# Parallel Priority Queues*

Maria Cristina PINOTTI† and Geppino PUCCI‡

## Abstract

This paper introduces the Parallel Priority Queue (PPQ) abstract data type. A PPQ stores a set of integer-valued items and provides operations such as insertion of $n$ new items or deletion of the $n$ smallest ones. Algorithms for realizing PPQ operations on an $n$-processor CREW-PRAM are based on two new data structures, the $n$-Bandwidth-Heap ($n$-H) and the $n$-Bandwidth-Leftist-Heap ($n$-L), that are obtained as extensions of the well known sequential binary-heap and leftist-heap, respectively. Using these structures, it is shown that insertion of $n$ new items in a PPQ of $m$ elements can be performed in parallel time $O(h + \log n)$, where $h = \log \frac{m}{n}$, while deletion of the $n$ smallest items can be performed in time $O(h + \log \log n)$.

**Keywords** Data structures, parallel algorithms, analysis of algorithms, heaps, PRAM model.

*This work has been partly supported by the Ministero della Pubblica Istruzione of Italy and by the C.N.R. project "Sistemi Informatici e Calcolo Parallelo"

†Istituto di Elaborazione dell'Informazione, C.N.R., Pisa, Italy

‡Dipartimento di Informatica, Università di Pisa, Pisa, Italy and International Computer Science Institute, Berkeley, CA, USA

# 1 Introduction

A *Priority Queue* is an abstract data type used in a wide variety of algorithms, for storing a set of labeled items and selecting the one associated with the smallest label [1]. In this note we propose a new data type, that we call the *Parallel Priority Queue* (PPQ) and devise two PPQ implementations for the CREW-PRAM. In this model of computation $n$ RAM processors $P_1, \ldots, P_n$ have concurrent unit time access to a shared memory, with the provision that two or more processors may read a cell simultaneously but concurrent write accesses to the same cell are prohibited [6].

We require that a PPQ allows efficient simultaneous insertions of $n$ new items, one for each processor, or the removal of the $n$ items associated with the $n$ smallest labels. A PPQ can be useful for the parallel implementation of techniques, such as *Branch-and-Bound* [9], which imply the solution of several subproblems, each associated with a different cost. By using a PPQ, at each stage the processors may efficiently select the $n$ "more promising" subproblems to be solved.

Formally, a PPQ $Q$ is an abstract data type consisting of a collection of (possibly replicated) items with associated integer-valued labels. Three operations are defined, namely:

- *Insert*$(< i_1, \ldots, i_n >, Q)$ for the insertion of items $i_1, \ldots, i_n$ in $Q$.
- *Deletemin*$(Q, n)$ for the deletion and return of the $n$ smallest labelled items.
- *Makequeue*$(S, Q)$ for the construction of $Q$ with the items of a set $S$.

An additional operation is provided by *meldable* PPQs, namely:

- *Meld*$(Q_1, Q_2, Q)$ for the combination of PPQs $Q_1$ and $Q_2$ into $Q$.

There have been several papers in the literature concerning the implementation of priority queues in a parallel environment [2, 3, 7, 10]. All of the above works are based on traditional heap structures and achieve only a limited degree of parallelism. The two PPQ implementations devised in this paper make use of two new data structures, the $n$-Bandwidth-Heap ($n$-H), for unmeldable PPQs, and the $n$-Bandwidth-Leftist-Heap ($n$-L), for meldable PPQs. These structures are obtained as extensions of the well known *binary heap* [5] and *leftist heap* [11] sequential structures, respectively. We develop efficient PRAM algorithms for all the above PPQ opera-

tions on $n$-H and $n$-L. Let $m$ be the number of items stored in an $n$-H (resp., $n$-L) and let $h = \log \frac{m}{n}$. In the next sections we show that on an $n$ processor CREW-PRAM:

- $n$ items can be inserted in an $n$-H (resp., $n$-L) in time $O(h + \log n)$.
- The $n$ smallest items stored in an $n$-H (resp., $n$-L) can be deleted in time $O(h + \log\log n)$.
- An $n$-H (resp., $n$-L) can be constructed from a set $S$ of $m$ items in time $O(\frac{m}{n} \log n)$.
- Two $n$-L structures of $m_1$ and $m_2$ elements can be melded in time $O\left(h_1 + h_2 + \log\log n\right)$, where $h_1 = \log \frac{m_1}{n}$ and $h_2 = \log \frac{m_2}{n}$.

