Parallel and Distributed Meldable Priority Queues Based on Binomial Heaps *

Vincenzo A. Crupi†, Sajal K. Das‡, and M. Cristina Pinotti†

†Istituto di Elaborazione dell'Informazione
Consiglio Nazionale delle Ricerche
Via S. Maria, 46
56126 Pisa, ITALY
E-mail: {crupi,pinotti}@iei.pi.cnr.it

‡Department of Computer Sciences
University of North Texas
P.O. Box 13886
Denton, TX 76203-6886, USA
E-mail: das@cs.unt.edu

Abstract

On the parallel random access machine (PRAM) models, we present an optimal implementation of a meldable *priority queue based on the binomial heap data structure. Doubly logarithmic time, cost-optimal algorithms are proposed on the EREW PRAM model for such queue-operations as Insert, Min, Extract-Min, and Union (often called* melding*). Our Union algorithm exploits its natural similarity with the problem of adding two binary integers. Two other operations,* Change-Key *and* Delete*, are also solved optimally on the CREW PRAM model and their amortized complexity is analyzed. The algorithm for Delete operation uses tricks for postponing the global effect of a node deletion. Additionally, we present an implementation of a* distributed *meldable priority queue on the single-port hypercube model. Adopting a b-bandwidth binomial heap as the underlying data structure and a mapping which minimizes the amount of data movement among the processors when two binomial trees are melded, the hypercube algorithm attains a nearly optimal amortized speed-up for the Insert, Extract-Min and Union operations.*

1 Introduction

Priority queues are very useful data structures for solving numerous problems which are based on the principle of selecting the smallest (or largest) element from a totally ordered set.

Definition 1 : A *priority queue*, Q, supports the following four standard operations.

1. *Make-Queue(Q): create and return a new empty queue.*
2. *Insert(Q, x): insert a node x into the queue Q.*
3. *Min(Q): return a pointer to the node with the minimum key in Q.*
4. *Extract-Min(Q): delete from Q the node with the minimum key and return a pointer to it.*

A queue is called *meldable* if it also supports the union of two queues, defined as

5. *Union(Q_1, Q_2): create and return a new queue after melding two queues Q_1 and Q_2, which are then destroyed.*

A queue can also provide two other operations.

6. *Change-Key(Q, x, k): change the current key value of a node x in the queue Q to the new key value k.*
7. *Delete(Q, x): delete node x from Q.*

There exist various data structures to implement priority queues in the sequential computing environment, namely binary heaps, leftist[1] heaps and binomial heaps [2]. Both leftist and binomial heaps are meldable priority queues and they exhibit almost the same worst-case time performance, although they have different topological properties. Also several parallel algorithms [1, 5, 8, 9, 10] have been designed for implementing priority queues on shared memory machines.

Let us recall that the *cost* [6] of a parallel algorithm is defined as the product of *maximum parallel time* and *maximum number of processors* used. The *work* [6] of a parallel algorithm is the sum of the products *parallel time* × *number of processors* for each phase of the algorithm. Clearly, the cost is an upper bound of the actual *work* performed. A parallel algorithm is *cost* (or *work*)*-optimal*, if its cost (or work) is proportional to the time complexity of the best known sequential algorithm for that problem.

Both the implementations of b-bandwidth priority queues based on binary heaps [9] and leftist heaps [10], are cost- and work-optimal on the exclusive-read and exclusive-write (EREW) PRAM model. Recently, Chen and Hu [1] improved the results in [9] achieving a better speed-up for the management of meldable priority queues, yet preserving the work-optimality on the concurrent-read and exclusive-write (CREW) PRAM

*This work is partially supported by Texas Advanced Technology Program grant TATP-003594031 and by Progetto 40% "Efficienza di Algoritmi e Progetto di Strutture Informative."

[1] A fully binary tree T is a *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.

model. Namely, the implementation in [1] supports Insert, Extract-min, Make-Queue and Union operations in $O(\log\log n)$ time, performing $O(\log n)$ work for a heap of n items.

In this paper, we deal with the parallel management of meldable priority queues based on the binomial heaps. The operations numbered 1-5 in Definition 1 are implemented on the EREW PRAM model, whereas the operations 6-7 require the CREW PRAM model. All of our algorithms require $O(\log\log n)$ time employing $p = O(\frac{\log n}{\log\log n})$ processors. We do not claim that the proposed parallel algorithms for meldable priority queues based on the *binomial heaps* offer better performance than that achieved with the help of leftist heaps [1]. Nevertheless, we found them interesting because the parallel implementations of both the Union and Delete operations follow a totally different strategy compared to their sequential counterparts.

Finally, we study on the hypercube (assuming the single-port communication model) an implementation of meldable priority queues supporting the standard operations 1-5 in Definition 1. To the best of our knowledge, this is the first implementation of a meldable priority queue on a synchronized distributed system. A load-balanced hypercube algorithm for unmeldable priority queues based on ordinary binary heaps has been recently proposed by Das, et al. [4].

The paper is organized as follows. Section 2 introduces the binomial heap data structure, and Section 3 proposes parallel implementations of the queue operations 1-5. In Section 4, the two relatively less addressed operations, Delete and Change-Key, are discussed. Section 5 devises a mapping of meldable priority queues on a single-port hypercube architecture. Section 6 concludes the paper.

2 Preliminaries on Binomial Heaps

Definition 2 *[2]: The binomial tree, B_k, of order k and consisting of 2^k nodes is a tree defined recursively as follows. (i) The binomial tree B_0 consists of a single node, (ii) for $k > 0$, B_k consists of two binomial trees B_{k-1} whose roots are linked together such that the root of one becomes the leftmost child of the other root.*

The root of B_k has degree k (i.e., the number of children), which is the maximum degree of any tree node. If the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, the i-th child is the root of a sub-binomial tree B_i. Thus there are exactly $\binom{k}{j}$ nodes at depth j of B_k.

