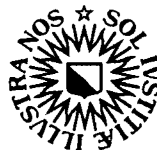
On parallel data structuring: A parallel priority queue

S.T. Fischer, M. Veldhorst

RUU-CS-92-19
April 1992



Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

On parallel data structuring: A parallel priority queue

S.T. Fischer, M. Veldhorst

Technical Report RUU-CS-92-19
April 1992

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

On Parallel Data Structuring: A Parallel Priority Queue *

S.T. Fischer[†] and M. Veldhorst
Departement of Computer Science, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Abstract

In this paper we design a priority queue that is suitable for parallel access on an EREW PRAM. To delete p elements with lowest priority from the priority queue takes $O(\log p + \log\log p + \frac{\log n}{p})$ time. The insertion of p elements in the priority queue takes $O(\log n + \log p)$ time. To decrease the priority of p elements in the priority queue also takes $O(\log n + \log p)$ time.

1 Introduction

In the design of sequential algorithms often better time bounds have been obtained by the use of sofisticated data structures, from which intermediate results could be obtained easily (i.e. in short time). It seems reasonable to expect that time bounds of algorithms for the PRAM can be improved by using advanced data structures, that allow for parallel access. Veldhorst [Vel87] gave an implementation of a stack and a queue for an EREW PRAM. Pinotti and Pucci [PP91] gave an implementation of a priority queue for a CREW PRAM.

In this paper we will give an implementation of a priority queue for an EREW PRAM. Formally, a priority queue PQ stores a set of n elements e, each with a priority $pr(e)$. A processor PE_i, $1 \leq i \leq p$, can perform one of the following operations on PQ.

1. $ParExtractmin(PQ)$
   Processor PE_i deletes an element from PQ, with a priority as low as possible that no processor PE_j deletes from PQ, $1 \leq j < i$.

*This work was supported by the ESPRIT II Basic Research Actions program of the EC under contract No 3075 (project ALCOLM)

[†]current address: Faculteit Wiskunde en Informatica, Plantage Muidergracht 24, 1018 TV Amsterdam, The Netherlands. This work has been partly supported by the Foundation for Computer Science in the Netherlands (SION) with financial support from the Netherlands Organization for Scientific Research (NWO)

2. $ParInsert(PQ, \langle e, pr(e)\rangle)$
   Processor PE_i inserts element e with priority $pr(e)$ in PQ.

3. $ParDecreasekey(PQ, \langle q, pr\rangle)$
   Processor PE_i decreases the priority of the element q to pr.

In our implementation, the *ParExtractmin* operation takes $O(\log p + \log\log p + \frac{\log n}{p})$ time. The *ParInsert* and the *ParDecreasekey* operation take $O(\log n)$ time. The elements of PQ are stored in the leaves of a weight-balanced tree. To achieve the time bound for *ParExtractmin*, we use the property of weight-balanced trees that the balance of a vertex v does not depend on the balances of its children, but only on the number of elements stored in its left and right subtree.
There are many sequential implementations of the priority queue. The figures 1 and 2 show results for different sequential implementations. In figure 1 worst-case time bounds are given, in figure 2 amortized time bounds. The binomial heap and fibonnacci heap are described in [CLR90]. Driscoll, Gabow, Shrairman and Tarjan, [DGST88], described a relaxed heap and a variant of the relaxed heap. Sleator and Tarjan, [ST86], described how to implement a priority queue using skew heaps.
Our time bounds for the parallel priority queue operations are low when compared

Priority queue implementation	*Extractmin*	*Insert*	*Decreasekey*
a stack or queue	$O(1)$	$O(n)$	$O(n)$
a binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
a balanced tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
a binomial heap	$O(\log n)$	$O(\log n)$	$O(\log n)$
a variant of a relaxed heap	$O(\log n)$	$O(1)$	$O(1)$

Figure 1: Worst-case time bounds of operations for different priority queue implementations

with the time bounds of the first five implementations of figure 1. We achieve an optimal speed up for the *ParInsert* and *ParDecreasekey* operation. If $p = \Theta(\log n)$, we achieve efficient speedup for the *ParExtractmin* operation. Our time bounds for the Insert and Decreasekey operations are high when compared with the time bounds of the last implementation of figure 1 and with the amortized time bounds of figure 2. But if $p = \Theta(\log n)$, we achieve a very fast time bound for the *ParExtractmin* operation.
There are two major differences between the implementation given in this paper and the implementation given by Pinotti and Pucci. First, their implementation is for

a CREW PRAM, while our implementation is for an EREW PRAM. Second, our time bounds are at least as good as those of Pinotti and Pucci, and our time bound for the *ParExtractmin* operation is better when $p = \Theta(\log n)$. In that case the *ParExtractmin* operation of our implementation runs in $O(\log\log n)$, while their *ParExtractmin* operation runs in $O(\log n)$ time. When comparing the time bounds, we accounted time for distributing the elements among all processors. This time is not included in the time bounds Pinotti and Pucci give in their paper.
The remainder of this paper is divided in 3 sections. Section 2 contains definitions, results and algorithms used throughout the paper. In section 3 we investigate the special properties of balancing a weight-balanced tree. Section 4 contains the implementation of the priority operations. We give two insert algorithms. The first is very easy to understand. The second is more difficult, but minimizes the communication between the processors.

2 Preliminaries

2.1 Definitions

The computational model
In this paper the computational model used is an EREW PRAM. A PRAM consists of p processors $PE_1, ..., PE_p$. Every processor has its own local memory. Furthermore, all processors have access to a global memory. Figure 3 shows a PRAM.
At a certain time step some processors are off, and do nothing during that time step and some processors are on. The processors that are on perform all the same operation, possibly on different data.
In an Exclusive Read Exclusive Write PRAM only one processor is allowed to read or write a global memory location at a certain time step.

Abstract Data Structures
An abstract data structure is a tuple $\langle S, OP \rangle$, where S is a set of elements and OP is a set of operations, that can be applied on the set.

Priority queue implementation	*Extractmin*	*Insert*	*Decreasekey*
a fibonnacci heap	$O(\log n)$	$O(1)$	$O(1)$
a relaxed heap	$O(\log n)$	$O(1)$	$O(1)$
a skew heap (bottom-up)	$O(\log n)$	$O(1)$	$O(1)$

Figure 2: Amortized time bounds of operations for different priority queue implementations

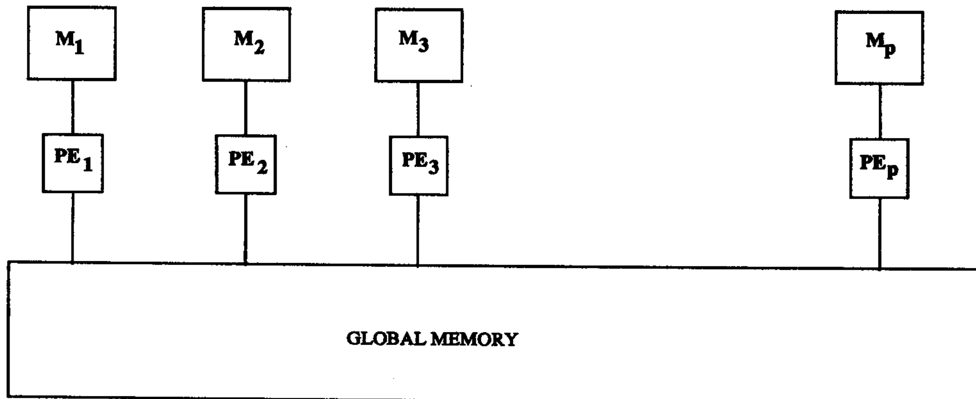


Figure 3: A PRAM

An abstract data structure is **implemented** using data structures (or standard data structures). For instance, the abstract data structure $STACK$ can be defined as $\langle S, OP \rangle$ where S is the set of stack elements $STACK$ and the set OP contains the operations $CREATE$, $ISEMPTY$, POP, $PUSH$ and TOP. A stack can be implemented using an array, a linked list, a binary tree etc.

Parallel access and parallel operation
Let DS be a data structure. Then different processors in an EREW PRAM can read information stored in DS as long as processors do not read the same memory location. Also different processors can update information stored in DS, as long as processors do not write the same memory location.
When different processors perform each an application of the same operation on the same data structure **parallel access** occurs. It is not a priori clear what the result must be when all processors have finished their operation. If the result of the simultanious actions is not clear, it should be specified explicitely.
To define the result of a parallel operation, sometimes the sequentializing principle is used. Here the effect of simultanious actions by the processors is as if the actions occurred in some (unspecified) serial order.
In the priority queue we develop in this paper, the sequentializing principle is obeyed when the effect of simultanious actions on the abstract data structure is considered. The sequentializing principle is not neccesarily obeyed when we consider the effect of the simultanious actions on the weight-balanced tree that implements the priority queue.