## 2 Background and Definitions

Let us briefly review some of the properties of the mentioned sequential structures. Both of them are based on *heap-ordered* binary trees, that is, trees where the label of each node is smaller than or equal to the labels of its children. A binary heap is a complete binary tree, where some rightmost nodes of the last level may be absent. Hence, a binary heap has a logarithmic depth and can be stored in an array, with positions $2i$ and $2i+1$ containing the left and right children of node $i$ [5].

Recall that the *rank* of a node $x$ of a binary tree is defined as:

$$\text{rank}(x) = \begin{cases} 0 & \text{if } x \text{ is void} \\ 1 & \text{if } x \text{ is a leaf} \\ 1 + \min\{\text{rank}(\text{left}(x)), \text{rank}(\text{right}(x))\} & \text{otherwise} \end{cases}$$

where $\text{left}(x)$ and $\text{right}(x)$ denote the left and right children of $x$. A leftist heap is a heap-ordered *leftist tree*, that is, a binary tree such that, for each node $x$, $\text{rank}(\text{left}(x)) \geq \text{rank}(\text{right}(x))$. It can be shown that, in a leftist tree, the path from the root to the rightmost leaf is one of the shortest paths in the tree (i.e., containing the least number of nodes), and that this path has at most logarithmic length in the number of nodes. Note that, unlike binary heaps, a leftist heap must be stored as a linked structure, with the ranks explicitly stored with the nodes.

It is not clear how to use binary or leftist heaps for an efficient parallel implementation of PPQ operations. In particular, the fast retrieval of the $n$ smallest items, needed by *Deletemin*, seems to be difficult on these structures. To find the $n$ minima in $O(1)$ parallel time, we will store a set of $n$ items at each node of the structure, so that the maximum label in the set is equal to or smaller than all the labels stored at the node's descendants. Under this organization, that we call *extended heap order*, the $n$ smallest values are found in the root.

Formally, we define an *$n$-Bandwidth-Heap* ($n$-H) and an *$n$-Bandwidth-Leftist-Heap* ($n$-L), respectively, as a binary heap and a leftist heap, whose nodes, each containing $n$ items, are arranged in extended heap order. Letting $m = f(n) \cdot n$ be the number of items stored in the heap, with $f(n)$ being any increasing integer function of $n$, we denote the height of an $n$-H, or the length of the rightmost path of an $n$-L, by $h \in O(\log \frac{m}{n})$. The $n$-bandwidth heaps can be realized like their sequential counterparts, with the only difference that each node contains a vector of $n$ items. For implementation purposes, we require that the items at a node appear in sorted order.

# 3 PPQ Operations

Let us present the algorithms to realize PPQ operations on $n$-H and $n$-L. The building blocks of such algorithms are the procedures PARALLEL-SORT($S$) and PARALLEL-MERGE($E_1, E_2$), where $S$ denotes a set of integer items to be sorted and $E_1, E_2$ are two ordered sequences to be merged. Sorting and merging are employed to efficiently establish and preserve the extended heap order during the heap operations. Recall that, with an $n$-processor CREW-PRAM, $n$ items can be sorted in time [4]

$$T_S \in \Theta(\log n) \tag{1}$$

while merging two vectors of cardinality $k_1$ and $k_2$ can be done in time [8]

$$T_M \in \Theta\left(\frac{k_1 + k_2}{n} + \log\log n\right) \tag{2}$$

As a particular case of relation 2, we have that two vectors of size $n$ can be merged in time $O(\log\log n)$ using $n$ processors.