Definition 3 *: A* binomial heap, *H, of size n is a collection of binomial trees such that*
(BH1) *each binomial tree in H maintains the min-heap ordering, i.e. the key of a node is no smaller than that of its parent.*
(BH2) *there is at most one binomial tree in H whose root has degree i, where $0 \leq i \leq \lfloor \log n \rfloor$. In other words, there is at most one binomial tree in H with 2^i nodes.*

Let $n = \sum_{i=0}^{\lfloor \log n \rfloor} b_i 2^i$ and let $< b_{\lfloor \log n \rfloor}, \ldots, b_0 >$ denote the binary representation of n. Then, from Property (BH2), the binomial tree B_i is present in the heap if $b_i = 1$. For example, a binomial heap of 11 nodes consists of the trees B_3, B_1 and B_0 since the binary representation of 11 is $< 1011 >$.

In order to represent a binomial tree, the following informations are stored in the shared memory for each node x.

- the fields *degree*, *key* and *parent* storing respectively the number of children of x, the key associated with it, and the pointer to its parent.
- an array L_x of size $\lfloor \log n \rfloor + 1$, which stores the pointers to the children of x. If x has degree k, $L_x[k-1], L_x[k-2], \ldots, L_x[0]$ points to the roots (i.e, children of x) of the binomial trees $B_{k-1}, \ldots, B_0$. It is assumed that the remaining position of L_x are set to nil.

Moreover, each binomial heap is represented by an array H of size $\lfloor \log n \rfloor + 1$ storing the pointers to the roots of the binomial trees in the heap.

3 Parallel Binomial Heaps

Due to the space limitation, we only present the parallel algorithm for the Union operation on which the Insert and Extract-Min operations are based. The implementation of the Min and Make-Queue operations is trivial.

The Union procedure receives as arguments two arrays (or pointers to them) H_1 and H_2 representing the two binomial heaps to be merged and returns the new binomial heap H. Let $n_M = \max\{n_1, n_2\}$, where n_1 and n_2 are respectively the number of elements in the two heaps represented by H_1 and H_2. Let a_i (resp., b_i) denote the boolean value indicating the presence of the binomial tree B_i in H_1 (resp., H_2), where $i = 0, \ldots, \lfloor \log n_M \rfloor + 1$. Precisely, $a_i = 1$ if and only if B_i is in H_1, otherwise $a_i = 0$. The resulting heap H will have $n = n_1 + n_2$ elements and from the binary representation of n we can analogously know which binomial trees will be present in H.

The Union is accomplished in three phases: (i) defining the linking chains, (ii) performing prefix minima in linking chains, and (iii) parallel linking the binomial trees.

3.1 Phase I: Linking Chains

During the Union operation, binomial trees of consecutive orders present in either H_1 or H_2 may be melded altogether in a larger tree. The orders of the binomial trees involved in such fusion form a *linking chain*. The linking chains are found out exploiting their similarities with *carry chains* of the binary addition operation.

Let us first borrow some terminologies and definitions from the binary addition process. For $i \in [0, \ldots, \lfloor \log n_M \rfloor + 1]$

- let $g_i = [(a_i + b_i) \operatorname{div} 2]$ denote the *carry generator* and $p_i = a_i \oplus b_i$ the *carry propagator*.
- let $c_i = \lfloor \frac{(a_i + b_i + c_{i-1})}{2} \rfloor$, or $c_i = g_i \vee (p_i \wedge c_{i-1})$ be the *binary sum carry* (assume $c_{-1} = 0$).
- let $s_i = [(a_i + b_i + c_{i-1}) \mod 2]$ denote the i-th digit of the *binary sum*.

These boolean variables are interpreted and used as follows in the binomial heap context. First, if $s_i = 1$, the tree B_i is present in H. After computing the variables g_i, c_i and p_i on the boolean representations of H_1 and H_2, we classify a position i as:

- *starting point* of a linking chain if $(g_i = 1) \wedge (p_{i+1} = 1)$. That is, the link between the two B_i's implies a further link with the unique B_{i+1} already contained in position $i+1$.
- *internal point* of a linking chain if $(p_i = 1) \wedge (c_{i-1} = 1) \wedge (p_{i+1} = 1)$. That is, the new binomial tree B_{i+1} from position i will be linked with the tree of the same order already in H_1 or H_2.
- *ending point* of a linking chain if $(p_i = 1) \wedge (c_{i-1} = 1) \wedge (p_{i+1} = 0)$. That is, the tree at position $i+1$ does not depend on the tree in position $i-1$. Note that a new chain possibly starts.

All the positions of the boolean representations of H_1 and H_2 which do not fall in the above three cases are called *independent points*. Figure 1 shows an example. The terms *str*, *end*, *int* and *ind* stand respectively for *starting*, *ending*, *internal* and *independent* points.

Position	7	6	5	4	3	2	1	0
a_i	0	1	1	0	1	0	1	0
b_i	0	0	1	0	0	1	1	1
g_i	0	0	1	0	0	0	1	0
p_i	0	1	0	0	1	1	0	1
c_i	0	1	1	0	1	1	1	0
s_i	1	0	0	1	0	0	0	1
Type	ind	end	str	ind	end	int	str	ind

Figure 1: Heap H_1 contains binomial trees B_1, B_3, B_5, B_6 and H_2 contains B_0, B_1, B_2, B_5. The merger heap H contains B_i if $s_i = 1$, for $0 \leq i \leq 7$.