Access trees
An access tree for p processors is a complete binary tree AT with leaves $l_1, l_2, ..., l_{2^{\lceil \log p \rceil}}$. Processor PE_i is associated with l_i. Suppose for instance that processors $PE_{i_1}, PE_{i_2}, ..., PE_{i_s}$ all want to read the same memory location. By letting the processors walking up and

down in AT it is possible to decide which processor is going to read the memory location and to inform all processors of its information in $O(\log p)$ time using s processors. Access trees are sometimes called partial sums trees, or PS-TREES. See for details [SV82]. It is easy to see that the depth of an access tree is $O(\log p)$.

2.2 Results

Walking up in trees

In the algorithms of the next sections, processors walk up from leaves to the root of a binary tree. We use several techniques of walking up a binary tree, because we need different sorts of processor cooperation.

Let T be a binary tree of depth h. Suppose T has root r and suppose that processor PE_i is associated with leaf l_i. Not with every leaf necessarily a processor is associated. Let v be an internal vertex of T. Suppose a processor is associated with a leaf in $T_{left(v)}$ and another processor is associated with a leaf in $T_{right(v)}$. When these processors walk up from there associated leaves to r, they do not have to arrive at the same time at v. This gives rise to three walking up techniques.

1. The processors walk up in T. They avoid concurrent reads and writes when they arrive at the same time at a vertex v in the following way.
   Let v have children v_1 and v_2. When a processor PE_i is associated with v_1 and at that time no processor is associated with v_2, PE_i associates itself with v.
   When a processor PE_i is associated with v_1, and at the same time a processor PE_j is associated with v_2, PE_i is associated with v and stores index j in a broadcast queue. If necessary PE_i can inform PE_j later about some information. Processor PE_j has finished the walking up in T.
   It is possible that a vertex is visited by more than one processor. A difficulty of the technique is to decide when all processors have finished walking up in T. Notice that every path in T contains at most h vertices. So a processor has finished the walking up after it has visited at most h vertices.

2. In the second walking up technique again processors do not need to arrive at the same time at a vertex. But during the walking up at most one processor is associated with a vertex v. To achieve this each vertex v has a mark field $visited(v)$. The first processor associated with v sets $visited(v)$ to true. When later a processor PE_j visits v, it finishes the walking up. If necessary, PE_j waits at v to get informed by PE_i about certain data.
   To decide when all processors have finished their walking up, again the fact is used that every tree path contains at most h vertices.

3. In the third walking up technique processors are forced to arrive at the same time at a vertex v. Let v be an internal vertex of T. Suppose a processor PE_i is associated with a leaf in $T_{left(v)}$ and another processor PE_j is associated with a

leaf in $T_{right(v)}$. Suppose that during the walking up PE_i will once be associated with $left(v)$ and PE_j with $right(v)$. Then PE_i or PE_j will be associated with v only when PE_i is associated with $left(v)$ and PE_j is associated with $right(v)$. The other processor has finished walking up and waits, if necessary, at v to get informed about certain data.

As soon as a processor is associated with the root of T, all processors have finished walking up in T. To achieve this a vertex v contains two mark fields $arrived(v)$ and $information(v)$. The field $information(v)$ indicates whether there will ever be a processor associated with v. The field $arrived(v)$ is set to true when a processor is associated with v. The *information* fields are set using the first walking up technique. There will be a processor associated with v only if the fields $arrived(left(v))$ and $information(left(v))$ have the same value and the fields $arrived(right(v))$ and $information(right(v))$ have the same value. When a processor is associated with v $arrived(v)$ is set to true.

Theorem 2.1 *Let T be a binary tree of height h. Suppose that with every leaf in T a processor is associated. When the processors walk up in T using any of the walking up techniques described above, every processor visits at most h vertices.*

Pointer jumping

Let L be a linked list of cp vertices, c a constant. With every c (consecutive) vertices in L a processor is associated. With a processor at most c vertices are associated. It is possible to determine the rank of every vertex in L using p processors in $O(\log p)$ time. Furthermore, suppose that every vertex v in L contains an element e_v. Then also the minimum element stored in L or the sum of all elements stored in L can be computed in time $O(\log p)$, using p processors.

The information is computed using pointer jumping, see for instance [CLR90].

Memory space management

To avoid concurrent reads and writes memory space is stored in a stack. Processors can ask for memory using the operation *allocate*. They can put memory on the stack using the operation *free*. p Processors can get a new memory unit or return a memory unit in $O(\log p)$ time using p processors, [Vel87]. We will assume that units of memory space, for instance empty vertices, are stored in the stack.

Distribution and help indices of processors

We will give two results concerning the distribution of processors. We start this section with a result concerning the computing of help indices for processors.

Result 1 *Let $PE_1, ..., PE_p$ be the processors of an EREW PRAM, where every processor knows its index. Consider any sequence of processors $PE_{i_1}, ..., PE_{i_l}$ of the EREW PRAM, where $i_j \in \{1, ..., k\}$ and $i_j < i_k$ iff $j < k$. Then in $O(\log p)$ time every processor PE_{i_j} can compute j.*

Now the results concerning the distribution of processors.

Result 2 *Let T be a binary tree of height h with at least p leaves. Suppose that in every vertex v the number of leaves of T_v is stored.*
Then p processors can be distributed among the p leftmost leaves in $O(h)$ time.

Result 3 *Let L be a linked list with at most cp vertices, c a constant.*
Suppose that with every vertex in L a processor is associated. Suppose that every processor is associated with at most c, not necessarily consecutive, vertices in L.
It is possible to associate every processor with at most c consecutive vertices in $O(\log p)$ time using the p processors.

ParRebuild
Let L be a linked list containing cp elements, c a constant. With every c elements a different processor is associated. During the *ParRebuild* operation a BB[1/3] tree T is constructed. A BB[1/3] tree is a balanced binary tree. In section 3 we will give a definition of a BB[1/3] tree. The leaves of T contain the elements of L. Suppose processor PE_i is associated with $v_1, ..., v_c$. The algorithm *ParRebuild* consists of the following steps:

1. Every processor is associated with at most c consecutive vertices. Using Result 3, this can be accomplished in $O(\log p)$ time.
2. Every processor PE_i asks c empty vertices. With $c-1$ vertices PE_i builds sequentially a BB[1/3]-tree T_i. For details, see [Ove83]. Afterwards it associates itself with the root of T_i. Notice that PE_i has one empty vertex more.
3. Now the whole BB[1/3]-tree is built by constructing the tree level after level. To construct a level every processor PE_i maintains a value $ind(i)$. Initially $ind(i) = i$. Suppose PE_i is associated with root v_i, $1 \le i \le p$. If for PE_i $ind(i)$ is odd, PE_i uses its empty vertex to create a parent node for v_i and v_{i+1}, if such a vertex v_{i+1} exists. The processor PE_{i+1} is associated with this new parent vertex. Now the *ind* value of PE_{i+1} is divided by two and a new level can be constructed.
   Let j be an index such that at a certain layer the vertex associated with PE_j did not get a new parent. Then PE_j gets a new *ind*-value $\lfloor j/2 \rfloor$.

It is easy to associate with every vertex on the leftmost path of the newly built BB[1/3]-tree a different processor in $O(\log p)$ time.
The *ParRebuild* algorithm runs in $O(\log p)$ time using p processors.

ParSearch
Searching p values $\alpha_1, \alpha_2 \dots \alpha_p$ in a binary search tree of depth h using p processors can be done in $O(h + \log p)$ time, using techniques discussed by Wagner, Paul and Vishkin, [PVW83]. Since sorting p values on an EREW PRAM with p processors can be done in $O(\log p)$ time, [Col88], we will assume that $\alpha_1 \le \alpha_2 \dots \le \alpha_p$. Figure 4 gives a detailed description of an algorithm for parallel search in a binary search tree using

Input
Chain $C = \langle \alpha_1, \alpha_2, ..., \alpha_p \rangle$, $\alpha_1 \leq \alpha_2 \leq ... \leq \alpha_p$, and binary search tree of depth h. Processor PE_i is associated with α_i.
Output
A set of pointers p_i and a set of indices ind_i. Suppose C_j^i arrives at leaf l_i. Then p_i points to l_i and $p_k = nil$ for $i<k \leq j$ and $ind_k = i$ for $i \leq k \leq j$.
Chain $C_j^i = \langle \alpha_i, ..., \alpha_j \rangle$ is associated with internal node w. Node w has search value $l(w)$. Processor PE_i is active and knows α_j and index j. All processors PE_k, $i<k \leq j$ are off.

The PE_i compares α_i and α_j with $l(w)$.
if $l(w)<\alpha_i$ **then** *C_j^i is sent to w's right child.*
if $l(w)>\alpha_j$ **then** *C_j^i is sent to w's left child.*
if $\alpha_i \leq l(w) \leq \alpha_j$
then *C_j^i is divided in two chains C_M^i, C_j^{M+1}, where $M = \lceil \frac{i+j}{2} \rceil$.*
if $l(w)>\alpha_M$ **then** *C_M^i is sent to w's left child.*
if $l(w)<\alpha_{M+1}$ **then** *C_j^{M+1} is sent to w's right child.*

Initial
$C_j^i = C_p^1$ and w is the root of T.