## 3.1 PPQ Operations on $n$-H

Let $E_P$ denote the ordered sequence of $n$ items stored in node $P$ of an $n$-H $Q$, and let $\max(E_P)$ denote the maximum element of $E_P$. Moreover, a *path* $\pi = P_1, \ldots, P_k$ is a sequence of nodes of $Q$ from the root $P_1$ to a leaf $P_k$, and, for a given path $\pi$, $E_\pi$ denotes the concatenation $E_{P_1} E_{P_2} \ldots E_{P_k}$. Note that $E_\pi$ is an ordered sequence. However, when we perform insertions or deletions, we will replace the set stored in $P_1$ or $P_k$ by another ordered set, which may violate the overall order on $E_\pi$. To reorder $E_\pi$ after such a replacement, we define the following procedure on a path $\pi$:

**procedure** REARRANGE( $\pi$, $P \in \{P_1, P_k\}$):
**begin**
$E :=$ PARALLEL-MERGE($E_P$, $E_{\pi - P}$);
Let $E^1, \ldots, E^{|\pi|}$ be the $|\pi|$ consecutive subsequences of $E$ of cardinality $n$;
**for** $i := 1$ to $|\pi|$ **do** $E_{P_i} := E^i$ **endfor**
**end**.

Since the copy of the $E^i$'s into the $E_{P_i}$'s takes parallel time $O(|\pi|)$, the parallel complexity $C_R$ of REARRANGE is dominated by the time required by PARALLEL-MERGE for two sequences of length $n$ and $n(|\pi| - 1)$ with $n$ processors. From 2 we have:

$$C_R \in O(|\pi| + \log\log n) \tag{3}$$

**Insertion.** To insert $n$ new items into an $n$-H $Q$, we first place them, in sorted order, in the leftmost vacant leaf $V_L$ of $Q$; then we rearrange the path $\pi$ from the root of $Q$ to $V_L$ with REARRANGE($\pi, V_L$). It is crucial to note that this operation preserves the extended heap order. In fact, for each node $P_i \in \pi$, $\max(E_{P_i})$ is not incremented by REARRANGE. Thus, the minimum elements stored in the children of $P_i$ are both greater than $\max(E_{P_i})$. A simple program for insertion is the following:

**program** INSERT($\{i_1, \ldots, i_n\}, Q$):
**begin**
Let $V_L$ be the leftmost vacant leaf of $Q$;
$E_{V_L} :=$ PARALLEL-SORT($\{i_1, \ldots, i_n\}$);
Determine the path $\pi$ from the root to $V_L$;
REARRANGE($\pi$, $V_L$)
**end**.

Since $|\pi| = h$, $\pi$ can be determined in time $O(h)$. Hence the time complexity $C_I$ of INSERT is dominated by the initial sorting phase and by the complexity of REARRANGE. From 1 and 3 it easily follows that:

$$C_I \in O(h + \log n) \tag{4}$$

**Deletion.** Let a *minimum path* $\mu$ of $Q$ be recursively defined as:

1. The root of $Q$ belongs to $\mu$.
2. Let a non leaf node $P$ belong to $\mu$. If $X$ and $Y$ are the children of $P$ and either $Y$ is void or $\max(E_X) \leq \max(E_Y)$, then $X$ belongs to $\mu$.

We define a procedure ADJUST on a minimum path $\mu$:

**procedure** ADJUST($\mu$):
**begin**
**foreach** $P \in \mu$ **do**
**if** Sibling($x$) exists **then**
$E :=$ PARALLEL-MERGE($E_P$, $E_{\text{Sibling}(P)}$);
Let $E^1$, $E^2$ be the left and right halves of $E$;
$E_P := E^1$;
$E_{\text{Sibling}(P)} := E^2$
**endif**
**endfor**
**end**.