We point out that a unique binomial tree B_{i_e+1} is generated from a linking chain starting at i_s and ending at i_e, where $i_e > i_s$, adding $i_e - i_s + 1$ links to the structure during the Union operation. The positions corresponding to the internal or ending points will not contain any binomial trees after the linking process. From Figure 1, we can see that the end of a chain in a certain ending point does not imply the start of another chain in the next position. Actually, between two chains there can be some independent points, which can further be divided into two types. The first type contains all independent points representing *independent links* which do not depend on the result of previous links. Whereas, the second type includes all independent points to which no action is associated in the sense that no links are made corresponding to them. More formally, an independent point i belongs to the first type if $g_i = 1$ and since it is not a starting point, $p_{i+1} = 0$. It belongs to the second type if one of the following two conditions holds:

- $(p_i = 1) \wedge (c_{i-1} = 0)$. This is verified when either H_1 or H_2 contains a binomial tree B_i but $c_{i-1} = 0$. Since there is no link at position $i-1$, no action takes place in i.
- $(g_i = 0) \wedge (p_i = 0)$, (i.e, $a_i \vee b_i = 0$). This holds if neither H_1 nor H_2 contains a binomial tree B_i. So there is no tree to be linked even if there is a carry in the previous position.

3.2 Phase II: Prefix Minima in Linking Chains

In the second phase of the Union procedure, we determine how the binomial trees in every chain should be linked together so as to guarantee the heap-property. Let us concentrate on a particular linking chain which starts at position i_s and ends at i_e, where $\lfloor \log n_M \rfloor + 1 \geq i_e > i_s \geq 0$ and $n_M = \max\{n_1, n_2\}$. To satisfy the heap-order property, we know that the linking process generates from i_s 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 known in literature as the *linking rule*. By induction, at position j where $i_e \geq j > i_s$, the root r_j of the binomial tree B_j must be linked with the root n_j of the tree resulting from the previous position $j-1$, which is a binomial tree of order j whose root is the minimum key chosen from the roots of binomial trees present in both H_1 and H_2 of order $\{i_s, i_s+1, \ldots, j-1\}$. A new binomial tree B_{j+1} having as root the minimum key between r_j and n_j is created. Then, the links in the chain from i_s to i_e can be added in parallel provided that each position has the information corresponding to the root that will result from the link of the previous position. This necessary information is computed by a segmented prefix minimum operation [6] on each linking chain. For this aim, an auxiliary array, say I_{lim}, of booleans with size $\lfloor \log n \rfloor + 1$ is needed to represent the configuration of the linking chains. The array elements are assigned as $I_{\text{lim}}[i] := \neg(p_i = 1 \wedge c_{i-1} = 1), \forall i = 0, \ldots, \lfloor \log n \rfloor + 1$. That is:

- if i is a starting or independent point, then $I_{\text{lim}}[i]$ is set to 1.
- if i is an internal or ending point, then $I_{\text{lim}}[i]$ is set to 0.

The segmented prefix minima, guided by I_{lim}, is executed on an array I_{value} of size $\lfloor \log n \rfloor + 1$ storing for each position i the pointer to the root containing the minimum key between the two roots pointed by $H_1[i]$ and $H_2[i]$. Obviously, $I_{\text{value}}[i] = nil$ if both $H_1[i]$ and $H_2[i]$ are *nil*.

Figure 2 shows an example where the segmented prefix minima are applied to the heaps H_1 and H_2. The rows for H_1 and H_2 report the key values of the

Position	13	12	11	10	9	8	7	6	5	4	3	2	1	0
H_1	6	-	4	8	7	6	12	-	2	-	-	10	3	5
H_2	-	3	-	-	5	13	-	9	-	7	5	-	4	-
Type	end	int	int	int	str	ind	end	int	int	int	int	int	str	ind
I_{lim}	1	0	0	0	1	1	1	0	0	0	0	0	1	1
I_{valueB}	6	3	4	8	5	6	12	9	2	7	5	10	3	5
I_{valueA}	3	3	4	5	5	6	2	2	2	3	3	3	3	5

Figure 2: The segmented prefix minima.

corresponding binomial tree roots. The letters B and A associated with the array I_{value} represent respectively the contents of I_{value} *before* and *after* the application of the prefix operation. For simplicity, we have assigned directly to I_{value} the keys of roots rather than the pointers to them. For illustration, let us examine the linking chain which starts at position 1 and ends at position 7. We can distinguish various fragments each of which contains the positions with the same root in the array I_{value} after the prefix-minima computation. We say that the root which dominates the fragment, i.e., the root with the minimum key in the fragment, is a *dominant* root. The chain under consideration consists of two fragments – the first one with key 3 and the second one with key 2. The importance of fragments to complete the Union operation is emphasized in the following two facts.

Fact 1 *: Let i_s and i_e be the extreme positions of a certain fragment. After Phases I and II of the Union procedure, the following holds:*

1.1 all binomial trees corresponding to the points of the fragment will become children of the dominant root. Moreover, the binomial tree B_i, where $i_e \geq i \geq i_s$, becomes the i-th child of the dominant root.

1.2 the resulting tree after the parallel link in the fragment will have order $i_e + 1$.

The next fact involves two consecutive fragments in a chain:

Fact 2 *: Let S_1 and S_2 be two consecutive fragments, i.e. S_2 starts at the position immediately next to which S_1 ends. Moreover, let r_{S_1} and r_{S_2} be the respective dominant roots and $i_{e_{S_1}}$ be the position in which 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 binomial tree with root r_{S_1} becomes a child of r_{S_2}.

