Parallel priority queues based on binomial heaps

Sajal K. Das a,*, M. Cristina Pinotti b

a Department of Computer Science and Engineering, The University of Texas at Arlington, P.O. Box 19015, Arlington, TX 76019-0015, USA
b Department of Mathematics, University of Trento, Povo (TN), Italy

Received 5 February 1998; received in revised form 15 January 2000; accepted 15 January 2000

Abstract

We present an optimal parallel implementation of a meldable priority queue based on the binomial heap data structure. Our main result is an interesting application of the parallel computation of carry bits in a full adder logic to binomial heaps, thus optimizing the parallel time complexity of the Union (often called melding) of two queues. The Union operation as well as Insert, Min, Extract-Min and Multiple-Insert require doubly logarithmic time and are work-optimal, employing \( p \in \Theta(\log n / \log \log n) \) processors on the EREW–PRAM model. Parallel algorithms for Delete and Decrease-Key operations work by postponing the global effect of a node deletion/update, and achieve doubly logarithmic time and work-optimality in the amortized sense. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Binomial heaps; Carry adder; Insert and delete operations; Meldable priority queues; Parallel data structures; Time complexity

1. Introduction

Priority queues are useful data structures for solving numerous problems based on the principle of selecting the smallest (or largest) element from a totally ordered set. A priority queue, \( \mathcal{Q} \), supports standard operations like Make-Queue(\( \mathcal{Q} \)) to create and return an empty queue; insert(\( \mathcal{Q}, x \)) to insert a node \( x \) into \( \mathcal{Q} \); Min(\( \mathcal{Q} \)) to return a
pointer to the node with the minimum key in $Q$; \texttt{Extract-Min}($Q$) to delete from $Q$ the node with the minimum key and return a pointer to it. A queue is called \textit{meldable} if it supports the operation \texttt{Union}($Q_1$, $Q_2$) to return a merger of the queues $Q_1$ and $Q_2$. In this paper, we consider queues which also support two other operations, namely, \texttt{Delete}($Q$, $x$) to delete a node $x$ from $Q$, and \texttt{Decrease-Key}($Q$, $x$, $k$) to change the current key value of a queue node $x$ to the new value $k$.

Various data structures such as binary heaps, leftist \footnote{A fully binary tree $T$ is a \textit{leftist} tree if for every node $x \in T$, the rightmost path of the subtree of $T$ rooted at $x$ is the shortest path from $x$ to a leaf.} heaps and binomial heaps have been proposed for sequential implementation of priority queues \cite{3}. Both leftist and binomial heaps are meldable in nature. For an $n$-node priority queue based on a binomial heap, the operation \texttt{Make-Queue} requires $O(1)$ time while all other operations require $O(\log n)$ time in the worst-case (see Table 1). Note that binary and leftist heaps consist of a unique tree, the root of which stores the smallest key. A binomial heap, on the other hand, consists of several trees, and any of their roots can store the smallest key. Hence the sequential implementation of the \texttt{Min} operation on binomial heaps do not compare favorably with binary or leftist heaps.

Parallel algorithms \cite{2,4,7,8,10} have also been proposed for implementing priority queues on the exclusive-read and exclusive-write (EREW), parallel random access machine (PRAM) model. Table 2 gives a performance summary of these algorithms. (Here $p$ is the number of processors.) As will be discussed later, the critical operation for parallel implementation of binomial heaps is the \texttt{Union} since it may construct long linking chains, each link ineluctably dependent on the previous one. We overcome this problem by exploiting the similarity of this problem with the propagation of carry bits in a full adder logic. The time complexity for the \texttt{Delete} and \texttt{Decrease-Key} operations for parallel binomial heaps are amortized, and so marked by asterisk *. A preliminary version of these results appeared in \cite{5}.

### 2. Preliminaries

This section defines a binomial heap data structure, outlines its representation and sequential implementation.

---

Table 1

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary heap</th>
<th>Leftist heap</th>
<th>Binomial heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Make-Heap}</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>\texttt{Insert}</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>\texttt{Min}</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>\texttt{Extract-Min}</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>\texttt{Union}</td>
<td>$\Theta(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>\texttt{Decrease-Key}</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>\texttt{Delete}</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Definition 1. A binomial tree, $B_k$, of degree $k$ consists of $2^k$ nodes and is recursively defined as follows [3,9]:

- The tree $B_0$ is a single node.
- For $k > 0$, $B_k$ consists of two binomial trees $B_{k-1}$ linked such that the root of one tree is the leftmost child of the other root.

There are exactly $\binom{k}{i}$ nodes at depth $i$ of the tree $B_k$, where $0 \leq i \leq k$. Also the root $r$ of $B_k$ has larger degree than any of its descendents. If the children of $r$ are numbered (starting from 0) from right to left, the $i$th child is the root of a binomial subtree $B_i$. An example of $B_3$ is the leftmost tree in Fig. 1. In order to define the binomial heap, let us recall that:

Definition 2. A tree maintains the min-heap order if each node stores a key which is smaller than or equal to the keys stored in its children.

Definition 3. An $n$-node binomial heap, $H$, is a collection of binomial trees such that each tree maintains the min-heap ordering, and there is at most one binomial tree $B_i$ for each degree $i$, where $0 \leq i \leq \lfloor \log n \rfloor$ in $H$.