Figure 4: Algorithm ParSearch

these techniques.
Let processor PE_k be associated with value α_k. During every step of the search algorithm PE_i tries to insert subchain $C_j^i = \langle \alpha_i, \alpha_{i+1}, ..., \alpha_j \rangle$ in subtree T_v, where v is an internal vertex of T. It is said that subchain $C_j^i = \langle \alpha_i, \alpha_2, ..., \alpha_j \rangle$ is associated with v, PE_i is active and PE_k is off, $i<k \leq j$. Initially C_p^1 is associated with the root of T. The insertion of a subchain in a subtree is done as follows.
Suppose that v has search value $l(v)$. During the step PE_i determines whether $\alpha_i \leq l(v) \leq \alpha_j$, $l(v)<\alpha_i$ or $l(v)>\alpha_j$. In the last two cases the whole subchain C_j^i can be sent to the right- or left child of v, respectively. Otherwise C_j^i is divided in two subchains C_M^i and C_j^{M+1}, where $M = \lfloor \frac{i+j}{2} \rfloor$. One of the two subchains is sent to a child of v. The vertices visited during a parallel search constitute search paths.

Definition 2.1 *Let T be a weight-balanced tree. Suppose a value α is searched in T during a parallel search. Suppose subchains containing α are associated with the vertices $v_1, ..., v_m$ in T during the parallel search . It is said that $v_1, ..., v_m$ constitute the search path of α in T.*

Notice that the search paths for α_i and α_j, $1 \leq i, j \leq p$, need not to be vertex disjoint. Lemma 2.2 states that during any step of the algorithm at most three subchains can be associated with a vertex. To prove lemma 2.2, we first need to prove another lemma. In

the lemma's, C_1, C_2 and C_3 are subchains. Let X be a real value then $X \leq C_i$ means that X is smaller than or equal to the smallest element in C_i and $C_i \leq C_j$ means that every element in C_i is smaller than or equal to every element in C_j.

Lemma 2.1 *Suppose that at a certain step i the subchains C_1, C_2 and C_3 are associated with a vertex v and $C_1 \leq C_2 \leq C_3$. Then at step $i-1$ C_2 was already associated with v as part of another subchain. During step $i-1$ the subchains C_1 and C_3 were sent from v's parent to v.*

Proof

The subchains C_1, C_2 and C_3 are associated with v at step i. During stage 1 there was a chain $C = \langle ..., C_1, ..., C_2, ..., C_3, ... \rangle$ associated with the root of T. Notice that the values in the subchains C_1,C_2 and C_3 followed the same search path in T from the root to vertex v, possibly as part of other subchains. This means that the fact, that C has been split, is caused by search values X on the search path with $X \leq C_1$ or $C_3 \leq X$. Suppose that at a certain time C is split between chains C_1 and C_2 by a search value X with $X \leq C_1$. Then the subchain $C' = \langle ..., C_2, ..., C_3, ... \rangle$ is sent to the right child of v. Later on C' is splitted between C_2 and C_3 by a label Y with $Y \geq C_3$. Afterwards $\langle ..., C_2, ... \rangle$ goes straight to vertex v. The subchain $\langle ..., C_2, ... \rangle$ will arrive at v at an earlier step than the subchain $\langle ..., C_3, ... \rangle$. Since C_2 and C_3 are associated with v at step i $\langle ..., C_2, ... \rangle$ is stopped at v by search value $l(v)$. Notice that $l(v) > C_1$ and $l(v) < C_3$. Therefore C_1 and C_3 cannot be stopped at v. Thus they arrive at step i at v.
The cases when C is split first between C_2 and C_3 by a search value $X < C_1$ or C is split first between C_1 and C_2 or C_2 and C_3 by a search value $Y > C_3$ are proved similarly.
End Proof

Lemma 2.2 *Let T be a binary search tree of depth h. During each step of the search algorithm at most three subchains are associated with a vertex in T.*

Proof

Suppose that during the first $i-1$ steps, $i \geq 1$, of the *ParSearch* algorithm at most two subchains are associated with a vertex in T. Then the lemma holds for the first $i-1$ steps.
The lemma for $s \geq i$ can be proved easily with induction. The argument used in the induction is the following. Suppose that at a stage $i+1$ the subchains C_1, C_2 and C_3 are associated with a vertex v. Then $C_1 < l(v) < C_3$. To see this, notice that according to lemma 2.1, C_2 was already associated with v during stage i as part of a subchain C_l^k. The subchain C_l^k has been split by the value $l(v)$, thus $\alpha_k < l(v) < \alpha_l$. The argument follows now from the fact that $C_1 < C_l^k < C_3$.
End Proof

Theorem 2.2 *The algorithm ParSearch has a time bound of $O(h + \log p)$ on a EREW PRAM.*

Proof
The lemmas imply that each step lasts $O(1)$ time, since during a step only a constant number of subchains is associated with a vertex.
Let α_i be some key in the initial chain C. Notice that C can be halved at most $\lceil \log p \rceil$ times. So α_i arrives at a leaf in $O(h + \log p)$ time.
End Proof

3 Weight-balanced Trees

3.1 Definition of Weight-balanced Trees

To implement the parallel priority queue we will use a special kind of binary search tree, a $BB[\alpha]$ or weight-balanced tree. Let T be a binary search tree. Data is stored in the leaves of T and all leaves are connected in a linked list. In the internal vertices of T a search value is stored. Every internal vertex v in T also contains the pointers *parent*, *left* and *right* to respectively the parent, left- and right child of v. The root of T has a parent pointer with value **nil**. Every internal vertex v has an additional value $size(v)$ which contains the number of elements stored in T_v. Throughout this paper T_v denotes the subtree of T rooted at v.

Definition 3.1 *[BM80] Let $\alpha \in]0,1[$ and let v be an internal vertex in a binary search tree T. The balance of v, denoted by $\beta(v)$, is defined by*

$$\beta(v) = \frac{size(left(v))}{size(v)}$$

A vertex v is called **balanced** *if $\alpha \leq \beta(v) \leq 1-\alpha$. Otherwise a vertex v is* **unbalanced**. *Tree T is a BB[α]-tree iff every internal vertex v is balanced. Tree T is a* **weight-balanced** *tree if there is an $\alpha \in]0,1[$, such that T is a BB[α] tree.*

It is easy to see that the depth of a BB[α]-tree is bounded by $O(\log n)$, [CLR90]. Vertices in a weight-balanced tree can become unbalanced, when an element is inserted or deleted. Notice that only vertices on the search path can become unbalanced. So after a single insert or delete operation $O(\log n)$ vertices are unbalanced. In this chapter we will only consider the problem of balancing an unbalanced vertex v with $\beta(v)<\alpha$. The case when $\beta(v)>1-\alpha$ is treated symmetrically.
An unbalanced vertex v can be balanced by a rotation at v or by rebuilding T_u, where $u = v$ or u is an ancestor of v. The figures 5 and 6 show two rotations, the single and the double rotation.
Let δ_1 be a vertex with right child δ_2 and let δ_2 have left child δ_3. Denote the balance of δ_i, $1 \leq i \leq 3$, after a rotation by $\beta'(\delta_i)$. Then after a single rotation

$$\beta'(\delta_1) = \frac{\beta(\delta_1)}{\beta(\delta_1) + (1-\beta(\delta_1))\beta(\delta_2)}$$

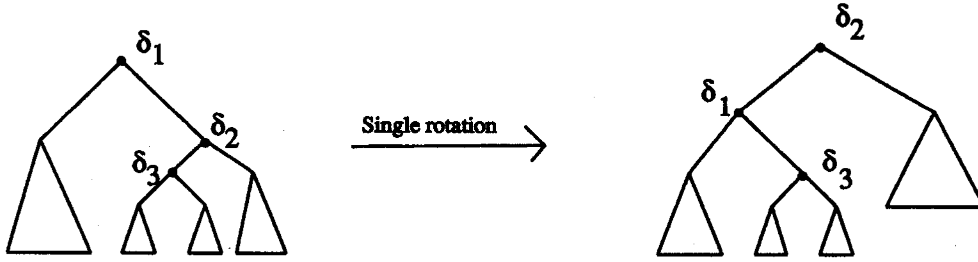


Figure 5: The single rotation

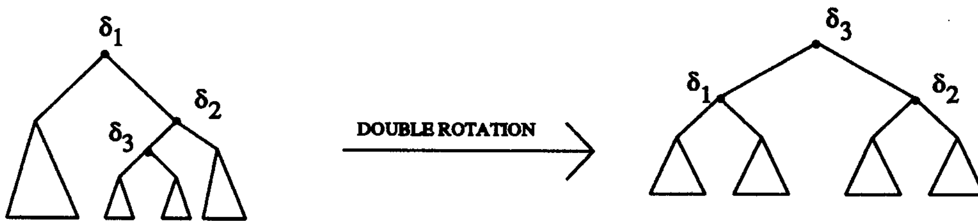


Figure 6: The double rotation

$$\begin{aligned}\beta'(\delta_2) &= \beta(\delta_1) + (1-\beta(\delta_1))\beta(\delta_2)\\ \beta'(\delta_3) &= \beta(\delta_3)\end{aligned}$$

And, after a double rotation

$$\begin{aligned}\beta'(\delta_1) &= \frac{\beta(\delta_1)}{\beta(\delta_1) + (1-\beta(\delta_1))\beta(\delta_2)\beta(\delta_3)}\\ \beta'(\delta_2) &= \frac{\beta(\delta_2)(1-\beta(\delta_3))}{1-\beta(\delta_2)\beta(\delta_3)}\\ \beta'(\delta_3) &= \beta(\delta_1) + (1-\beta(\delta_1))\beta(\delta_2)\beta(\delta_3)\end{aligned}$$

For details, see [BM80].
Blum and Melhorn prove the following theorem, [BM80].