3.3 Phase III: Parallel Linking of Binomial Trees

In this phase, with the help of Facts 1 and 2, the processors can actually execute all links to complete the Union operation of two heaps H_1 and H_2. Let us concentrate on the processor which works in position $i \neq 0$, which is first assumed to be an internal or an ending point. Let t stand for $I_{\text{value}}[i]$, for simplicity of notation. Here two cases arise:

Case 1: If $t = I_{\text{value}}[i-1]$ then we set $r_i \uparrow .parent := t$ and $L_t[i] := r_i$, where r_i is the root of the unique tree B_i contained in either $H_1[i]$ or $H_2[i]$.

Case 2: If $t \neq I_{\text{value}}[i-1]$ then $I_{\text{value}}[i-1] \uparrow .parent := t$ and $L_t[i] := I_{\text{value}}[i-1]$.

If i is a starting or independent point, the following third case also holds which includes $i = 0$.

Case 3: If $g_i = 1$ then apply the linking rule to the roots $H_1[i]$ and $H_2[i]$.

We now describe how the new heap H is generated as a result of the Union operation. For each $i = 0, \ldots, \lfloor \log n \rfloor$, the value for $H[i]$ initially set to nil, is reassigned as follows:

1. if $(g_i = 1) \wedge (p_{i+1} = 0)$, then $H[i+1] := t$ and $t \uparrow .degree := i + 1$, recalling $t = I_{\text{value}}[i]$. That is, the binomial tree resulting from the link at position i (independent point with $g_i = 1$) has order $i + 1$ which is updated, so the new heap must contain a pointer to the root at the position $i + 1$.
2. if $(i \geq 0) \wedge (p_i = 1) \wedge (c_{i-1} = 0)$, then $H[i] := t$. This deals with all independent points. The unique binomial tree of either H_1 or H_2 is simply copied in H.
3. if i is an ending point, then $H[i+1] := t$ and $t \uparrow .degree := i + 1$. A binomial tree B_{i+1} arises from the ending point i with the root contained in t, so we update its degree and make $H[i+1]$ point to it.

Let us analyze the total running time of the procedure Union on the EREW PRAM model consisting of p processors. In Phase *Linking chains*, the carries c_i can be determined on the EREW PRAM model in time $O(\log\log n + \frac{\log n}{p})$ requiring $O(p \log\log n + \log n)$ work [2, 6], while the other information such as g_i, p_i and the point classification, take constant parallel time. It is easy to see that the second phase is performed in $O(\frac{\log n}{p})$ parallel time with $O(\log n + p)$ work, of which $O(\log\log n + \frac{\log n}{p})$ time [6] and $O(p \log\log n + \log n)$ work is spent in the segmented prefix computation. The third phase takes $O(\frac{\log n}{p})$ time and $O(\log n + p)$ work. Therefore,

Theorem 1 *: On the EREW PRAM model, the Union, Insert and Extract-Min operation require $O(\log\log n + \frac{\log n}{p})$ time and $O(\log n)$ optimal work employing $p = O(\log n / \log\log n)$ processors.*

4 Delete and Change-Key Operations

In this section, we describe the *Delete* operation and present its amortized analysis. The term *amortized* means that the cost of an operation is averaged over the worst-case cost of a sequence of operations. The parallel approach for implementing the Delete operation is quite different from the sequential counterpart, even though it is not new to the data structure management. Actually, each Delete has an immediate effect only on a restricted set of nodes in the structure depending on the node involved in the removal, thus delaying the update of the whole data structure after a certain number of deletions. A deleted node persistent in the data structure is referred to as an *empty* node, and it is marked by the key $-\infty$. To count the number of Delete operations carried out, we use a counter referred to as *deleted*, while the array *Del* of size $\lfloor \frac{\log n}{\log\log n} \rfloor$ stores the pointers to the empty nodes.

From now on, we assume that the node x to be deleted is not a root, since in this case the operation

Delete can be executed in a manner similar to Extract-Min.

Due to the presence of the empty nodes, let us introduce the following nomenclature. An *empty* sub-binomial tree contains only empty nodes. Moreover, a node is said to be *live* if it is not empty. A sub-binomial tree is *non-empty* if its root is live, although some of its nodes may be empty. Moreover, the representation of the binomial heap is augmented as follows. In addition of the fields degree, key and parent, each node x stores two arrays L_x and D_x of size $\lfloor \log n \rfloor + 1$. Each child of a node x of the binomial heap is stored in either the array D_x or L_x. Let D_x (resp., L_x) be the array storing the pointers to the roots of the empty (resp., not-empty) binomial trees, which are children of x. For example, in Figure 3(a), $D_{p(x)}[0] := z$, $D_{p(x)}[1] := nil$ and $D_{p(x)}[2] := nil$ whereas $L_{p(x)}[0] := nil$, $L_{p(x)}[1] := y$, and $L_{p(x)}[2] := x$.

The operation Delete (Q, x) starts by increasing the counter *deleted* and storing x in the array *Del*. Then, the operation Take-Up(x) is invoked to update the data structure locally to x guaranting that any subsequent operation before the final arrangement, executed by the Arrange-Heap procedure, can be accomplished correctly.

4.1 Procedure Take-Up

This procedure guarantees the following invariant.

Invariant 1 *: After each execution of Take-Up, any node y of the binomial heap yields:*

1.1 if y is live, *L_y points to at least one binomial tree. In other words, there are at least two live nodes in the sub-binomial tree rooted at y, namely the root of the binomial tree pointed by L_y and y itself.*

1.2 if y is empty, *the sub-binomial tree rooted at y is empty. the binomial trees (if any) pointed by D_y are formed by all empty nodes.*

1.3 for each position i, with $0 \leq i \leq y \uparrow .degree$, $D_y[i] \neq nil$ or $L_y[i] \neq nil$. That is, the binomial tree rooted at y is complete, but some of its nodes are empty.