Table 2
Parallel time complexity of heap operations on the EREW-PRAM model

<table>
<thead>
<tr>
<th>Operation</th>
<th>Parallel binary heap [8]</th>
<th>Parallel leftist heap [2]</th>
<th>Parallel binomial heap (this paper)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>$O(\log \log n + (\log n)/p)$</td>
<td>$O(\log \log n)$</td>
<td>$O(\log \log n + (\log n)/p)$</td>
</tr>
<tr>
<td>Min</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$O(\log \log n + (\log n)/p)$</td>
</tr>
<tr>
<td>Extract-Min</td>
<td>$O(\log \log n + (\log n)/p)$</td>
<td>$O(\log \log n)$</td>
<td>$O(\log \log n + (\log n)/p)$</td>
</tr>
<tr>
<td>Union</td>
<td>-</td>
<td>$O(\log \log n)$</td>
<td>$O(\log \log n + (\log n)/p)$</td>
</tr>
<tr>
<td>Decrease-Key</td>
<td>$O(\log \log n + (\log n)/p)$</td>
<td>-</td>
<td>$O(\log \log n + (\log n)/p)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$O(\log \log n + (\log n)/p)$</td>
<td>-</td>
<td>$O(\log \log n + (\log n)/p)$</td>
</tr>
</tbody>
</table>

Fig. 1. A binomial heap $H$ of 11 nodes, with the list of root and sibling pointers.
Let \( (b_{\log n}, \ldots, b_0) \) denote the binary representation of \( n \). Then the tree \( B_i \in H \) if and only if \( b_i = 1 \). For example, the binomial heap, \( H \), in Fig. 1 has 11 nodes. Since the binary representation of 11 is \( (1 \ 0 \ 1 \ 1) \), \( H \) consists of three trees: \( B_3, B_1 \) and \( B_0 \).

2.1. Sequential implementation

Let us outline how binomial heaps support various operations sequentially [3].

The operation \( \text{Union}(H_1, H_2) \) merges the lists of two roots corresponding to \( H_1 \) and \( H_2 \) into a single list \( H \), and also orders the binomial trees in \( H \) by increasing degrees. The merger list may have at most two roots of the same degree so as to satisfy the property of a binomial tree. These roots are linked until there remains at most one root of a given degree. The linking of two equal-degree binomial trees, say \( B_k \), in \( H_1 \) and \( H_2 \) results in a larger tree \( B_{k+1} \). But there may already be another \( B_{k+1} \) in the input binomial heap, so an additional link between the two trees, \( B_{k+1} \), is created. Thus a link may produce a chain of additional links, each depending on the result of the previous one.

As an example, consider Fig. 2 which also shows the keys stored at the tree roots. The roots of the degree-1 trees \( I \) and \( L \), are linked to generate a \( B_2 \) whose root is then linked with the root of \( J \), thus obtaining a new \( B_3 \). Again, the root of this \( B_3 \) is linked with that of \( E \), creating the \( B_4 \) in Fig. 3. This linking process then stops since neither \( H_1 \) or \( H_2 \) contains a \( B_4 \).

Similarly, the root of the degree-5 tree \( D \) is linked with that of \( G \), and then with the root of \( C \), yielding a new \( B_7 \) (Fig. 4). Finally, the two degree-8 trees, \( A \) and \( F \), are

![Diagram of binomial heaps](image-url)
linked together to form a new $B_9$, as depicted in Fig. 5. Note that each time a new link is added the min-heap order is preserved.

The sequential time complexity of the Union operation is $O(\log n)$ where $n = \max(n_1, n_2)$ and $n_i$ is the size of $H_i$, for $i = 1, 2$. In fact, it is possible that the Union operation combines all the binomial trees in $H_1$ and $H_2$ into a unique tree (for example, when $n_1 = 2^k - 1$ and $n_2 = 1$).

The operation Extract-Min($H$) is implemented with the help of the Union. First, the root $r$ with the minimum key in $H$ is determined by traversing the list of roots.

Fig. 3. Binomial trees of degrees 1, 2, 3 are linked to form a single binomial tree of degree 4.

Fig. 4. Binomial trees of degrees 5, 6 are linked to build a single binomial tree of degree 7.
Then $r$ is removed, and $H$ is melded with the list of children of $r$. The $\text{Insert}(H, x)$ operation is implemented by $\text{Union}(H, H')$ where $H'$ consists of the tree $B_0$ corresponding to the single node $x$ to be inserted. Both $\text{Insert}$ and $\text{Extract-Min}$ takes $O(\log n)$ time, where $n$ is the size of the largest heap.

3. Representation of parallel binomial heaps

Before proceeding further, let us represent a binomial heap, $H$, in such a way that it allows efficient execution of various operations in parallel.

A binomial heap $H$ is represented by an array of size $\log \log n + 1$. If $B_k$ is a tree of $H$, the element $H[k]$ stores a pointer to the root of $B_k$. Otherwise $H[k]$ is nil. Each node $x$ of $H$ contains the following information:

- three fields – degree, key and parent – storing, respectively, the number of its children, the key associated with it, and the pointer to its parent.
- an array $x \uparrow \cdot L$ of size $\log n + 1$, which stores the pointers to the children of $x$. If $x$ has degree $k$, its $i$th child is the root of $B_i$ and its pointer is stored in $x \uparrow \cdot L[i]$, where $0 \leq i \leq k - 1$.
- an array $x \uparrow \cdot D$ of size $\log n + 1$; its use will be postponed until Section 6.

For simplicity, we assume that arrays $x \uparrow \cdot L$ and $x \uparrow \cdot D$, for each node $x$, have the worst-case size $\log n + 1$ which is required by at most one node of $H$. Thus, the entire structure will occupy $O(n \log n)$ space. Since the actual size is equal to the degree of node $x$, which however may change as operations are performed, the best solution to save space is to use extendible arrays whose length is increased by one in $O(1)$ time.