Theorem 3.1 *Let $\alpha \in [\frac{2}{11}, 1 - \frac{1}{2}\sqrt{2}[$ and let δ_1, δ_2 and δ_3 be as before. Let $\beta(\delta_1) < \alpha$ and $\alpha \leq \beta(\delta_2)$, $\beta(\delta_3) \leq 1 - \alpha$. Suppose*

$$\frac{size(left(\delta_1))}{size(\delta_1) - 1} \geq \alpha$$

i.e. T is obtained by insertion of a leaf in T_{δ_2}, or

$$\frac{size(left(\delta_1)) + 1}{size(\delta_1) + 1} \geq \alpha$$

i.e. T is obtained by deletion of a leaf from $T_{left(\delta_1)}$.
Then if $\beta(\delta_2) \leq \frac{1}{2-\alpha}$, δ_1 can be balanced using a single rotation at δ_1.
If $\beta(\delta_2) > \frac{1}{2-\alpha}$, then δ_1 can be balanced using a double rotation at δ_1.

Rebuilding T_u is done by building a BB[1/3]-tree on the leaves of T_u. The algorithm *ParRebuild* described in section 2 can be used to rebuild T_u.

3.2 Rotations

In the previous section we stated a result of Blum and Melhorn. Now we will prove some stronger results concerning rotations. Let δ_1, δ_2 and δ_3 be as before. In the first theorem we show when δ_1 can be balanced using one rotation assuming that δ_2 and δ_3 are balanced.

Theorem 3.2 *Let $\alpha \in]2/11, 1-\frac{1}{2}\sqrt{2}[$, δ_1, δ_2 and δ_3 as before. Suppose $\beta(\delta_1) < \alpha$, $\alpha \leq \beta(\delta_2) \leq 1-\alpha$ and $\alpha \leq \beta(\delta_3) \leq 1-\alpha$. If*

$$\beta(\delta_1) \geq max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\}$$

Then

1. *δ_1 can be balanced using a single rotation if $\beta(\delta_2) \leq \frac{1}{2-\alpha}$. After the single rotation δ_2 and δ_3 are still balanced.*
2. *δ_1 can be balanced using a double rotation if $\beta(\delta_2) > \frac{1}{2-\alpha}$. After the double rotation δ_2 and δ_3 are still balanced.*

Proof
Suppose that $\beta(\delta_2) \leq \frac{1}{2-\alpha}$.
Then after a single rotation at δ_1

$$\begin{aligned} \beta'(\delta_1) &= \frac{\beta(\delta_1)}{\beta(\delta_1)+\beta(\delta_2)(1-\beta(\delta_1))} \\ &\leq \frac{\alpha}{\alpha+\alpha(1-\alpha)} \\ &\leq 1-\alpha, \text{ since } 1-3\alpha+\alpha^2 \geq 0 \end{aligned}$$

And

$$\begin{aligned} \beta'(\delta_1) &= \frac{\beta(\delta_1)}{\beta(\delta_1)+\beta(\delta_2)(1-\beta(\delta_1))} \\ &\geq \frac{\frac{\alpha}{(1-\alpha)(2-\alpha)}}{\frac{\alpha}{(1-\alpha)(2-\alpha)}+\frac{1}{2-\alpha}(1-\frac{\alpha}{(1-\alpha)(2-\alpha)})} \\ &= \frac{\alpha}{1-\frac{\alpha}{2-\alpha}} \\ &\geq \alpha \end{aligned}$$

Also after the single rotation δ_2 is balanced, since

$$\begin{aligned}\beta'(\delta_2) &= \beta(\delta_1)+\beta(\delta_2)(1-\beta(\delta_1)) \\ &\leq \alpha+\frac{1-\alpha}{2-\alpha} \\ &\leq 1-\alpha, \text{ since } 1-4\alpha+2\alpha^2 \geq 0\end{aligned}$$

And

$$\begin{aligned}\beta'(\delta_2) &= \beta(\delta_1)+\beta(\delta_2)(1-\beta(\delta_1)) \\ &\geq \alpha^2+\alpha(1-\alpha^2) \\ &\geq \alpha\end{aligned}$$

Now, suppose $\beta(\delta_2) > \frac{1}{2-\alpha}$ then for $\beta'(\delta_1)$:

$$\begin{aligned}\beta'(\delta_1) &= \frac{\beta(\delta_1)}{\beta(\delta_1)+\beta(\delta_3)\beta(\delta_2)(1-\beta(\delta_1))} \\ &\leq \frac{\alpha}{\alpha+\frac{\alpha(1-\alpha)}{2-\alpha}} \\ &= \frac{2-\alpha}{3-2\alpha} \\ &\leq 1-\alpha, \text{ since } 1-4\alpha+2\alpha^2 \geq 0\end{aligned}$$

and,

$$\begin{aligned}\beta'(\delta_1) &= \frac{\beta(\delta_1)}{\beta(\delta_1)+\beta(\delta_3)\beta(\delta_2)(1-\beta(\delta_1))} \\ &\geq \frac{\frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}}{\frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}+(1-\alpha)^2(1-(\frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}))} \\ &= \frac{\alpha}{\alpha+(1-\alpha)} \\ &= \alpha\end{aligned}$$

For the new balance of δ_2

$$\begin{aligned}\beta'(\delta_2) &= \frac{(1-\beta(\delta_3))\beta(\delta_2)}{1-\beta(\delta_2)\beta(\delta_3)} \\ &\leq \frac{(1-\alpha)^2}{1-\alpha(1-\alpha)} \\ &\leq 1-\alpha\end{aligned}$$

and,

$$\begin{aligned}\beta'(\delta_2) &= \frac{(1-\beta(\delta_3))\beta(\delta_2)}{1-\beta(\delta_2)\beta(\delta_3)} \\ &\geq \frac{\frac{\alpha}{2-\alpha}}{1-\frac{1-\alpha}{2-\alpha}} \\ &= \alpha\end{aligned}$$

At last, the new balance of δ_3

$$\begin{aligned}\beta'(\delta_3) &= \beta(\delta_1) + \beta(\delta_3)\beta(\delta_2)(1-\beta(\delta_1)) \\ &\leq \alpha + (1-\alpha)^3 \\ &\leq 1-\alpha, \text{ since } 1-3\alpha+\alpha^2 \geq 0\end{aligned}$$

and,

$$\begin{aligned}\beta'(\delta_3) &= \beta(\delta_1) + \beta(\delta_3)\beta(\delta_2)(1-\beta(\delta_1)) \\ &\geq \frac{1}{2}\alpha + \frac{\alpha(1-\frac{1}{2}\alpha)}{2-\alpha} \\ &= \alpha\end{aligned}$$

End Proof

In the next theorem we consider the situation when a rotation takes place at δ_1 and δ_2 and δ_3 are not necessarily balanced.

Theorem 3.3 *Let $\alpha \in [\frac{2}{11}, 1-\frac{1}{19}\sqrt{190}]$. Let δ_1, δ_2 and δ_3 be as before. Suppose that $0.9\alpha \leq \beta(\delta_1) < \alpha$ and that $0.9\alpha \leq \beta(\delta_2) \leq 1-0.9\alpha$. Then*

1. *The vertices δ_1 and δ_2 can be balanced using a single rotation at δ_1, if $\beta(\delta_2) \leq \frac{1}{2-\alpha}$.*
2. *The vertices δ_1 and δ_3 can be balanced using a double rotation at δ_1 if $\beta(\delta_2) > \frac{1}{2-\alpha}$ and $0.9\alpha \leq \beta(\delta_3) \leq 1-0.9\alpha$. After the double rotation $0.9\alpha \leq \beta'(\delta_2) \leq 1-0.9\alpha$.*