Let us now explain in details the procedure Take-Up in order to verify the Invariant 1. Take-Up(x) makes the node x empty substituting the key of x with $-\infty$. Let the parent of x be $p(x)$, and the degrees of x and $p(x)$ be k_x and $k_{p(x)}$ respectively. Then, $L_{p(x)}[k_x]$ is set to *nil* since the root x of the binomial tree B_{k_x} has become an empty node. Although B_{k_x} is no longer in $L_{p(x)}$, it cannot be inserted in $D_{p(x)}$ as some live nodes may still be stored in it. Therefore, in order to verify Invariant 1, B_{k_x} and the sub-binomial tree rooted at $p(x)$ are melded as follows. The list $D_{p(x)}$ of the children of $p(x)$, which are empty binomial trees, is updated by inserting in it the single node x (ignoring its children and parent information) as well as D_x. In this way, Invariant 1.2 is satisfied. Similarly, $L_{p(x)}$ must be rebuilt in such way that Invariant 1.1 is also guaranteed. Given two binomial trees yielding Invariant 1.1 and combining them by the linking rule, a new binomial tree satisfying Invariant 1.1 is obtained. So, $L_{p(x)}$ is rebuilt by Union($L_{p(x)}, L_x$). Finally, after the melding between the children arrays L and D, each root of the resulting binomial trees are made pointing to its parent, $p(x)$. In fact, after union there could be some roots either in $L_{p(x)}$ or $D_{p(x)}$ which have the pointer, *parent*, pointing to its previous parent, x. So these steps are necessary to restore the *parent* relation among the merged children.

As the sub-binomial tree rooted at $p(x)$ was initially a complete binomial tree of order $k_{p(x)}$ and no node has actually been removed from it, it holds either $D_{p(x)}[i] \neq nil$ or $L_{p(x)}[i] \neq nil$ for each position i, where $0 \leq i \leq k_{p(x)}$, as claimed in Invariant 1.3.

Figure 3(a) illustrates a binomial heap before the Take-Up procedure is applied to the node x. The lists of the children of $p(x)$ are $D_{p(x)}[0] := z$, $D_{p(x)}[1] := nil$ and $D_{p(x)}[2] := nil$ while $L_{p(x)}[0] := nil$, $L_{p(x)}[1] := y$, and $L_{p(x)}[2] := x$. For node x, $D_x[0] := s$ and $D_x[1] := nil$, whereas $L_x[0] := nil$ and $L_x[1] := w$. For node y, $D_y[0] := nil$, $L_y[0] := t$. Figure 3(b) shows the same heap after the execution of Take-Up(x). The children lists of node $p(x)$ are $D_{p(x)}[0] := z$, $D_{p(x)}[1] := x$, and $D_{p(x)}[2] := nil$ while $L_{p(x)}[2] := y$, and the remaining pointers are null. Fr node x, $D_x[0] := s$ and $L_x[0] := nil$ and for node y, $L_y[0] := t$ and $L_y[1] := w$. The situation at the node z is the same before and after the Take-Up operation.

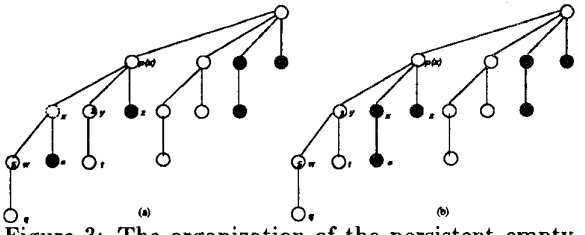

Figure 3: The organization of the persistent empty nodes represented by black nodes: (a) before Take-Up(x), and (b) after Take-Up(x).

The main goal of the Take-Up procedure is to preserve the data structure for future operations – Min, Extract-Min, Union, Insert, Change-Key – in the presence of persistent deleted nodes. For example, if no local arrangement is done, Extract-Min could find all the roots of the binomial heap empty, thus being unable to return the minimum key in the claimed time bounds. In contrast, after the Take-Up operation, if all the roots of the binomial heap are empty, we have ensured (see Invariant 1.2) that the binomial heap itself is void.

In conclusion, it is easy to see that Invariant 1 is satisfied after the execution of the Union operation as well as any other operation. Given H and H' whose nodes satisfy Invariant 1, the binomial tree returned by the Union(H, H') will preserve Invariant 1. In fact, Union(H, H') will link the lists of the binomial trees not empty in H and H'. As mentioned while explaining the Take-Up procedure, the linking rule applied to the live roots preserves Invariant 1.1.

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 $\lfloor \log n \rfloor + 1$ and at most one union is invoked.

4.2 Procedure Arrange-Heap

We have seen how the Take-Up procedure organizes the deleted nodes in the binomial trees after each deletion. Then, after $\lfloor \frac{\log n}{\log\log n} \rfloor$ deletion, the empty nodes are released by the Arrange-Heap procedure. First of all, the empty nodes are bubbled up. The bubbling up of the empty nodes takes place in parallel assigning one processor to each empty node and using some tricks to avoid concurrent writing on the same memory location. In fact, a node v with $key \neq -\infty$ can have two (or more) empty nodes v_1 and v_2 among its children. So the processors which deal with them, could try to access concurrently v and simultaneously swap its key. No conflict arises if the bubbling up process is organized as follows:

Fact 3 *: If the empty nodes are ordered according to their distances from the roots, and the operations for the bubbling up process are scheduled in a pipelined manner starting into the empty nodes of minimum distance and considering nodes at the same distance from the left to the right, there will be at most one processor attempting to access a given node. In other words, no access conflicts will arise.*

However, the more powerful CREW PRAM model is now required to compute the distance of each empty node from the root. When the bubbling up phase is complete, a node is ignored if it contains an empty key. Moreover, as all the empty nodes have been gathered to their root and deleted, it is ensured that if a node is live, then all the nodes in the subtree rooted at it are live. Therefore, to regenerate the heap it will be sufficient to meld the live children of the deleted nodes with the untouched binomial trees of H (if any). The untouched trees correspond to the binomial trees of H which contained no persistent nodes at the beginning of the Arrange-Heap procedure. Precisely, we first generate a new heap H' from all the lists L of the live children of the deleted nodes, and then meld H' with the complete sub-binomial trees of H.