4. Union operation in parallel

Consider melding of two binomial heaps $H_1$ and $H_2$ of sizes $n_1$ and $n_2$. Let $n_M = \max\{n_1, n_2\}$. For $i = 0, \ldots, \lfloor \log n_M \rfloor + 1$, let $a_i = 1$ denote the presence of the binomial tree $B_i \in H_1$; otherwise $a_i = 0$. Similarly the boolean $b_i$ is defined for $H_2$. The resulting heap $H = \text{Union}(H_1, H_2)$ will have $n = n_1 + n_2$ elements and the binary representation of $n$ will indicate the presence of component binomial trees in $H$. 

![Fig. 5. The heap $H = (B_9, B_7, B_4, B_0)$ resulting from the Union of $H_1$ and $H_2$ in Fig. 2.](image-url)
As pointed out earlier, $\text{Union}(H_1, H_2)$ may lead to chains of dependent links which seem inherently sequential in nature. A careful analysis of such chains shows that they occur in accordance with the carry propagation chains of the binary addition of $n_1$ and $n_2$. This is the key observation for the parallel implementation of the union operation.

Precisely, our approach for melding two heaps $H_1$ and $H_2$ consists of three phases. In the first phase, we discover in parallel the linking chains by locating the propagation chains of carries that occur while adding two binary integers $n_1$ and $n_2$ corresponding to the sizes of $H_1$ and $H_2$. In the second phase, by a parallel prefix minima computation, we decide how to connect the binomial trees in each linking chain in such a way that the min-heap order is preserved. Finally, in the third phase, we actually build the new structure by adding simultaneously all the appropriate pointers.

4.1. Phase I: Linking chains

The degrees $(i, i + 1, \ldots, j)$ of $H_1$ and $H_2$ form a linking chain for the $\text{Union}(H_1, H_2)$ operation if the $j - i + 2$ binomial trees of degree $(i, i + 1, \ldots, j)$, with precisely two binomial heaps of degrees $i$ and one heap of degrees $(i + 1, \ldots, j)$, are linked all together in a binomial tree of degree $j + 1$ after the union operation.

In other words, a linking chain occurs at positions $(i, i + 1, \ldots, j)$ if, summing $n_1$ and $n_2$, a carry propagation chain starts at bit position $i$ and ends up at bit position $j$.

In the next section, we revise how to compute efficiently a carry propagation chain, and we apply this result to our problem in Section 4.1.2.

4.1.1. Carry propagation chains in binary addition

Given two $n$-bit numbers $a = \langle a_{n-1}, a_{n-2}, \ldots, a_2, a_1, a_0 \rangle$ and $b = \langle b_{n-1}, b_{n-2}, \ldots, b_2, b_1, b_0 \rangle$, their sum $s = \langle s_n, s_{n-1}, \ldots, s_2, s_1, s_0 \rangle$ is produced by adding the bits from left to right, and propagating a carry from position $i$ to position $i + 1$, for $i = 0, 1, \ldots, n - 1$. Precisely, in the $i$th bit position, we take as inputs $a_i$ and $b_i$ and a carry bit $c_{i-1}$, and produce a sum bit $s_i = (a_i + b_i + c_{i-1}) \mod 2$ and a carry bit $c_i = \lfloor \frac{1}{2} (a_i + b_i + c_{i-1}) \rfloor$. Since there is no carry for position 0, we assume that $c_{-1} = 0$. The carry $c_{n-1}$ is the bit $s_n$ of the sum. This process is strictly sequential. However, note that two of the input values at each position are ready long before the carry at that position is ready. Hence the computation can be accelerated, as first shown in [1].

Let $\land$ and $\oplus$, respectively, denote the boolean $\text{AND}$ and $\text{XOR}$ operators. A carry is generated at position $i$ if and only if $g_i = a_i \land b_i = 1$. While, a carry is propagated at position $i$ if and only if $p_i = a_i \oplus b_i = 1$. Observe that $c_i = g_i \lor (p_i \land c_{i-1})$.

The bit positions $(i, i + 1, \ldots, j - 1, j)$ form a carry propagation chain if a carry is generated at position $i$ and the same carry is propagated up to position $j$, where $j > i$. Then, in terms of the boolean variables $g_i$ and $p_i$, the carry propagation chain $(i, i + 1, \ldots, j - 1, j)$:
- starts at position $i$ if $g_i = 1$ and if $p_{i+1} = 1$,
- terminates at position $j$ if $p_{j+1} = 0$ and for all the internal positions, $i + 1 \leq z \leq j$, it holds $g_z = 0$ and $p_z = 1$. 
Not all the bit positions belong to a carry propagation chain. In fact, there are bit positions in which the carry is generated but not propagated. Similarly, some bit positions are able to propagate but no new carries to cast. Finally, there are bit positions where nothing happens.

Now, in order to efficiently compute the carries, we introduce a new operator $\circ$ as follows: $(g,p) \circ (g',p') = (g \lor (p \land p'), p \land p')$ for boolean functions $g, g', p, p'$. Let

$$
(G_i, P_i) = \begin{cases} 
(g_1, p_1) & \text{if } i = 1, \\
(g_i, p_i) \circ (G_{i-1}, P_{i-1}) & \text{if } 2 \leq i \leq n.
\end{cases}
$$

As shown in [1], $c_i = G_i$. Then the carry computation reduces to $(G_i, P_i) = (g_1, p_1) \circ (g_2, p_2) \circ \cdots \circ (g_i, p_i)$ for $i = 1, \ldots, n - 1$. Since the operator $\circ$ is associative, the carry computation is a prefix computation and can be fully parallelized [3]. To summarize, the carry bits can be computed in any order, and in particular in parallel in a tree like order. Once all the carry bits are computed in parallel, the occurrences of the carry propagation chains (or, equivalently, the linking chains) are known.