Proof
Suppose that $\beta(\delta_2) \leq \frac{1}{2-\alpha}$.
For the vertex δ_1, after a single rotation

$$\begin{aligned}\beta'(\delta_1) &= \frac{\beta(\delta_1)}{\beta(\delta_1)+\beta(\delta_2)(1-\beta(\delta_1))} \\ &\leq \frac{\alpha}{\alpha+0.9\alpha(1-\alpha)} \\ &\leq 1-\alpha, \text{ since } 0.9-2.8\alpha+0.9\alpha^2 \geq 0\end{aligned}$$

and,

$$\begin{aligned}\beta'(\delta_1) &= \frac{\beta(\delta_1)}{\beta(\delta_1)+\beta(\delta_2)(1-\beta(\delta_1))} \\ &\geq \frac{0.9\alpha}{0.9\alpha+\frac{1-0.9\alpha}{2-\alpha}} \\ &= \frac{1.8\alpha-0.9\alpha^2}{1+0.9\alpha-0.9\alpha^2} \\ &\geq \alpha, \text{ since } 0.8-1.8\alpha+0.9\alpha^2 \geq 0\end{aligned}$$

for vertex δ_2, after the single rotation

$$\begin{aligned}
\beta'(\delta_2) &= \beta(\delta_1) + \beta(\delta_2)(1 - \beta(\delta_1)) \\
&\leq \alpha + \frac{1-\alpha}{2-\alpha} \\
&= \frac{1 + \alpha - \alpha^2}{2 - \alpha} \\
&\leq 1 - \alpha, \text{ since } 1 - 4\alpha + 2\alpha^2 \geq 0
\end{aligned}$$

and,

$$\begin{aligned}
\beta'(\delta_2) &= \beta(\delta_1) + \beta(\delta_2)(1 - \beta(\delta_1)) \\
&\geq 0.9\alpha + 0.9\alpha(1 - 0.9\alpha) \\
&\geq \alpha
\end{aligned}$$

Suppose that $\beta(\delta_2) > \frac{1}{2-\alpha}$ and that $0.9\alpha \leq \beta(\delta_3) \leq 1 - 0.9\alpha$. For vertex δ_1 after a double rotation at δ_1,

$$\begin{aligned}
\beta'(\delta_1) &= \frac{\beta(\delta_1)}{\beta(\delta_1) + \beta(\delta_3)\beta(\delta_2)(1 - \beta(\delta_1))} \\
&\leq \frac{\alpha}{\alpha + \frac{0.9\alpha}{2-\alpha}(1 - \alpha)} \\
&= \frac{2 - \alpha}{2.9 - 1.9\alpha} \\
&\leq 1 - \alpha, \text{ since } 0.9 - 3.8\alpha + 1.9\alpha^2 \geq 0
\end{aligned}$$

and,

$$\begin{aligned}
\beta'(\delta_1) &= \frac{\beta(\delta_1)}{\beta(\delta_1) + \beta(\delta_3)\beta(\delta_2)(1 - \beta(\delta_1))} \\
&\geq \frac{0.9\alpha}{0.9\alpha + (1 - 0.9\alpha)^3} \\
&= \frac{0.9\alpha}{1 - 1.8\alpha + 2.43\alpha^2 - 0.729\alpha^3} \\
&\geq \alpha, \text{ since } 0.1 - 1.8\alpha + 2.43\alpha^2 - 0.729\alpha^3 \leq 0
\end{aligned}$$

For vertex δ_2 after a double rotation at δ_1,

$$\begin{aligned}
\beta'(\delta_2) &= \frac{\beta(\delta_2)(1 - \beta(\delta_3))}{1 - \beta(\delta_2)\beta(\delta_3)} \\
&\leq \frac{(1 - 0.9\alpha)^2}{1 - 0.9\alpha + 0.81\alpha^2} \\
&\leq 1 - 0.9\alpha, \text{ since } \alpha \geq 0
\end{aligned}$$

and,

$$\begin{aligned}
\beta'(\delta_2) &= \frac{\beta(\delta_2)(1-\beta(\delta_3))}{1-\beta(\delta_2)\beta(\delta_3)} \\
&\geq \frac{\frac{0.9\alpha}{2-\alpha}}{1-\frac{1-0.9\alpha}{2-\alpha}} \\
&\geq 0.9\alpha
\end{aligned}$$

For vertex δ_3 after a double rotation at δ_1,

$$\begin{aligned}
\beta'(\delta_3) &= \beta(\delta_1)+\beta(\delta_3)\beta(\delta_2)(1-\beta(\delta_1)) \\
&\leq \alpha+(1-0.9\alpha)^2(1-\alpha) \\
&= 1-1.8\alpha+2.61\alpha^2-0.81\alpha^3 \\
&\leq 1-\alpha, \text{ since } 0.8-2.61\alpha+0.81\alpha^2 \geq 0
\end{aligned}$$

and,

$$\begin{aligned}
\beta'(\delta_3) &= \beta(\delta_1)+\beta(\delta_3)\beta(\delta_2)(1-\beta(\delta_1)) \\
&\geq 0.9\alpha+0.9\alpha\frac{1}{2-\alpha}(1-0.9\alpha) \\
&\geq \alpha, \text{ since } 0.7-0.71\alpha \geq 0
\end{aligned}$$

End Proof

4 Implementation of the Priority Queue

4.1 Introduction

In this section we give an implementation for a priority queue for an EREW PRAM. Formally, a priority queue stores a set of n elements e, each with a priority $pr(e)$. A processor PE_i, $1 \leq i \leq p$, can perform one of the following operations on PQ.

1. $ParExtractmin(PQ)$
   Processor PE_i deletes an element from PQ, with a priority as low as posible that no processor PE_j deletes from PQ, $1 \leq j < i$.

2. $ParInsert(PQ, \langle e, pr(e)\rangle)$
   Processor PE_i inserts element e with priority $pr(e)$ in PQ.

3. $ParDecreasekey(PQ, \langle q, pr\rangle)$
   Processor PE_i decreases the priority of element q to pr.

Let S be a collection of abstract data structures $DS_1, ..., DS_n$. Suppose some processors perform an operation on DS_j, some processors perform an operation on DS_k, etc. In order to perform operations on DS_j the processors that access DS_j should get a help

index such that $PE_1, ..., PE_q$ will perform an operation on DS_j. To determine the help indices, with every abstract data structure an access tree is associated. Using this access tree PE_i can compute its help index in $O(\log p)$ time, see result 1. Before an operation starts, the processors $PE_1, \ldots, PE_q$ need to be distributed among the vertices on the leftmost path. Using an array of pointers with a pointer to every vertex on the leftmost path, this can be accomplished in $O(\log p)$ time. Let the leftmost path contain m vertices. Processor PE with helpindex j is associated to the vertices on the leftmost path to which the pointers of the array with index $(j-1)\frac{m}{q}+1, \ldots, j\frac{m}{q}$ point. Notice that the time bound for an operation on a collection of priority queues is determined by the slowest priority queue, i.e. the priority queue that takes most time to perform an operation.

4.2 Implementation

To implement a priority queue PQ suitable for parallel access we use a weight-balanced tree T. The elements of PQ are stored in the leaves of T, from left to right in the order of increasing priority. T is indexed using the priorities associated with the elements.
With every vertex on the leftmost path of T a processor is associated and a processor is associated with at most c consecutive vertices on the leftmost path. When inserting elements in PQ the elements are inserted in T. Where a new element e is inserted in T, depends on its associated priority $pr(e)$. So when p elements $e_1, e_2, ...e_p$ with associated priorities $pr(e_1), \ldots, pr(e_p)$ are inserted in PQ, first a search operation on T is performed for the values $pr(e_1), \ldots, pr(e_p)$. We assume that $p = \Theta(\log n)$.

4.3 ParExtractmin

Let weight-balanced tree T implement priority queue PQ. Suppose the processors $PE_1, \ldots, PE_p$ all want to perform $ParExtractmin(PQ)$. In this section we describe an algorithm that is performed by every processor PE_i, $1 \leq i \leq p$. Figure 7 shows the frame of the algorithm.

During the preprocess step, processor PE_1 determines the vertex v on the leftmost

---

Preprocess
Delete the elements from PQ with lowest priority that $PE_1, ..., PE_{i-1}$ do not delete from PQ.
Balance T
Restore

Figure 7: The frame of the algorithm that is executed by PE_i

---

path of T, such that $size(v) \geq p$ and v is as low in T as possible.
Then the processors $PE_1, \ldots, PE_p$ distribute themselves among the p leftmost leaves of T_v.
During the delete step the p leftmost leaves of T are deleted from the linked list. The

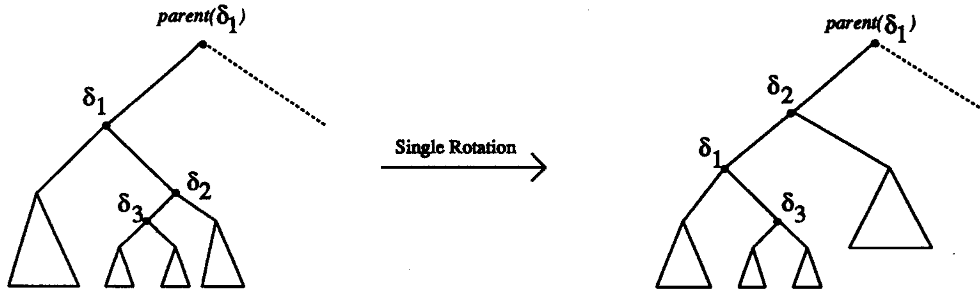


Figure 8: A single left rotation on the leftmost path of T

processors adjust the *size* information in the vertices on the leftmost path of T.
The algorithm finishes with a *restore* step. During the restore step processors are associated again with at most c adjoining vertices on the leftmost path of T.
The *balance* step of the algorithm is the most complicated step. During the balance step unbalanced vertices are balanced using rotations or by rebuilding subtrees. To decide how a vertex is balanced, first a vertex u is determined, that satisfies the following properties.