Letting $x_i = Del[i]$, for $i = 1, \ldots, \lfloor \frac{\log n}{\log\log n} \rfloor$, we use a balanced binary tree technique to generate H' from arrays L_{x_i}. In the first iteration, the pairs of arrays $(L_{x_1}, L_{x_2}), (L_{x_3}, L_{x_4}), \ldots$ are combined in parallel. Each pair united by two processors returns a single heap. So at the end of the first iteration, we have $\lceil \frac{\log n}{2\log\log n} \rceil$ new heaps. Analogously, at the second iteration each pair of these new heaps is combined using 4 processors yielding $\lceil \frac{\log n}{4\log\log n} \rceil$ heaps. In general, at the i-th iteration, $\lceil \frac{\log n}{2^i \log\log n} \rceil$ heaps are obtained from the ones of the $(i-1)$th iteration. Each pair at iteration i is combined in parallel using 2^i processors. Hence, after $O(\log(\log n/\log\log n)) = O(\log\log n)$ iterations, a new heap H' is generated.

Using our parallel Union operation for each pair, the time for the i-th iteration is $O(\log\log n + (\log n)/2^i)$. The time for generating the heap H':

$$\sum_{i=1}^{\log\log n} (\log\log n + \tfrac{\log n}{2^i}) = O((\log\log n)^2 + \log n) = O(\log n).$$

Finally, the heap H' and the complete sub-binomial trees of H are combined to result in the regenerated heap which does not contain empty nodes. The time spent for this is $O(\log\log n)$. Therefore, the operation Arrange-Heap requires $O(\log n)$ total time and $O(\log^2 n/\log\log n)$ work on the CREW PRAM model.

Theorem 2 *: A sequence of $O(\frac{\log n}{\log\log n})$ Delete operations on the CREW PRAM model, requires $O(\log n + \frac{\log^2 n}{p\log\log n})$ time and $O(\log^2 n/\log\log n)$ work employing $p = O(\frac{\log n}{\log\log n})$ processors. Therefore, each Delete requires $O(\log\log n)$* amortized *parallel time and $O(\log n)$* amortized *work.*

Moreover, since a *Change-Key* operation can be implemented by a Delete operation followed by a suitable Insert operation, the Change-Key operation achieves the same performance as above.

5 Hypercube Implementation

In this section, we implement a a distributed meldable priority queue based on binomial heaps. We will see that an optimal mapping, which distributes the data structure among the processors on the hypercube and also minimizes the exchange of data during the manipulation of the heap, is not enough to achieve cost-optimal implementation of the usual heap operations such as Insert, Extract-Min and Union. Therefore, a slightly different underlying data structure for a distributed meldable priority queue will be introduced.

Let Q_q be a q-dimensional hypercube and let Π be an embedded Hamiltonian path in Q_q, based on the binary reflected Gray-codes [7]. Also, let $\Pi(i)$ be the ith processor on this path, for $0 \leq i \leq 2^q - 1$.

Definition 4 *: A* distributed binomial heap *H of size n is mapped onto Q_q such that for a node $x \in H$ with $degree(x) = i$, the node x is assigned to the processor* $\Pi(i \bmod 2^q)$.

Since the maximum degree and the minimum degree of a binomial heap of size n are $\lfloor \log n \rfloor$ and 0, respectively, no more than $\lfloor \log n \rfloor + 1$ processors can be used effectively.

Figure 4 depicts the mapping of a binomial heap of size 27 onto a hypercube Q_2 along with the Hamiltonian-path Π of a hypercube Q_2. Therefore, $\Pi(0) = 0, \Pi(1) = 1, \Pi(2) = 3, \Pi(3) = 2$.

The proposed mapping has interesting properties.

Property 1 *: For any $0 \leq i \leq \lfloor \log n \rfloor - 2^q$, the roots of the binomial trees $B_i, \ldots, B_{i+2^q-1}$ of H are mapped onto the processors $\Pi(i \bmod 2^q), \Pi((i+1) \bmod 2^q), \ldots, \Pi((i+2^q-1) \bmod 2^q)$ of Q_q. That is, the processors to which they belong, form the Hamiltonian path Π of Q_q.*

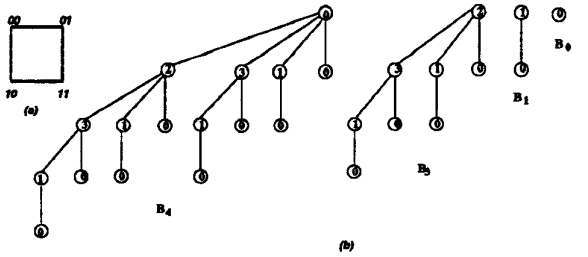

Figure 4: Hypercube mapping of a binomial heap. Each node contains the index of the processor assigned to it.