### 4.1.2. Linking chains

In this section, we draw the similarity between the carry propagation chains and the linking chains. Let $a_i = 0$ (resp., $a_i = 1$) denote the absence (resp., presence) of the binomial tree $b_i \in H_1$. Similarly let the boolean $B_i$ be defined for $H_2$.

For melding two heaps, a position (or index) $i$ will be called a:

- **starting (str) point** for a linking chain if $g_i \land p_{i+1} = 1$,
- **internal (int) point** for a linking chain if $p_i \land c_{i-1} \land p_{i+1} = 1$,
- **ending (end) point** for a linking chain if $p_i \land c_{i-1} = 1$ and $p_{i+1} = 0$,
- **independent (ind) point** if none of the above.

The indices 0, 1, \ldots, 9 in the example of Figs. 2 and 5 are classified as shown in Table 3.

A starting point $i$ implies that $B_i$ is present in both heaps $H_1$ and $H_2$ (i.e., $g_i = 1$), and either $H_1$ or $H_2$ contains $B_{i+1}$ (i.e., $p_{i+1} = 1$). So the link between the two $B_i$’s implies a further link between the newly generated $B_{i+1}$ and the unique $B_{i+1}$ already present in either $H_1$ or $H_2$. Therefore, the linking chain starts at position $i$.

An internal point $i$ indicates that either $H_1$ or $H_2$ has a binomial tree $B_i$ (i.e, $p_i = 1$). Additionally, from the previous position $i - 1$, a link will generate $B_i$ (i.e, $c_{i-1} = 1$).

<table>
<thead>
<tr>
<th>Position</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_i \in H_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$b_i \in H_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$s_i \in H_1 \cup H_2$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$g_i$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$p_i$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$c_i$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Type</td>
<td>ind</td>
<td>ind</td>
<td>ind</td>
<td>end</td>
<td>str</td>
<td>ind</td>
<td>end</td>
<td>int</td>
<td>str</td>
<td>ind</td>
</tr>
</tbody>
</table>

Table 3

Heaps $H_1 = (B_6, B_5, B_1, B_1)$, and $H_2 = (B_8, B_5, B_2, B_1, B_0)$
Thus the linking process at position \( i \) will generate another tree \( B_{i+1} \) to be linked with the unique \( B_{i+1} \) (i.e., \( p_{i+1} = 1 \)) already contained in \( H_1 \) or \( H_2 \). Therefore, the position \( i \) is an internal point within a chain.

An **ending point** \( i \) means that either \( H_1 \) or \( H_2 \) contains \( B_i \) (i.e., \( p_i = 1 \)) and from position \( i - 1 \), a link will give rise to another \( B_i \) (i.e., \( c_{i-1} = 1 \)). In addition, the final condition \( p_{i+1} = 0 \) states that either both \( H_1 \) and \( H_2 \) contain \( B_{i+1} \), or neither of them does. If \( p_i \land c_{i-1} = 1 \), a binomial tree \( B_{prev} \) of degree \( i + 1 \) is generated by the linking process at position \( i \). However, when \( p_{i+1} = 0 \), we distinguish the following two sub-cases.

(i) If both \( H_1 \) and \( H_2 \) contain \( B_{i+1} \), the linking process skips \( B_{prev} \) and links the two \( B_{i+1} \)'s with roots in \( H_1[i+1] \) and \( H_2[i+1] \).

(ii) If neither \( H_1 \) nor \( H_2 \) has \( B_{i+1} \), the linking process moves to the position \( i + 1 \).

In both the sub-cases, the current chain stops in \( i \) since the link at position \( i + 1 \) does not depend on the tree \( B_{prev} \), and a new chain possibly starts.

The positions corresponding to the internal or ending points will not contain any binomial tree after the linking process (see Figs. 2 and 5). In fact, a linking chain starting at \( i \) and ending at \( i_e \) generates a tree \( B_{i_e+1} \). From Table 3, the end of a chain in a certain ending point does not imply the start of another chain in the next position. Between the two chains, there may exist **independent** points of two subtypes.

The first subtype are those independent points which represent **independent links** that are not dependent on the previous links. Such an independent point \( i \) satisfies \( g_i = 1 \land p_{i+1} = 0 \).

The second subtype of independent points satisfy one of the two conditions given below:

- \( p_i = 1 \land c_{i-1} = 0 \). This is true when either \( H_1 \) or \( H_2 \) contains \( B_i \) but \( c_{i-1} = 0 \). Since there is no link at position \( i - 1 \), no action takes place in \( i \).
- \( g_i = 0 \land p_i = 0 \) (i.e., \( a_i \lor b_i = 0 \)). This holds if neither \( H_1 \) nor \( H_2 \) contains \( B_i \) and there is no tree to be linked.

**4.2. Phase II: Prefix minima on linking chains**

This phase deals with the linking of binomial trees in the chains to preserve the min-heap order. Without loss of generality, let us concentrate on a particular linking chain which starts at position \( i_s \) and ends at \( i_e \), where \( 0 \leq i_s < i_e \leq \lceil \log n_M \rceil + 1 \) and \( n_M = \max\{n_1, n_2\} \). To satisfy the min-heap order, for position \( i_s \), the linking process (Phase I) generates a binomial tree \( B_{i_s+1} \) whose root, say \( n_{i_s+1} \), is the one with the minimum key between the roots of the two binomial trees, \( B_{i_s} \). This is called the linking rule.