1. $u = v$ or u is an ancestor of v.

2. For every ancestor w of u:

$$\beta(w) \geq max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\}$$

3. $\beta(u) < max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\}$

Subtree T_u will be rebuilt. All unbalanced ancestors of u are balanced using a rotation. The rotations can be performed in $O(1)$ time.

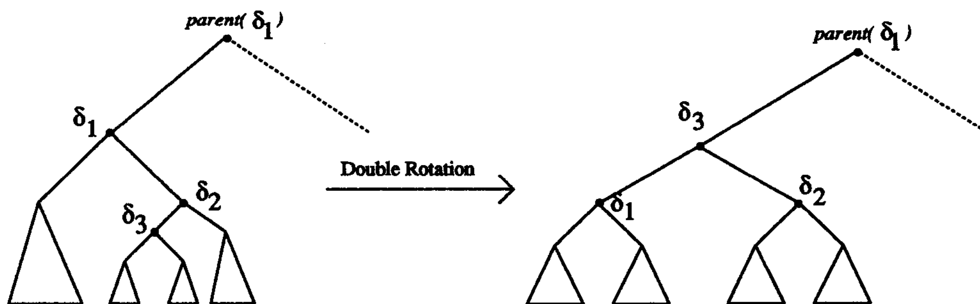


Figure 9: A double left rotation on the leftmost path of T

Lemma 4.1 *Let tree T be a BB[α]-tree. Let δ_1 be a vertex on the leftmost path of T. Let δ_1 have right child δ_2 and parent $parent(\delta_1)$ and let δ_2 have left child δ_3. Suppose that*

$$max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\} \leq \beta(\delta_1) < \alpha$$

Then

1. *When a rotation takes place at δ_1 only information is used and updated in the vertices δ_1, δ_2 and δ_3.*
2. *If after balancing δ_1, vertex r becomes the parent of δ_1, $r = \delta_2$ or $r = \delta_3$, then $parent(\delta_1)$ becomes the new parent of r. The relation between r and $parent(\delta_1)$ can not be altered by a leftrotation at $parent(\delta_1)$.*

Proof
The first fact stated in this theorem can be easily seen from the figures 5 and 6. The second fact can be seen from the figures 8 and 9.
End Proof

Notice that the vertices δ_2 and δ_3 are not on the leftmost path of T. Therefore all necessary rotations are independent of each other , and we have the following corollary.

Corollary 1 *Let T be a BB[α]-tree. All vertices on the leftmost path of T with*

$$max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\} \leq \beta(\delta_1) < \alpha$$

can be balanced in $O(1)$ time, using $p = \Theta(\log n)$ processors.

Proof
The vertices can be balanced using single or double rotations, see theorem 3.2. All single rotations can take place at the same time and all double rotations can take place at the same time, see theorem 4.1.
Because a rotation can be performed in $O(1)$ time, the parallel execution of all rotations takes also $O(1)$ time. We assumed that $p = \Theta(\log n)$.
End Proof

Rebuilding T_u can be done in $O(\log p)$ time.

Lemma 4.2 *Let u be as above. Then rebuilding T_u with p processors can be done in $O(\log p)$ time.*

Proof
Before deleting the elements from T_u, $\beta(u) \geq \alpha$. After deleting the elements from T_u, $\beta(u) < \gamma$, where $\gamma < max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\}$.

Thus before deletion T_u contained at most cp elements, where $c = \frac{1}{\alpha-\gamma}$. Using the *parRebuild* algorithm of section 2.2, it is easy to see that T_u can be rebuilt in time $O(\log p)$.
End Proof

The time complexity is stated in the following theorem.

Theorem 4.1 *Let PQ be a priority queue containing n elements. Then p processors of an EREW PRAM can perform a ParExtractmin operation on PQ in* $O(\log p + \log\log p + \frac{\log n}{p})$ *time.*

Proof
Notice that for $O(\log p)$ vertices on the leftmost path of T, $size(u) < p$. So when PE_1 walks up from the leftmost leaf of T to the root, it finds after $O(\log p)$ time a vertex u with $size(u) \geq p$. The distribution of the processors among the p leftmost leaves of T_v can be done in $O(\log p)$ time, see result 2. So the preprocess step takes $O(\log p)$ time. The delete step takes $O(1)$ time, since with every c vertices on the leftmost path of T a different processor is associated.
The balance step takes $O(\log p)$ time, see corollary 1 and lemma 4.2.
Finally, the restore step takes $O(\log p)$ time, see result 3.
End Proof

4.4 ParInsert

Introduction
In this section we give two insert algorithms that insert $e_1, \ldots, e_p$ with priority $pr(e_i), 1 \leq i < p$, in PQ using p processors in $O(\log n + \log p)$ time. Since sorting p values on an EREW PRAM with p processors takes $O(\log p)$ time, [Col88], we will assume that $pr(e_1) \leq pr(e_2) \leq \ldots \leq pr(e_p)$. The first algorithm is a parallelization of the sequential insert algorithm, [BM80], and works according a bottom-up approach. The disadvantage of this algorithm is that there is a lot of communication between processors.
The second algorithm minimizes the communication between processors, and works top down.
Both insert algorithms exist of three steps:

1. The places where the elements should be inserted are searched.

2. The elements are inserted.

3. The resulting tree is balanced.

If $pr(e_1)$ is smaller than the smallest priority of an element stored in T, then e_1 is first inserted. Therefore we assume that $pr(e_1)$ is equal to or greater than the smallest priority stored in T. In both insert algorithms processor PE_i is associated with element e_i.

Algorithm ParInsert1
In the first parallel insert algorithm the three steps of the introduction are described as follows:

1. The places in T, where the new elements should be inserted, are searched using the ParSearch algorithm on the key value $pr(e_1), \ldots, pr(e_p)$. During the search the information about the new balances is updated.

2. Every processor PE_s, $1 \le s \le p$, asks one empty vertex v_s. The empty vertices become the new leaves in T. The pointer $right(v_s)$ is set to vertex v_{s+1} iff $ind_s = ind_{s+1}$. Consider subchain $C_j^i = \langle \alpha_i, \ldots, \alpha_j \rangle$ with $ind_i = \ldots = ind_j$. Suppose p_i is set to leaf l_i and l_i has its pointer *right* set to leaf l. Processor PE_j stores l. The right pointer of l_i is set to v_i. Thus the elements of C_j^i are stored in a linked list. Now a BB[1/3]-tree $T_{C_j^i}$ is built on this list with *ParRebuild*. Then in T, l_i is replaced by $T_{C_j^i}$ and processor PE_i is associated with the root of $T_{C_j^i}$.

3. Consider T after the new elements have been inserted. Let u_i be an internal vertex in T, such that

$$\beta(u_i) < max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\}$$

or

$$\beta(u_i) > 1 - max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\}$$

and for every ancestor v_i of u_i,

$$max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\} \le \beta(v_i) \le 1 - max\{\frac{\alpha}{(1-\alpha)(2-\alpha)}, \frac{\alpha(1-\alpha)}{1+\alpha-\alpha^2}\}.$$

PE_i determines such a vertex u_i on the search path of α_i, if such a vertex u_i exists.

   (a) The subtrees T_{u_i} are rebuilt. The processors used for this rebuilding are the ones associated with the newly inserted elements in T_{u_i}. Since before insertion $\beta(u_i) \ge \alpha$ and after insetion $\beta(u_i) < 0.9\alpha$ there are enough processors to rebuild T_{u_i} in $O(\log p)$ time.

   (b) Unbalanced ancestors v_i of u_i are balanced using a rotation. No rotation takes place at v_i until all vertices in its left- and right subtree are balanced. To achieve this the processors walk up along tree paths using the third walking up technique. A single rotation is used if $\beta(right(v_i)) \le \frac{1}{2-\alpha}$. Otherwise

v_i is balanced using a double rotation.
Notice that only rotations performed at vertices on the leftmost path, cause an extra vertex to appear on the leftmost path. So with every c consecutive vertices on the leftmost path a processor can be associated in $O(\log p)$ time, see result 3.

Theorem 4.2 *The first parallel insert algorithm as described in this section runs in* $O(\log n + \log p)$ *time using* p *processors of an EREW PRAM.*

Proof
The search part takes $O(\log n + \log p)$ time using p processors.
The insert part takes $O(\log p)$ time using p processors.
The balance part takes $O(\log n + \log p)$ time using p processors, since every search path contains $O(\log n)$ vertices and rebuilding takes $O(\log p)$ time.
End Proof

Algorithm ParInsert2
Also the second parallel insert algorithm consists of a search part, an insert part and a balance part. The insert part of *ParInsert2* is exactly the same as the insert part of *ParInsert1*. The search and balance part of *ParInsert2* are described below. Except when subtrees are rebuilt no communication occurs between processors.