Note that ADJUST does not violate the extended heap order. Furthermore, this procedure rearranges the elements stored in the nodes along $\mu$ and their siblings so that we can "shift" $\mu$ up of one position and still preserve the extended heap order. This feature of ADJUST will be used in the implementation of the DELETEMIN operation. A straightforward implementation of ADJUST on the PRAM takes time $O(h \log\log n)$. However, a better time complexity is achieved by assigning $\max\left\{\left\lfloor \frac{n}{h} \right\rfloor, 1\right\}$ processors to each node of $\mu$ to perform the PARALLEL-MERGE operation between that node and its sibling. The algorithm will then consist of $\left\lceil \frac{h}{n} \right\rceil$ phases, each performing $\min\{h, n\}$ instances of PARALLEL-MERGE on distinct pairs of nodes. Since each phase requires $O(\min\{h, n\} + \log\log n)$ time (from 2), ADJUST can be realized in total time:

$$C_A \in O(h + \log\log n) \tag{5}$$

Let now $R$ and $F_R$ respectively denote the root and the rightmost non vacant leaf of $Q$ at height $h$. Deletion is performed by the program DELETEMIN given below. Recalling that the set $E_R$ contains the $n$ smallest elements of $Q$, at first DELETEMIN returns $E_R$, and replaces the set in $R$ by the elements in $F_R$. Then ADJUST and REARRANGE are called on the minimum path $\mu$ of $Q$ to re-establish the extended heap order. Note that in the ordered sequence $E_\mu$, created by REARRANGE, some elements previously residing on $P_i \in \mu$ may now be assigned to its father $P_{i-1}$. However, the extended heap order is not violated as ADJUST has modified $\mu$ so that $\max(E_{P_i})$ (hence all the values of $E_{P_i}$) become smaller than the minimum value of $E_{\text{Sibling}(P_i)}$.

**program** DELETEMIN( $Q, n$):
**begin**
Let $R, F_R$ be the the root and the rightmost non vacant leaf of $Q$;
**return** $E_R$;
$E_R := E_{F_R}$;

```
        Determine the minimum-path μ ;
        ADJUST(μ);
        REARRANGE(μ, R)
    end.
```

The time complexity $C_D$ of Deletemin is dominated by the execution of REARRANGE and ADJUST, since the other operations take altogether $O(h)$ time. Hence:

$$C_D \in O(h + \log\log n) \tag{6}$$

Note that, for $h \geq \log\log n$ (i.e., $m \geq n \log n$) DELETEMIN takes time proportional to the $n$-H height. The slowdown, for the case $h < \log\log n$ is due to the merge algorithm, which cannot attain linear speedup if the number of processors is close to the number of elements to be merged [8].

**Construction.** We are finally left with implementing $n$-H construction. Let $S[1, \ldots, n]$ be the set of $n$ elements to be stored in the new heap $Q$. We first build an $n$-H tree whose nodes contain sorted items. These nodes do not necessarily satisfy the extended heap order, which is subsequently created, level by level, starting from the leaves . Specifically, for each subtree $T$ at level $k$, its minimum-path $\mu_T$ is determined, adjusted and rearranged. Construction is performed by the following program:

```
program MAKEQUEUE(S, Q):
    begin
        for i := 1 to m/n − 1 do {⋆⋆ sorting phase ⋆⋆}
            Q[in + 1, ..., (i + 1)n] := PARALLEL-SORT(S[in + 1, ..., (i + 1)n])
        endfor;
        for k := h downto 1 do {⋆⋆ heapify phase ⋆⋆}
            for each node P at level k do
                Let T be the tree rooted at P;
                Determine the minimum-path μ_T;
                ADJUST(μ_T);
                REARRANGE(μ_T, P)
            endfor
        endfor
    end.
```

The correctness of the above algorithm can be easily proved by induction on the height of the $n$-H. As to its time complexity, from 1 it follows that the sorting phase can be accomplished in parallel time $O(\frac{m}{n} \log n)$. During the heapify phase, at each node $P$ of level $k$, all processors operate in parallel to re-establish the extended heap order in the subtree rooted at $P$. Hence,

each step of the inner loop takes time $O(h - k + \log\log n)$ for a total time of

$$O\left(\sum_{j=1}^{h} \frac{m}{n2^j}(j + \log\log n)\right) = O\left(\frac{m}{n}\log\log n\right)$$

Therefore, the complexity $C_{MQ}$ of MAKEQUEUE is altogether:

$$C_{MQ} \in O\left(\frac{m}{n}\log n\right) \tag{7}$$

### 3.2 PPQ Operations on $n$-L

Due to their vectorial representation, $n$-H cannot efficiently realize meldable PPQs. Melding is instead the basic operation for $n$-L, since it provides the basis for all the other PPQ operations of insertion, deletion and heap construction.