At position \( i_s + 1 \), the root \( r_{i_s+1} \) of \( B_{i_s+1} \) belonging to either \( H_1 \) or \( H_2 \) is linked to the root \( n_{i_s+1} \). This generates a tree \( B_{i_s+2} \) having as root, say \( n_{i_s+2} \), the minimum key between \( r_{i_s+1} \) and \( n_{i_s+1} \). By induction, the binomial tree \( B_j \) at position \( j \) where \( i_s \leq j < i_e \), must be linked with the tree \( B_j \) resulting from position \( j - 1 \), whose root is the minimum key among the roots of the trees \( \{B_{i_s}, B_{i_s+1}, \ldots, B_{i-1}\} \) belonging to both \( H_1 \) and \( H_2 \). The links in the chain from \( i_s \) to \( i_e \) can be executed in parallel provided each position has information regarding the root that will result from the link of the
previous position. It does not matter how many such chains exist since they are mutually independent.

This necessary information can be computed in parallel using a segmented prefix minima operation [6] on the linking chains. An auxiliary boolean array, say \( I_{\text{lim}} \), of size \( \lceil \log n \rceil + 1 \) is used to represent the configuration of the linking chains. The array elements are assigned as \( I_{\text{lim}}[i] := \neg(p_i \land c_{i-1}) \), where \( i = 0, \ldots, \lceil \log n \rceil + 1 \). In other words, if \( i \) is a starting or independent point, \( I_{\text{lim}}[i] = 1 \); otherwise it is 0.

The segmented prefix minima is computed on an array, \( I_{\text{value}} \), of size \( \lceil \log n \rceil + 1 \) which is initialized as follows:

- if \( i \) is a starting point, then \( I_{\text{value}}[i] := H_1[i] \) provided \( H_1[i] \uparrow \cdot \text{key} \leq H_2[i] \uparrow \cdot \text{key} \); otherwise \( I_{\text{value}}[i] := H_2[i] \). Thus, according to the linking rule, \( I_{\text{value}}[i] \) is assigned to the root with the minimum key.
- if \( i \) is an internal or ending point, then \( I_{\text{value}}[i] \) is set to the unique root of the binomial tree \( B_i \) contained in either \( H_1[i] \) or \( H_2[i] \).
- the effect of the segmented prefix is null on the independent points, yet we initialize the corresponding entries of the array \( I_{\text{value}} \) as follows. If \( i \) is an independent point of first subtype, \( I_{\text{value}}[i] := H_1[i] \); otherwise \( I_{\text{value}}[i] = \text{nil} \).

The procedure \( \text{Union} \) is formally presented in Fig. 6. An example of the segmented prefix minima computation on the heaps \( H_1 \) and \( H_2 \) is shown in Table 4. The rows for \( H_1 \) and \( H_2 \) depict the key values of the roots of the corresponding binomial trees. The row \( \text{local minima} \) contains the key values of the roots after the linking rule has been applied in those positions \( i \) where \( a_i = b_i = 1 \). The row \( \text{prefix minima} \) shows the key values of the roots stored in a temporary array, \( I_{\text{value}} \), after the segmented prefix computation. Finally, for each chain, we can distinguish its fragments, each of which contains the positions with the same root in the array \( I_{\text{value}} \). The root which dominates the fragment is termed as the dominant root. In Table 4, the chain from positions 1 to 3 consists of three fragments such as \( f_{1,1} \) with key 4, \( f_{2,2} \) with key 3, and \( f_{3,3} \) with key 2.

The importance of fragments is emphasized in the following two facts.

**Fact 1.** Let \( i_s \) and \( i_e \) be the extreme positions of a fragment. After Phases I and II of the procedure \( \text{Union} \), the following holds:

1.1. All binomial trees corresponding to the points of the fragment will become children of the dominant root. Moreover, the tree \( B_i \), where \( i_s \leq i \leq i_e \), becomes the \( i \)th child of the dominant root.

1.2. The tree resulting from the parallel link in the fragment has degree \( i_e + 1 \).

**Fact 2.** Let \( S_1 \) and \( S_2 \) be two consecutive fragments in the chain (i.e., \( S_2 \) starts at the position immediately after \( S_1 \) ends). Let \( r_{S_1} \) and \( r_{S_2} \) be the respective dominant roots and \( i_{S_1} \), \( i_{S_2} \) be the position at which the fragment \( S_1 \) ends. Then:

2.1. The root \( r_{S_2} \) has a key value smaller than that of \( r_{S_1} \).

2.2. After the linking process, the tree with roots \( r_{S_1} \) becomes a child of \( r_{S_2} \).
4.3. Phase III: Parallel linking of binomial trees

Applying Facts 1 and 2, the processors can be scheduled on all links to complete the operation \( \text{Union}(H_1, H_2) \). Let us consider the processor assigned to the first position \( i \neq 0 \) which is either an internal or ending point. Two cases arise:

**Case 1.** If \( I_{\text{value}}[i] = I_{\text{value}}[i - 1] \), then \( r_i \triangleright \cdot \text{parent} := I_{\text{value}}[i] \) and \( I_{\text{value}}[i] \triangleright \cdot L[i] := r_i \), where \( r_i \) is the root of the unique tree \( B_i \) in either \( H_1[i] \) or \( H_2[i] \).
Table 4
Illustration of the segmented prefix minima

<table>
<thead>
<tr>
<th>Position</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>key(a_i) ∈ H_1</td>
<td>20</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>key(b_i) ∈ H_2</td>
<td>10</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>local minima</td>
<td>10</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>prefix minima</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fragments</td>
<td>f_5,6</td>
<td>f_5,6</td>
<td>f_3,3</td>
<td>f_2,2</td>
<td>f_1,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Case 2. If \( I_{\text{value}}[i] \neq I_{\text{value}}[i - 1] \), then \( I_{\text{value}}[i - 1] \uparrow \cdot \text{parent} := I_{\text{value}}[i] \) and \( I_{\text{value}}[i] \uparrow \cdot L[i] := I_{\text{value}}[i - 1] \).