Search
During the search part again the places are searched where the new elements are inserted. Furthermore, information is gathered about which processor is going to balance which vertex. Processor PE_i maintains during the search path a balance queue Q_i.
Balance queue Q_i is filled using the following convention.
Let subchain $C = \langle k(e_i), \ldots, k(e_j) \rangle$ *be the first subchain that is associated with a vertex* v*. Then* v *is put in the queue* Q_i *of processor* PE_i*.*
To be able to decide whether a subchain is the first subchain visiting a certain vertex a field *associated*(v) is maintained in v. Before the search part *associated*$(v) = false$, for every internal vertex v.

Balance
Unbalanced vertices on search paths in T are balanced top-down. To avoid concurrent reads and writes, internal vertices in T store an *allowed* field. An internal vertex v in T is balanced using a rotation only if *allowed*$(v) = true$.
An internal vertex v stores also a field *rebuilt*(v). The field *rebuilt*(v) is set to true if there is a vertex u such that T_u will be rebuilt and $u = v$ or u is an ancestor of v. During the walking down all necessary rotations are performed. After the walking down subtrees are rebuilt. Initially for every internal vertex v, *allowed*$(v) =$ *rebuilt*$(v) = false$, except for the root r which has *allowed*$(r) = true$ and *rebuilt*$(r) = false$.

Let processor PE_i have associated balance queue Q_i.
while Q_i is not empty
do balance step on vertex v_i, where v_i is the first vertex on Q_i.
  if $allowed(v_i) = true$ or $rebuilt(v_i) = true$
  then remove v_i from Q_i and set $allowed(v_i)$ and $rebuilt(v_i)$ to $false$.
od
Rebuild the subtrees

Figure 10: The algorithm performed by processor PE_i during the balance part

The frame of the balance part is described in figure 10.
During each balance step a processor considers the first vertex on its balance queue. A vertex v is removed from the balance queue after the balance step only if $allowed(v) = true$ or if $rebuilt(v) = true$. The loop is iterated until all balance queues are empty.
Again, during the balance step, vertices are balanced using rotations or by rebuilding subtrees. Let δ_1 be an internal vertex of T with right child δ_2, such that after insertion $\beta(\delta_1) < \alpha$. Let δ_2 have right child δ_3. Below we give the conditions, when T_{δ_1} must be rebuilt, when a single rotation should take place at δ_1 and when a double rotation should take place at δ_1. We only treat the case when $\beta(\delta_1) < \alpha$. The case when $\beta(\delta_1) > 1 - \alpha$ is treated symmetrically.

1. A single rotation, which is performed when
   (a) $0.9\alpha \leq \beta(\delta_1) < \alpha$
   (b) $0.9\alpha \leq \beta(\delta_2) \leq \frac{1}{2-\alpha}$
2. A double rotation, which is performed when
   (a) $0.9\alpha \leq \beta(\delta_1) < \alpha$
   (b) $\frac{1}{2-\alpha} < \beta(\delta_2) \leq 1 - 0.9\alpha$
   (c) $0.9\alpha \leq \beta(\delta_3) \leq 1 - 0.9\alpha$
3. Rebuilding T_u, where $u = \delta_1$ or u is an ancestor of δ_1. This happens in all other cases.

The factor 0.9 is rather arbitrary. A factor is needed to assure that the rebuilding of subtrees takes $O(\log p)$ time. Notice that also theorem 3.3 has a value 0.9α in its conditions.
Suppose that at a certain balance step, vertex δ_1 is the first vertex on balance queue Q_i of processor PE_i, $1 \leq i \leq p$.

1. If $allowed(\delta_1) = rebuilt(\delta_1) = false$, nothing happens in that balance step with δ_1. Vertex δ_1 stays at the front of Q_i.

2. If $rebuilt(\delta_1) = true$ then $rebuilt(\delta_2)$ is set to true. The same happens to the *rebuilt* field in the left child of δ_1.

3. If $allowed(\delta_1) = true$ and δ_1 is balanced, the *allowed* fields of the children of δ_1 are set to true.

4. If $allowed(\delta_1) = true$ and δ_1 should be balanced using a single rotation, processor PE_i performs a single rotation at δ_1. The *allowed* fields of δ_2 and the *allowed* fields of the new children of δ_1 (the children of δ_1 after the rotation) are set to true.

5. If $allowed(\delta_1) = true$ and δ_1 should be balanced using a double rotation, processor PE_i performs a double rotation at δ_1. The *allowed* fields of δ_2 and of δ_3 and of the new children of δ_1 (the children of δ_1 after the rotation) are set to true.

6. If $allowed(\delta_1) = true$ and T_{δ_1} should be rebuilt, $rebuilt(\delta_1)$ is set to true. Also the *rebuilt* fields of the children of δ_1 are set to true.

Suppose that during a balance step vertex v is considered, and that $allowed(v) = true$ or $rebuilt(v) = true$. Notice then that the *allowed* or *rebuilt* fields of v's children are set to true. If a rotation has been performed at v, also the *allowed* fields of v's new children are set to true. This fact will be used later on to prove the time complexity of the algorithm. We need the next lemma to prove that no concurrent reads and writes occur during a balance step.

Lemma 4.3 *Let v be a vertex such that during balance step i allowed(v) is set to true. Then all ancestors of v are balanced.*

Proof
The lemma is proved with induction on i. With balance step 0 the initialization is meant.
basis $i = 0$, the lemma is trivially true.
hypothesis Suppose the lemma holds for all balance steps before the i'th balance step.
induction
Let v be an internal vertex in T and suppose $allowed(v)$ is set to true during balance step i. There are five cases why $allowed(v)$ is set to true during balance step i.

1. At the beginning of balance step i, $allowed(parent(v)) = true$ and $parent(v)$ is a balanced vertex. The *allowed* field of $parent(v)$ has been set to true at balance step $i' < i$.
   Using the hypothesis the lemma holds for this case.

2. During balance step i a single left rotation is performed at a vertex w and $v = right(w)$, $v = left(w)$ or $v = left(right(w))$. Using theorem 3.3, notice that w and $right(w)$ are balanced vertices after the rotation. The balances of ancestors of $right(w)$ are balanced since $allowed(w)$ has been set to true after at balance step $i' < i$.

3. During balance step i a single right rotation is performed at a vertex w and $v = left(w)$, $v = right(w)$ or $v = right(left(w))$. Symmetrically to the case above.

4. During balance step i a double left rotation is performed at a vertex w and $v = right(w)$, $v = left(w)$, $v = left(right(w))$ or $v = left(left(right(w)))$. Using theorem 3.3, notice that w and $left(right(w))$ are balanced vertices after the rotation. The balances of ancestors of $right(w)$ are balanced since $allowed(w)$ has been set to true after balance step $i' < i$.

5. During balance step i a double right rotation is performed at a vertex w and $v = left(w)$, $v = right(w)$, $v = right(left(w))$ or $v = right(right(left(w)))$. Symmetrically to the case above.

End Proof

Theorem 4.3 *No concurrent reads and writes occur during the balance part.*

Proof
Suppose v is a vertex with $allowed(v) = true$ and suppose v is the first vertex on Q_i. Furthermore, suppose $allowed(v)$ has been set to true at balance step j.
All ancestors of v where balanced at that time, lemma 4.3.
At the balance step j, when processor PE_i examines v, still all ancestors of v are balanced. Therefore no rotation takes place at an ancestor of v when PE_i examines v.
End Proof

The balance part finishes when the balance queues for all processors are empty and the subtrees have been rebuilt. We now will analyse after how many balance steps the balance queues are empty.
Let vertex δ_1 be the first vertex on balance queue Q_i. When $allowed(\delta_1) = true$ or $rebuilt(\delta_1) = true$, δ_1 will be removed from Q_i after the balance step. Since Q_i contains $O(\log n)$ vertices, after $O(\log n)$ of such balance steps Q_i is empty.
Consider the case that $allowed(\delta_1) = rebuilt(\delta_1) = false$. Then nothing happens with δ_1 during that balance step. Processor PE_i performs a busy wait until $allowed(\delta_1) = true$ or $rebuilt(\delta_1) = true$.
To bound the number of balance steps of this kind, we associate a help queue HQ_i with Q_i. HQ_i contains all the vertices on the search path of PE_i. Vertices on the search path, that are in Q_i, are called marked vertices. Other vertices in HQ_i are unmarked vertices. For analytic purpose, we pretend that PE_i performs balance steps on HQ_i instead of on Q_i. The balance steps PE_i performs on HQ_i are described in figure 11. Instead of performing a busy wait on Q_i, PE_i removes unmarked vertices from HQ_i until there is a marked vertex δ_1 at the front of HQ_i. When this happens $allowed(\delta_1) = true$ or $rebuilt(\delta_1) = true$, as is proved in the next lemma.

while HQ_i is not empty
do Let δ_1 be the first vertex on HQ_i.
  if δ_1 is a marked vertex on HQ_i
  then PE_i treats δ_1 as in figure 10.
  else Suppose δ_1 is an unmarked vertex on HQ_i.
    Then PE_i removes δ_1 from HQ_i.
od

Figure 11: The balance step on HQ_i

Lemma 4.4 *Let HQ_i be the help queue associated with balance queue Q_i, as above. Suppose that during a balance step processor PE_i examines the first vertex v on HQ_i, where v is either marked or unmarked.*
Then $allowed(v) = true$ or $rebuilt(v) = true$.