Since a node x of degree i is the root of a sub-binomial tree B_i, it holds:

Property 2 : *A node x of degree i, where $0 \leq i < 2^q \leq \lfloor \log n \rfloor$, and its children in decreasing order of the degree are embedded in the sub-path $\Pi(i), \ldots, \Pi(0)$ of the hypercube Q_q. Moreover, any 2^q adjacent children of a node x of degree i, where $2^q \leq i \leq \lfloor \log n \rfloor$, are embedded in the Hamiltonian path Π of Q_q.*

Unfortunately, the memory load for each processor is not well balanced in this scheme. Namely, in a binomial heap of size $n = 2^k - 1$, there are 2^{k-j-1} nodes having degree j, where $0 \leq j \leq k-1$, and all these nodes are assigned to the same processor $\Pi(j \bmod 2^q)$. However, this mapping is convenient when the linking rule is applied. Since only the root of the new binomial tree changes its degree, to preserve the mapping it is sufficient to move information relative to that root between two adjacent processors. Precisely:

Property 3 : *When two binomial trees B_i are combined to obtain B_{i+1}, where $0 \leq i \leq \lfloor \log n \rfloor$, the mapping is preserved by moving the root of the binomial tree with the smaller key from $\Pi(i \bmod q)$ to $\Pi(i+1 \bmod q)$. Since a node of the binomial tree has $O(\log n)$ information associated with it, $O(\log n)$ time is required to implement the linking rule.*

From now on, the pointer to a node y is substituted with the pair $(P(y), L(y))$ where $P(y) = \Pi(degree(y))$ is the processor to which y is mapped and $L(y)$ is the local address of the node y in the memory of $P(y)$. Moreover, the variable $H[i]$ stored into the processor $\Pi(i)$ points to the root of the binomial tree B_i in the binomial heap H if it exists, or $H[i]$ is nil if $B_i \notin H$.

Even if only one node is moved while applying the linking rule, we are still spending a time proportional to the more expensive operation on the binomial heaps in sequential environment. Hence, to overcome this drawback, we introduce a working data structure, called *b-bandwidth binomial heap* or b-binomial heap.

Definition 5 : *In a* b-bandwidth binomial tree, *each node stores an array of b keys (instead of a single key) sorted in non-decreasing order. The* heap order *is extended to the bandwidth binomial tree such that each key in a node x is no smaller than each key in its parent $p(x)$.*

A b-binomial heap H *of size nb is a collection of $\lfloor \log n \rfloor + 1$, b-bandwidth binomial trees and supports the following operations:* **(1)** Make-Queue(H) *to create and return a new empty queue H,* **(2)** Multi-Insert($H, K[1, \ldots, b]$) *to insert simultaneously b items,* **(3)** Multi-Extract-Min(H) *to delete the smallest b items from H,* **(4)** b-Union(H_1, H_2) *to meld two b-bandwidth heaps.*

Definition 6 : *A distributed meldable priority queue $\mathcal{Q}$ of size nb consists of:*

(*i*) *a* b-binomial heap H *of size nb.*

(*ii*) *two arrays of size b, namely* Forehead($\mathcal{Q}$) *and* Waiting($\mathcal{Q}$). *Forehead($\mathcal{Q}$) contains in sorted order the items returned by the last Multi-Extract-Min(H) operation but not yet actually deleted by $\mathcal{Q}$. Waiting($\mathcal{Q}$) is a binary min-heap containing the items to be inserted in $\mathcal{Q}$ but not yet inserted in H. Both* Forehead($\mathcal{Q}$) *and* Waiting($\mathcal{Q}$) *are stored in some prespecified processor, generically referred to as the I/O processor.*

The priority queue $\mathcal{Q}$ based on the b-binomial heap supports the standard operations Make-Queue, Insert, Min, Extract-Min operations presented in Section 3, and also the b-Union operation between two b-binomial heaps. In the standard operations, Forehead($\mathcal{Q}$) and Waiting($\mathcal{Q}$) act as two buffers.

To perform Insert($\mathcal{Q}, x$), the single key x in the min-heap Waiting($\mathcal{Q}$) is inserted. After receiving b new items, Multi-Insert($H, K[1, \ldots, b]$) is invoked for actually inserting in H the largest b items, namely $K[1, \ldots, b]$, of Forehead($\mathcal{Q}$) and Waiting($\mathcal{Q}$). The Extract-Min($\mathcal{Q}$) operation returns the smallest item among the elements stored in Forehead($\mathcal{Q}$) and Waiting($\mathcal{Q}$) if Forehead($\mathcal{Q}$) is not empty. After b smallest items are returned, a Multi-Extract-Min(H) is invoked. Clearly, an Insert or Extract-Min which does not involve multiple operations will require $O(\log b + q)$ time for deleting/inserting a value into Waiting (as necessary) as well as sending/receiving the reply from the I/O processor. It still remains to discuss how to implement the multiple operations. As both Multi-Insert-Key and Multi-Extract-Min are based on the b-Union(Q_1, Q_2) operation, we can limit our discussion to the implementation of the b-Union operation. The Union operation will follow the PRAM implementation (Section 3) in a straightforward manner, except for a preprocessing phase working on the keys stored in the roots of the binomial trees of H_1 and H_2. The preprocessing stage is accomplished in order to satisfy the extended heap order (Definition 5).

During the *preprocessing* phase, each maximum key stored in the roots of the binomial trees forming H_1 and H_2 are sorted in non-decreasing order. Let $B_{i_0}, B_{i_2}, \ldots, B_{i_f}$, where $f \leq \lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor$, be the sequence of binomial trees induced by ordering the maximum keys in their roots. Then, the maximum key stored in the root of B_{i_j} is smaller than or equal to the maximum key stored in the root of $B_{i_{j+1}}$, for $0 \leq j \leq f$. The $bf + b$ keys stored in the roots

are sorted in non-decreasing order, and the sorted sequence is distributed in such way that its items in positions $jb, \ldots, (j+1)b-1$ are assigned to the root B_{i_j}, where $0 \leq j \leq f$.