If \( i \) is a starting or independent point, a third case also arises which includes \( i = 0 \).

Case 3. If \( g_i = 1 \), the binomial trees with roots \( H_1[i] \) and \( H_2[i] \) are linked to generate \( B_{i+1} \) whose root is the one with the smaller key value.

Case 1 refers to the positions belonging to a fragment specified by the same dominant root \( I_{\text{value}}[i] \); Case 2 deals with the dominant roots of two consecutive fragments; while Case 3 refers to the independent points whose possible links can be built directly by the corresponding processors.

We now describe how the merger heap, \( H \), is generated as a result of the Union operation. For each \( i = 0, \ldots, \lfloor \log n \rfloor \), the entry \( H[i] \) was initialized to nil and is now reassigned as follows:
1. if \( g_i = 1 \land p_{i+1} = 0 \), then \( H[i + 1] := I_{\text{value}}[i] \) and \( I_{\text{value}}[i] \uparrow \cdot \text{degree} := i + 1 \),
2. for independent points, \( H[i] := I_{\text{value}}[i] \). The unique binomial tree in \( H_1 \) or \( H_2 \) is simply copied in \( H \).
3. if \( i \) is an ending point, then \( H[i + 1] := I_{\text{value}}[i] \) and \( I_{\text{value}}[i] \uparrow \cdot \text{degree} := i + 1 \).

The procedure Union in Fig. 6 uses two sub-procedures. The first one, Choose-NotNull (\( H_1[i], H_2[i] \)), returns \( H_1[i] \) if it is not nil, otherwise it returns \( H_2[i] \). If both of them are null, it returns nil. The other sub-procedure, Combine (\( H_1[i], H_2[i] \)), implements the linking rule as given below.

procedure Combine\( (r_1, r_2) \)
\[ i := r_1 \uparrow \cdot \text{degree}; \]
\[ \text{if} \ r_2[i] \uparrow \cdot \text{key} < r_1[i] \uparrow \cdot \text{key} \text{ then swap } r_1 \text{ with } r_2; \]
\[ r_2 \uparrow \cdot \text{parent} := r_1; /r_2 \text{ becomes child of } r_1; /r \]
\[ r_1 \cdot L[i] := r_2. \]

4.4. Complexity analysis

Theorem 1. On the EREW–PRAM model, the operation \( \text{Union}(H_1, H_2) \) requires \( O(\log \log n + (\log n)/p) \) time and \( O(\log n) \) work, which is optimal employing \( p = O(\log n/\log \log n) \) processors.

Proof. Employing \( p \) processors on the EREW model, the values of booleans \( g_i, p_i \) and \( c_i \) (Steps 1–3 of procedure Union) can be computed in \( O(\log \log n + (\log n)/p) \) time, requiring \( O(p \log \log n + \log n) \) work [6]. Loops corresponding to Steps 4–10 and Steps 19–28 require \( O((\log n)/p) \) time and \( O(\log n + p) \) work.
Let us examine next the memory accesses for the loops. The procedure \textit{Combine} called at Step 13 accesses different memory locations for different values of \( i \). While invoking the procedure \textit{Combine} \( (I_{\text{value}}[i], r_i) \) at Step 16, all processors working with the elements in the same fragment go to the same dominant root, \( r \), to change information about the children of \( r \) in the array \( L \), but each of these processors accesses a different location of \( L \). In Step 17 of the procedure \textit{Combine}, only one processor attaches the root with the larger key to the one with the smaller key. All the processors working with two roots update the array \( L \) by accessing different positions, and thus there is no conflict in reading or writing on the shared memory. Hence the theorem. □

\textbf{Corollary 1.} On the EREW–PRAM model, the operations \textit{Min} and \textit{Extract-Min} require \( O(\log \log n) \) time and \( O(\log n) \) work.

\textbf{Proof.} Straightforward, since operations like \textit{Min} and \textit{Extract-Min} are based on the \textit{Union}. □

5. Insert operation in parallel

The operation \textit{Insert}(\( H, x \)) which inserts a new node \( x \) into a heap \( H \), uses the procedure \textit{Union} and thus requires \( O(\log \log n) \) time and \( O(\log n) \) work. We shall see that even \( \log n \) items can be inserted concurrently within this time bound. The multiple insertion can be easily extended to deal with an arbitrary number, \( l \), of new items. Since inserting \( l \) items into an empty queue \( H \) is equivalent to building a new queue, our approach also yields a work-optimal EREW–PRAM algorithm for constructing a binomial heap.

5.1. Inserting multiple elements: \textit{Multiple-insert}(\( H, (x_1, \ldots, x_l) \))

Let \( x_1, \ldots, x_l \) be the new nodes to be inserted in the heap \( H \), where \( l = 2^d \). A binomial heap \( H' = B_d \) is first built and then the operation \textit{Union}(\( H, H' \)) is applied.