Proof
Let v be the j'th vertex on HQ_i and suppose that $rebuilt(v) = false$.
If $i = 1$, then the lemma holds trivially.
Let $i > 1$. Consider T before any rotation has taken place. Let $w = parent(v)$; if w is not the root of T, let $u = parent(w)$, and if u is not the root of T, let $u' = parent(u)$. Notice that HQ_i contains u', u, w and v, adjoining and in that order, see figure 12.
Suppose v is considered for the first time in balance step i. Suppose that at the time that w was considered w was still the parent of v. Then $allowed(v)$ was set to true, since by treating w the *allowed* fields of all w's children are set to true.
Suppose that when w was considered, v was not anymore a child of w. Then this has happened by one of the following cases.

1. A double rotation has been performed at u' and $u = right(u')$ and $w = left(u)$ or $u = left(u')$ and $w = right(u)$.
   If after the rotation, v is a child of u', then $allowed(v)$ is set during that balance step.
   If after the rotation, v is a child of u, then $allowed(v)$ will be set in the next balance step, when u is treated.

2. A double rotation has been performed at u.
   Then v becomes the new parent of u and $allowed(v)$ is set to true.

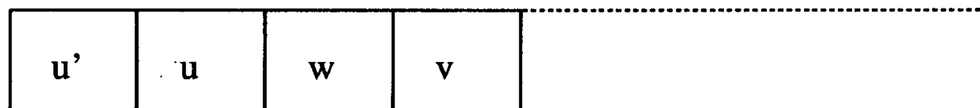


Figure 12: The order of vertices on HQ_i

3. A single rotation has been performed at u.
   Then v becomes the new child of u and *allowed*(v) is set to true.

End Proof

Theorem 4.4 *After $O(\log n)$ balance steps the balance queues of all processors are empty. All vertices v with rebuilt(v)=false are balanced.*

Proof
From help queue HQ_i associated with balance queue Q_i, after each balance step a vertex (either marked or unmarked) is removed, see also lemma 4.4. Since HQ_i contains $O(\log n)$ marked and unmarked vertices, after $O(\log n)$ balance steps HQ_i, and thus Q_i, are empty.
Use theorem 3.3 to see that all vertices v with *rebuilt*(v) $= false$, that are removed from a balance queue, are balanced.
End Proof

Rebuilding a subtree T_v is done by all processors that have inserted a new element in T_v. It is easy to see that enough elements have been inserted in T_v to rebuild T_v in $O(\log p)$ time. Notice that just before rebuilding, v has a balance less than 0.9α. Before insertion, v had a balance greater than or equal to α.
Let T'_v be the subtree before rotations have been performed at T. Let δ_1, δ_2 and δ_3 be as before. Suppose $v = \delta_2$. If $T'(v) = T(v)$, enough new elements have been inserted in $T(v)$, since $\beta(v) < 0.9\alpha$. There are three reasons why $T(v)$ and $T'(v)$ can differ. Only the last case is of interest, since only then $T(v)$ will be rebuilt.

1. Suppose a single rotation has been performed at δ_1. By theorem 3.3, in that case v is a balanced vertex.

2. Suppose a double rotation has been performed at *parent*(δ_1). By theorem 3.3, in that case v is a balanced vertex.

3. Suppose a double rotation has been performed at δ_1. In the next theorem it is proved that then enough elements have been inserted in T_v to rebuild T_v in $O(\log p)$ time.

Theorem 4.5 *Suppose T_v must be rebuilt. Suppose T_v is rebuilt using the processors that have inserted an element in T_v. Then enough processors have inserted an element in T_v to rebuild T_v in $O(\log p)$ time.*

Proof
Let T'_v be as before. If $T'_v = T_v$ the theorem is trivial.
If $T'_v \neq T_v$, theorem 3.3 tells us that v in T'_v has a balance less than 0.9α. Thus in that

case the theorem is also trivial.
End Proof

Notice that only rotations performed at vertices on the leftmost path, cause an extra vertex to appear on the leftmost path. So with every c consecutive vertices on the leftmost path a processor can be associated on $O(\log p)$ time, see result 3.
The time complexity of the second insert algorithm is stated in the following theorem.

Theorem 4.6 *The second parallel insert algorithm as described in this section runs in $O(\log n + \log p)$ using p processors.*

Proof
The search part takes again $O(\log n + \log p)$ time using p processors.
The insert part takes $O(\log p)$ time.
Since every vertex on a balance queue is processed for $O(1)$ time theorem 4.4 implies that the balance part takes $O(\log n + \log p)$ time.
End Proof

Theorem 4.7 *Let PQ be a priority queue containing n elements.*
Then p processors of an EREW PRAM can perform a ParInsert operation on PQ in $O(\log n)$ time.

bf Proof
Use the first or second parallel insert algorithm.
bf End Proof

4.5 ParDecreasekey

Input to the *ParDecreasekey* algorithm is the weight-balanced tree T and the tuples $\langle q_1, pr_1\rangle, \ldots, \langle q_p, pr_p\rangle$, where q_i is a pointer to leaf l_i in T and pr_i is a key value, $1 \le i \le p$. The element e_i in l_i will get new key value pr_i.
It is easy to perform *ParDecreasekey* in $O(\log n + \log p)$ time using p processors by first deleting the leaves l_i, $1 \le i \le p$, using the ParDelete algorithm described below and then to insert the elements e_i with new key value k_i, $1 \le i \le p$, using a parallel insert algorithm.

4.5.1 ParDelete

Input to the *ParDelete* algorithm are the pointers $q_1, \ldots, q_p$, and weight-balanced tree T. Processor PE_i will delete the element in the leaf to which q_i points, $1 \le i \le p$. The *ParDelete* algorithm consists of two steps.

1. The Delete Part
   Let pointer q_i point to l_i. Then processor PE_i must delete l_i and every ancestor a_i of l_i that has at most one son, after deleting all elements in T_{a_i}. To delete these vertices on the tree path TP from l_i to the root of T, PE_i walks up along TP using the third walking up technique.
   During the walking up PE_i maintains the pointer q_i'. Suppose that at a certain moment PE_i is associated with a vertex v. Then if q_i' has value nil, all elements in T_v are deleted. If q_i' points to a vertex w, all elements in T_v that are not deleted are stored in T_w. The vertices on TP with one child are deleted by replacing such a vertex with its only child. Vertices on TP that have no children are deleted. During the delete part the balance information in vertices is updated.

2. The Balance Part
   The balance part of the *ParDelete* algorithm is analogous to the balance part of the algorithm ParInsert1.

Theorem 4.8 *The ParDelete algorithm runs in* $O(\log n)$ *time.*

Proof
By the above discussion.
End Proof

Using the ParDelete algorithm as described above, the time complexity of the ParDecreasekey algorithm can be bounded by $O(\log n + \log p)$.

Theorem 4.9 *Let PQ be a priority queue containing n elements.*
Then p processors of an EREW PRAM can perform a ParDecreasekey operation on PQ in $O(\log n)$ *time.*

Proof
Clear by the above discussion.
End Proof

References

[BM80] N. Blum and K. Melhorn. On the avereage number of rebalancing operations in weight-balanced trees. *Theoret. Comput. Sci.*, 10:303–320, 1980.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[Col88] R. Cole. Parallel merge sort. *J. Comp. System Sci.*, 17:770–785, 1988.

[DGST88] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: An alternative to fibonnacci heaps with application to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[Ove83] M.H. Overmars. The design of dynamic data structures. *Springer Lecture Notes in Computer Science*, 156, 1983.

[PP91] M.C. Pinotti and G. Pucci. Parallel priority queues. *Inform. Process. Lett.*, 40:33–40, 1991.

[PVW83] W. Paul, U. Vishkin, and H. Wagener. Parallel computation on 2-3 trees. *Springer Lecture Notes in Computer Science*, 154:579–609, 1983.

[ST86] D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15:52–59, 1986.

[SV82] Y. Shiloach and U. Vishkin. An $O(n^2 log n)$ Parallel Max-Flow Algorithm. *Jrnl. of Algorithms*, 3:128–146, 1982.

[Vel87] M. Veldhorst. Linked allocation for parallel data structures. Technical Report RUU-CS-87-18, University of Utrecht, Dep. of Computer Science, October 1987.