**Melding** To meld two $n$-L, $Q_1$ and $Q_2$, into an $n$-L $Q$, we combine their paths $\rho_1$ and $\rho_2$ from the roots to the rightmost leaves into a single path $\rho$ of length $h = h_1 + h_2$, where $h_1 = |\rho_1|$ and $h_2 = |\rho_2|$, $h_1 \leq h_2$. The remaining nodes of $Q_1$ and $Q_2$ are then attached to this path opportunely. Finally rank readjustment is performed, to guarantee the leftist property of Q.

We represent a node $P$ of an $n$-L by a record $P = (E,$ left, right, rank), where $P.E$ is the sorted sequence of the $n$ values stored in $P$ (denoted by $E_P$ in section 3.1); $P$.left and $P$.right are the pointers to the left and right children of $P$, respectively; and $P$.rank is the rank of $P$. Furthermore, for a node $P$, we define $L(P)$ as the pair ($P$.left, max($P.E$)). As before, we denote by $E_{\rho_i}$, with $i = 1, 2$, the concatenation of the lists $P.E$ stored at the nodes of path $\rho_i$. Similarly, we define $L_{\rho_i}$ to be the sequence of the pairs relative to the nodes of $\rho_i$. We say that $L_{\rho_i}$ is *ordered*, because the second components of its pairs appear in non decreasing order. Melding is performed by the program MELD given below. (Note the use of the Pascal-like notation new($P$) and $P\uparrow$, to denote a new record to be created, and its fields).

**program** MELD($Q_1$, $Q_2$, $Q$):
**begin**
Let $\rho_1$ and $\rho_2$ be the rightmost paths of $Q_1$ and $Q_2$, respectively;
Let $h_1 = |\rho_1|$ and $h_2 = |\rho_2|$;
$E :=$ PARALLEL-MERGE($E_{\rho_1}$, $E_{\rho_2}$);
$L :=$ PARALLEL-MERGE($L_{\rho_1}$, $L_{\rho_2}$);
{$\star\star$ The merge of $L_{\rho_1}$ and $L_{\rho_2}$ is based on the total order
defined over the second field of the pairs $\star\star$}
Let $E^1, \ldots, E^{h_1+h_2}$ be the $(h_1 + h_2)$ consecutive subsequences of $E$ of cardinality $n$;
Let $L^1, \ldots, L^{h_1+h_2}$ be the $(h_1 + h_2)$ first components of the pairs in $L$;

```
    h := h1 + h2;
    new(P); P↑:= (E^h, L^h, nil, 1);
    for i := h − 1 downto 1 do
        N := (E^i, L^i, P, 1);
        if (N.right↑).rank > (N.left↑).rank
          then swap(N.left, N.right)
        endif
        N.rank := (N.right↑).rank +1;
        new(P); P↑:= N
    endfor;
    Q := P
end.
```

Note that the **for** loop in MELD performs both the path construction and the rank readjustment phases. To show that the above algorithm is correct, we observe that the construction of the rightmost path $\rho$ guarantees the extended heap order as, for each left subtree $T$ pointed at by a node $P$ in $\rho$, the values in $P.E$ can only be less or equal to the ones in the node of $\rho_1$ or $\rho_2$ that originally pointed to $T$. As to the time complexity $C_M$ of MELD, it can be easily seen that it is determined by the PARALLEL-MERGE step needed to create $E$ and $L$, (all the other phases are performed in $O(h_1 + h_2)$ time). From 2 we have:

$$C_M \in O(h_1 + h_2 + \log\log n) = O(h + \log\log n) \tag{8}$$

Hence, for $h_1 + h_2 \in \Omega(\log\log n)$ the meld operation takes time proportional to the height of the resulting tree.