The algorithm to construct \( B_d \) in parallel is based on the known technique of building a computational binary tree, in which the leaves store the binomial tree \( B_0 \) corresponding to the items \( x_i \) for \( 1 \leq i \leq l \). An internal node of the binary tree represents the linking operation between its two children. The first iteration divides \( x_1, \ldots, x_l \) into \( l/2 \) pairs: \( (x_1, x_2), (x_3, x_4), \ldots, (x_{l-1}, x_l) \). For each pair, the procedure \textit{Combine} is invoked to generate a binomial tree \( B_1 \), thus producing \( l/2 \) binomial trees. After the \( n \)th iteration, there will be \( l/2^n \) binomial trees, \( B_j \). This iteration runs in \( O(1 + 1/(p2^n)) \) parallel time on a \( p \)-processor EREW model. Hence, after \( d \) iterations we obtain the unique binomial tree \( B_d \) containing all \( l \) elements. The total time required is

\[
T_p = \sum_{i=0}^{d} O \left( 1 + \frac{2^d}{p2^i} \right) = O \left( d + \frac{2d}{p} \right).
\]
For \( p = O(2^d/d) \), \( T_p = O(d) \) and the optimal work is \( O(2^d) \). Melding \( H \) with \( B_d \), yields the following.

**Theorem 2.** On the EREW–PRAM model, the operation \( \text{Multiple-Insert}(H, (x_1, \ldots, x_{2^d})) \) requires \( O(\log \log n + (\log n)/p + d + (2^d/p)) \) time and \( O(2^d + \log n) \) work which is optimal employing \( p = \min \{O(\log n/\log \log n), O(2^d/d)\} \) processors.

### 6. Delete and decrease-key operations in parallel

This section implements \textit{Delete} and \textit{Decrease-Key} operations and presents their amortized analyses. The term “amortized” means that the cost of an operation is averaged over the worst-case costs of a sequence of operations. We demonstrate how to make use of the local array \( x \uparrow \cdot D \) associated with a node \( x \), introduced in Section 3. The underlying approach for implementing \textit{Delete} and \textit{Decrease-Key} operations is quite different from their sequential counterparts.

#### 6.1. Delete

Depending on the node to be removed, each \textit{Delete} operation has an immediate effect only to a restricted set of nodes in the heap. Thus an update of the entire data structure can be delayed for a future instant. Operations having a local effect on the heap are executed first by the \textit{Take-Up} procedure described below, while a global update is performed after a certain number of deletions. A counter, \( \text{deleted} \), is increased by 1 after each \textit{Delete} operation. After \( n/2 \) deletions, the whole data structure is rebuilt from scratch discarding all the deleted nodes persistent in the structure. The procedure \textit{Delete}(\( H, x \)) is outlined below assuming that \( x \) is not a root.

**procedure Delete**(\( H, x \))

\[
\text{\begin{verbatim}
/* x is not a root of one of the binomial trees in H, otherwise procedure Extract-Min is invoked */
\text{deleted := deleted + 1;}
\text{Take-Up(x);}
\text{if deleted = n/2 then build from scratch.}
\end{verbatim}
\]

#### 6.2. Take-up procedure

Let us introduce some additional terminology. From now on, a deleted node persistent in the data structure will be referred to as an \textit{empty node}, and marked by the key \(-\infty\). A binomial subtree which contains only empty nodes is termed \textit{empty binomial subtree}. A node is said to be \textit{live} if it is not empty. A binomial subtree is called \textit{non-empty} if its root is live, although some of its nodes may be empty. As will be discussed, all the binomial subtrees in our structure will be forced to be \textit{empty} or \textit{non-empty}.

The presence of the empty nodes affects the representation of the binomial heap as follows. The array \( x \uparrow \cdot D \) (resp., \( x \uparrow \cdot L \)) stores the pointers to the empty (resp., live)
roots of the binomial trees, which are children of \( x \). Precisely, the root of the \( i \)th child of \( x \), where \( 0 \leq i < x \uparrow \cdot \text{degree} \), is stored in \( x \uparrow \cdot D[i] \) if the child \( B_i \) is an empty binomial tree. Otherwise, it is stored in \( x \uparrow \cdot L[i] \).

The procedure \textit{Take-Up} shown in Fig. 7 satisfies the following invariant.

**Invariant 1.** After executing the procedure \textit{Take-Up}, any node \( y \) of the heap satisfies one of the following conditions:

1.1. \( y \) is live, i.e., some, but not all, of its children may be empty binomial trees. That is, for each position \( i \), either \( y \uparrow \cdot D[i] \neq \text{nil} \) or \( y \uparrow \cdot L[i] \neq \text{nil} \), where \( 0 \leq i \leq y \uparrow \cdot \text{degree} \).

1.2. \( y \) is empty, i.e., all nodes in the binomial subtree rooted at \( y \) are empty nodes. In other words, \( y \uparrow \cdot D[i] = \text{nil} \) and \( y \uparrow \cdot L[i] = \text{nil} \) where \( 0 \leq i \leq y \uparrow \cdot \text{degree} \).