Let us analyze the complexity of the preprocessing phase, which preserves the heap-order even when the linking rule is applied. The sorting of P items on a P-processor hypercube requires $O(P \log P)$ time [7]. Optimal algorithms are also known [7] for sorting M items in $O(\frac{M \log M}{P})$ time in the hypercube for any $M \geq P^{1+\epsilon}$, where ϵ is a constant. Furthermore, M packets can be routed in $O(\frac{M \log P}{P})$ time on the P-processor hypercube assuming that each processor starts and finishes with $O(\frac{M}{P})$ packets. Thus, for $P = 2^q$ and $M = O(b \log n)$, the overall preprocessing time complexity is $O(\frac{b \log n(\log b + \log\log n)}{2^q})$ where $b = \Omega(2^{q\epsilon})$ and $\epsilon > 1$.

Then, we apply Phases I and II (which define linking chains and perform prefix minima on these chains) to the b-bandwidth binomial heaps working on the maximum key stored in each root of the binomial trees. As seen in Section 3, Phases I and II are accomplished by suitable prefix computations on the roots of the binomial heaps. Hence, from Property 1, prefix computations on the hypercube nodes are performed in such way that $\Pi(i)$ stores the sum of the elements stored in the processors $\Pi(0), \ldots, \Pi(i)$. This is done by a prefix computation, called *Hamiltonian Prefix*, on a suitable binary tree embedded into the Hamiltonian path, and requires $O(\frac{\log n}{2^q} + q)$ time [3, 4].

Finally, to actually build the melded heap, Phase III requires data movements. For example, some roots become children of a dominant root. This dominant root receives the addresses of the new children and then, it is moved to a different processor since its degree has changed.

According to Fact 1.1, all binomial trees B_i, for $i_s < i \leq i_e$, in the same fragment are in the increasing order. The addresses of their roots will be sent to the dominant root, stored in $\Pi(i_s)$, along the Hamiltonian path starting from $\Pi(i_e)$ and ending at $\Pi(i_s+1)$. This can be done in parallel for each fragment, and the required time is proportional to the maximum length of the fragment, i.e., $O(\log n)$. Moreover, from Fact 1.2, the dominant root will have degree i_e+1. This means that the updated root stored in $\Pi(i_s)$ must be moved to $\Pi(i_e+1)$, requiring time proportional to the amount of $O(\log n)$ information transferred multiplied by the distance between $\Pi(i_e+1)$ and $\Pi(i_s)$. Observing that all the dominant roots can be moved together on the Hamiltonian path, at most $O(\log^2 n)$ time is required. Hence:

Theorem 3 : *Using 2^q processors, the b-Union operation takes $O(\log^2 n + \frac{b \log n \log b}{2^q})$ time, where $b \geq 2^{q\epsilon}$ and $\epsilon > 1$. Hence, a single Insert or Extract-Min has amortized cost $O(\frac{T 2^q}{b}) = O(\log n \log b + \frac{2^q \log^2 n}{b})$. In particular, when $2^q = O(\log n)$ and $b = \Omega(\frac{\log^2 n}{\log\log n})$, Insert and Extract-Min require $O(\log\log n)$ amortized time and $O(\log n \log\log n)$ cost, thus achieving an amortized speed-up of $O(\frac{\log n}{\log\log n})$.*

6 Conclusions

We have presented on the PRAM model an optimal parallel implementation of meldable priority queues based on the binomial heap data structures. We have also proposed a dynamic mapping of meldable priority queues (based on binomial heaps) on the hypercube networks, which has low communication overhead and a sub-optimal amortized speed-up. However, designing a cost-optimal hypercube algorithm as well as avoiding the CREW PRAM model for the Delete operation are still open problems, which are the focus of our future research.

References

[1] D. Z. Chen and X. Hu, "Fast and Efficient Operations on Parallel Priority Queues", *Proceeding of the Fifth Annual International Symposium on Algorithms and Computation*, Beijing, China, August 1994, pp. 279-287.

[2] Cormen, T.H., Leiserson, C.E. and Rivest R.L., *Introduction to Algorithms*, McGraw Hill, New York, 1990.

[3] S.K. Das, M.C. Pinotti, and F. Sarkar, "Optimal and Load Balanced Mapping of Parallel Priority Queues in Hypercubes", *IEEE Trans. on Parallel and Distributed Systems*, to be published, 1996.

[4] S.K. Das, M.C. Pinotti, e F. Sarkar, "Efficient Implementation of Min-Max Heap and Hamiltonian-Suffix in a Hypercube", Tech Rep. CRPDC-96-1, Dept. of Computer Sciences, University of North Texas, Denton, TX 76203, Jan. 1996.

[5] N. Deo and S. Prasad, "An Optimal Parallel Priority Queue", *Journal of Supercomputing*, Vol. 6, 1992, pp. 87-98.

[6] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA 1992.

[7] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[8] S. Olariu and Z. Wen, "Optimal Initialization Algorithms for a Class of Priority Queue", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 2, No. 4, 1991, pp. 423-430.

[9] M. C. Pinotti and G. Pucci, "Parallel Priority Queues", *Information Processing Letters*, Vol. 40, 1991, pp. 33-40.

[10] M.C. Pinotti and G. Pucci, "Parallel Algorithms for Priority Queue Operations", *Theoretical Computer Science*, Vol. 148, 1995, pp. 171-180.