All the other operations on $n$-L heaps are based on melding. To insert $n$ new items into an $n$-L $Q$, we make them into a one-node $n$-L heap and meld it with $Q$. To delete the $n$ minima from $Q$, we remove its root and meld the remaining left and right subtrees, which are in turn $n$-L heaps. Finally, the $n$-L construction can be realized by first building a list $\mathcal{L}$ of $\frac{m}{n}$ one-node heaps, and then iteratively melding the elements of $\mathcal{L}$ until only one remains. From easy calculations, it follows that the time complexities $C_I$, $C_D$ and $C_{MQ}$ of these three operations are exactly as in the $n$-H case.

# 4 Conclusions

The PPQ data type introduced in this paper is based on the idea of extending the sequential Priority Queue structures to a parallel context. Adopting the CREW-PRAM model of parallel computation, we have defined two PPQ data structures, the $n$-H and the $n$-L, respectively

realizing unmeldable and meldable queues. We have then employed optimal algorithms for sorting and merging devised for the CREW-PRAM for the efficient implementation of the basic operations of insert, deletemin, makequeue and meld.

For a better assessment of the efficiency of our structures, consider a parallel version of *Heapsort*, made of a Makequeue followed by $\frac{m}{n}$ Deletemin operations, where $m$ and $n$ are the elements to be sorted and the number of available processors, respectively. Letting $C_{HS}(m,n)$ be the time complexity of such parallel Heapsort, from 6 and 7 we have:

$$C_{HS}(m,n) \in O\left(\frac{m}{n}\log n + \sum_{j=1}^{\log\frac{m}{n}} \frac{m}{n2^j}(\log\frac{m}{n} - j + \log\log n)\right) = O\left(\frac{m}{n}\log m\right)$$

which is clearly optimal for any value of $m$ and $n$, with $m \geq n$. The optimality of the above algorithm is due to the use of Cole's $O(\log n)$ complex parallel sorting. However, for $m > n\log\log n$, we can slightly adapt the sorting phase of makequeue to use simpler sorting algorithms [8] without increasing the overall running time.

Finally, we want to point out that the extended heap order, as defined in section 2, does not require that the elements at each node be sorted. We are currently investigating the techniques needed to implement PPQ operations without making use of sorting but still achieving the same time complexities.

**Acknowledgements** We thank Fabrizio Luccio and Andrea Pietracaprina for helpful discussions concerning this work. We also thank an anonymous referee for a number of excellent suggestions and for pointing out reference [8].

## References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

[2] F. Bauernöppel and H. Jung. Implementing Abstract Data Structures in Hardware. In *Proc. 13th Conf. on Mathematical Foundations of Computer Science*, LNCS 324, Springer, Berlin, Germany, 1988, 172-179.

[3] J. Biswas and J.C. Browne. Simultaneous Update of Priority Structures. In *Proc. of the 1987 Int. Conf. on Parallel Processing*, 1987, 124-131.

[4] R. Cole. Parallel Merge Sort. *SIAM Journal of Computing*, **17**(4), 1988, 130-145.

[5] R.W. Floyd. Algorithm 245: Treesort. *Communications of the ACM*, **7**(12), 1964, 701.

[6] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proc. 10th Annual ACM Symp. on Theory of Computing*, 1978, 114-118.

[7] D.W. Jones. Concurrent Operations on Priority Queues. *Communications of the ACM*, **32**(1), 1989, 132-137.

[8] C.P. Kruskal. Searching, Merging and Sorting in Parallel Computation. *IEEE Transactions on Computers*, **C-32**(10), 1983, 942-946.

[9] R. Karp, M. Saks and A. Wigderson. On a Search Problem Related to Branch-and-Bound Procedures. In *Proc. 27th Symp. on Foundations of Computer Science*, 1986, 19-28.

[10] V.N. Rao and V. Kumar. Concurrent Access of Priority Queues. *IEEE Trans. on Computers*, **C-37**(12), 1988, 1657-1665.

[11] R.E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, Penn., 1983.