The scope of the procedure \textit{Take-Up} is to record the children of \( x \) and its parent, \( p(x) \), so that Invariant 1 is satisfied and also the data structure is preserved to continue with the \textit{Min}, \textit{Extract-Min}, \textit{Union}, \textit{Insert}, and \textit{Decrease-Key} operations in the presence of persistent deleted nodes. For example, if no local arrangement is made, \textit{Extract-Min} could find all the roots of the binomial heap as empty, thus being unable to return the minimum key in the claimed time bounds. In contrast, after the \textit{Take-Up} operation, if all the roots of the binomial heap are empty, Invariant 1.2 ensures that the binomial heap itself is void. As soon as a binomial tree in the heap is

```plaintext
procedure \textit{Take-Up}(x)
01 \( x \uparrow \cdot \text{key} := -\infty \); /* node \( x \) is marked empty */
02 \( k_x := x \uparrow \cdot \text{degree}; \)
03 \( k_{p(x)} := x \uparrow \cdot \text{parent}; \)
04 \( p(x) := p(x) \uparrow \cdot \text{degree}; \)
05 \( k_{p(x)} \uparrow \cdot L[k_x] := \text{nil}; /* B_{k_x} is no longer in the array \( L \) */ \)
06 \( p(x) \uparrow \cdot L := \text{Union}(p(x) \uparrow \cdot L, x \uparrow \cdot L); /* to satisfy Invariant 1.1. */ \)
07 \textbf{for} \( i = 0, \ldots, k_{p(x)} \) \textbf{pardo}
08 \textbf{if} \( p(x) \uparrow \cdot L[i] \neq \text{nil} \) \textbf{then} \( p(x) \uparrow \cdot L[i] \uparrow \cdot \text{parent} := p(x); \)
09 \textit{Insert}(p(x) \uparrow \cdot D, x); /* to satisfy Invariant 1.2 */
10 \( p(x) \uparrow \cdot D := \text{Union}(p(x) \uparrow \cdot D, x \uparrow \cdot D); /* to satisfy Invariant 1.2 */ \)
11 \textbf{for} \( i = 0, \ldots, k_{p(x)} \) \textbf{pardo}
12 \textbf{if} \( D_{p(x)}[i] \neq \text{nil} \) \textbf{then} \( p(x) \uparrow \cdot D[i] \uparrow \cdot \text{parent} := p(x); \)
```

Fig. 7. The operation \textit{Take-Up}, assuming that the node \( x \) is not a root.
completely empty, it can be checked by the Take-Up or the Union procedure and then discarded (see Table 5).

Note that Invariant 1 is satisfied after the execution of the Union operation as well as other operations based on it. In fact, \( \text{Union}(H, H') \) will simply link the lists of non-empty binomial trees in \( H \) and \( H' \). Since the internal nodes (possibly empty) are untouched and the roots of the resulting binomial trees are live nodes, Invariant 1 will be satisfied.

Therefore, the procedure Take-Up requires \( O(\log \log n) \) time and \( O(\log n) \) work on the EREW–PRAM model since the maximum degree of a node is \( \lfloor \log n \rfloor + 1 \). Let us now derive the amortized complexity of the Delete operation, considering that the queue is rebuilt from scratch when half of its items are empty (see Fig. 8).

**Theorem 3.** On the EREW–PRAM model, the Take-Up procedure requires in the worst-case \( O(\log \log n + (\log n)/p) \) time and \( O(\log n) \) work. Each Delete operation has an amortized parallel time of \( O(\log \log n) \) and \( O(n) \) work.

**Proof.** The procedure Take-Up, invoked at each deletion, performs at most three Union operations and therefore, it requires in the worst-case \( O(\log \log n) \) time and \( O(\log n) \) work. After \( n/2 \) deletions, a new queue is built from the remaining \( n/2 \) live nodes of the queue in \( O(n/p) \) time for an overall \( O(n) \) work. Therefore, each Delete requires \( O(1) \) amortized time for such reconstruction from scratch, and the procedure Take-Up dominates the complexity of each single delete operation. The empty

---

**Table 5**
The status of a tree \( B_4 \) before and after procedure Take-Up(x)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>( p(x) )</td>
<td>( z )</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>( y )</td>
<td>-</td>
</tr>
<tr>
<td>Before</td>
<td>( s )</td>
<td>( x )</td>
<td>nil</td>
<td>-</td>
<td>nil</td>
<td>( w )</td>
<td>-</td>
</tr>
<tr>
<td>After</td>
<td>( p(x) )</td>
<td>( z )</td>
<td>( x )</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>( y )</td>
</tr>
<tr>
<td>After</td>
<td>( s )</td>
<td>nil</td>
<td>nil</td>
<td>-</td>
<td>( t )</td>
<td>( w )</td>
<td>-</td>
</tr>
</tbody>
</table>

---

Fig. 8. Persistent empty nodes (shown in black): (a) before Take-Up(x) and (b) after Take-Up(x).
nodes persistent in the structure can only affect the constants (not the order of the magnitude) involved in the time complexity of each operation.

6.3. Decrease-key

The operation $\text{Decrease-Key}(H, x, k)$ is implemented using the $\text{Delete}$ and $\text{Insert}$ as sub-procedures. After deleting the node $x$, a new node is inserted in the heap and filled in with the new key $k$. Since the new key $k$ may be even larger than the old value, a more suitable name for $\text{Decrease-Key}$ is $\text{Change-Key}$. Its performance is the same as the $\text{Delete}$ operation.

7. Conclusions

We have presented an optimal parallel implementation of meldable priority queues based on the binomial heap data structures. Algorithms for the standard heap-operations like $\text{Insert}$, $\text{Delete}$, $\text{Min}$, $\text{Extract-min}$, and $\text{Union}$ require doubly logarithmic parallel time and are work-optimal on the EREW–PRAM model. Our parallel $\text{Union}$ algorithm looks at the shape of the final binomial queue and exploits its natural similarity with the binary integer addition. We have also designed an optimal algorithm for insertion of $l$ items simultaneously.

Two less commonly known but important operations, $\text{Delete}$ and $\text{Decrease-Key}$, are also implemented on the EREW–PRAM model. They work by postponing the global effect of a node deletion and using some tricks to keep the (persistent) deleted nodes organized in the tree.

Acknowledgements

The authors are grateful to the anonymous referees and the Editor, Robert Hiromoto, for helpful comments which improved the quality of the paper.

References