. To implement the priority queues on a processors array a representation is required that is distributed among the processors. The canonical distribution is to let processor p store nodes of rank p.

The representation we distribute is the following. Assume that the sons of a node are ordered from right-to-left in increasing rank order (this allows us to talk about the leftmost and rightmost sons of a node). A node v is represented by a record with the fields:

e : the element associated to v,
r : the rank of v,
left, right : pointers to the left and right brothers of v,
leftmost-son : a pointer to the leftmost son of v,
f : a pointer to the father of v, if v is the leftmost son. Otherwise NIL.

The array $Q.L$ is replaced by linked lists. Finally an array *rightmost-son* is maintained that for each node stores a pointer to the rank zero son of the node or to the node itself if it has rank zero. Notice that this representation only has pointers between nodes with rank difference at most one.

It is straightforward to modify the code given in Fig. 4 to this new representation. The only essential difference is when performing DELETE. The first rank zero son of v to be moved to $Q.L$ is found by using the array *rightmost-son*. The succeeding sons are found by using the *left* pointers.

On a processor array we let processor p store all nodes of rank p. In addition processor p stores $Q.L[p]$ for all priority queues Q. The array *rightmost-son* is stored at processor zero. The "locations" that DELETE and DECREASEKEY refer to are now not the nodes but the corresponding entries in the *rightmost-son* array.

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 in the involved linked lists) that can be accessed in constant time. This immediately allows us to pipeline the operations, such that we for each new operation perform exactly four steps of each of the previous operations. Notice that no latency is involved in performing the queries: The answer to a FINDMIN query is known immediately.

Theorem 8. *On a processor array of size $O(\log n)$ each of the operations* MAKEQUEUE, INSERT, MELD, FINDMIN, EXTRACTMIN, DELETE *and* DECREASEKEY *can be supported in constant time.*

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

4. 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.
5. 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.
6. James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.
7. Robert W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.
8. Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self–adjusting heap. *Algorithmica*, 1:111–129, 1986.
9. Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proc. 25rd Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 338–346, 1984.
10. Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.
11. C. M. Khoong. Optimal parallel construction of heaps. *Information Processing Letters*, 48:159–161, 1993.
12. F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.
13. 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.
14. Maria Cristina Pinotti, Sajal K. Das, and Vincenzo A. Crupi. Parallel and distributed meldable priority queues based on binomial heaps. In *Int. Conference on Parallel Processing*, 1996.
15. Maria Cristina Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40:33–40, 1991.
16. Maria Cristina Pinotti and Geppino Pucci. Parallel algorithms for priority queue operations. In *Proc. 3rd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 621 of *Lecture Notes in Computer Science*, pages 130–139. Springer Verlag, Berlin, 1992.
17. A. Ranade, S. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and locality in priority queues. In *Proc. 6th Symposium on Parallel and Distributed Processing*, pages 490–496, 1994.
18. Nageswara S. V. Rao and Weixiong Zhang. Building heaps in parallel. *Information Processing Letters*, 37:355–358, 1991.
19. Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.
20. J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

This article was processed using the LaTeX macro package with LLNCS style