Parallel Priority Queues

Maria Cristina Pinotti and Geppino Pucci

Progetto finalizzato
"Sistemi Informatici e Calcolo Parallelo"

Nota interna B4-29

Luglio 1990

Parallel Priority Queues

Maria Cristina Pinotti† and Geppino Pucci‡

†Istituto di Elaborazione dell'Informazione, C. N. R., Pisa, Italy.
‡Dipartimento di Informatica, Università di Pisa, Pisa, Italy.

Abstract

A Parallel Priority Queue (PPQ) is defined as an abstract data type for storing a set of integer-valued items and providing operations such as insertion of n new items or deletion of the n smallest ones. In this paper, the PPQ data type is implemented on the PRAM model of parallel computation. Two data structures are introduced, the n-Bandwidth-Heap (nH) and the n-Bandwidth-Leftist-Heap (nL), based on the well known sequential binary-heap and leftist-heap organizations, respectively. Efficient parallel algorithms are then given for the basic operations of insertion, deletion and heap construction on nH and nL, based on known parallel sorting and merging algorithms.

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

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

1. Introduction

A *Priority Queue* (PQ) is an abstract data type used in a wide variety of algorithms, for storing a set of labelled items and selecting the one associated with the smallest label [1]. In this note, we propose two Parallel implementations of a Priority Queue (PPQ) on the CREW-PRAM. In this model of computation n RAM processors P_1, ..., 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 [4].

We require that a PPQ allow 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 those techniques, such as Branch-and-Bound, which imply the solution of several subproblems, each associated with a different cost [5]. 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:

1) *Insert* ($<i_1, \ldots, i_n>, Q$) for the insertion of items i_1, ..., i_n in Q;

2) *Deletemin* (Q) for the deletion and return of the n smallest labelled items;

3) *Makequeue* (S, Q) for the construction of Q with the items of a set S.

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

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

In this paper, we devise two PPQ implementations, based on the sequential structures known as *binary heap* [8] and *leftist heap* [6]. The first implementation, called *n-Bandwidth-Heap* (nH), is unmeldable. The second, called *n-Bandwidth-Leftist-Heap* (nL) is instead meldable.

2. Background and definitions

First, let us briefly review some of the properties of the mentioned sequential data 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, binary heaps keep a logarithmic depth and can be stored in an array, where the positions 2·i and 2·i+1 contain the left and right children of node i.

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

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

where left(x) and 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, rank (left(x)) ≥ rank (right(x)). It can be shown that, in a leftist tree, the shortest path from the root to a leaf is the rightmost one in the tree, 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 rank explicitly stored with the nodes.

In a parallel context, the above structures seem not to be suitable for an efficient realization of PPQ operations. For instance, using a binary heap or a leftist heap, the n^{th} smallest value, needed by Deletemin, would be available after $\Omega(n)$ time, as the position of the i^{th} smallest item depends on the position of the previous i-1 smallest ones. To find the n smallest elements in parallel, in O(1) 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* (nH) and an *n-Bandwidth-Leftist-Heap* (nL) respectively as a binary heap and a leftist heap, whose nodes, containing n items, are arranged in extended heap order. Letting m = f(n)·n be the total number of items stored in the heap, we denote the height of an nH, or the length of the rightmost path of an nL by $h \in O\left(\log\frac{m}{n}\right)$. The 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 the PPQ operations on nH and nL. The building blocks of such algorithms are PARALLEL-SORT and PARALLEL-MERGE that are employed to efficiently establish and preserve the extended heap order during the operations. Recall that, with an n-processor CREW-PRAM, n items can be sorted in time [3]:

$$T_{Sort} = \Theta (\log n); \quad (1)$$

while merging two vectors of cardinality k_1 and k_2 can be accomplished in time [7]:

$$T_{Merge} = \Theta (\min \{(k_1+k_2)/n + \log(k_1k_2\log n/n), (k_1+k_2) (\log\log (k_1+k_2))/n\}). \quad (2)$$

As a particular case of relation (2), we have that two vectors of size n can be merged in time O (loglog n) by using n processors, while the same operation can instead be performed in time O (log n) if only n/log n processors are available.

3.1 PPQ operations on nH

Let E_P denote the ordered sequence of n items stored in a node P of an nH and 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 heap Q from the root P_1 to a leaf P_k, and for a given path π, E_π denotes the concatenation $E_{P1} E_{P2} \ldots E_{Pk}$. Note that E_π is an ordered sequence. However, when we inserte or delete a node, we will replace the set stored in P_1 or P_k with another ordered set, which may violate the overall order of E_π. To re-order E_π after such replacement, we define the following procedure on π:

procedure REARRANGE (π, 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 with cardinality n;
  for i := 1 to $|\pi|$ **do** $E_{Pi} := E^i$ **endfor**;
end.

The time complexity C_R of REARRANGE is dominated by the time of PARALLEL-MERGE between two sequences of length n($|\pi|$ - 1) and n. We have from (2):

$$C_R \in O (\min \{|\pi|+\log n, |\pi|\log\log n.\}). \quad (3)$$

Insertion. To insert n new items into an nH, say Q, we first place them, in sorted order, in the leftmost vacant leaf L of Q; then we rearrange the path π from the root of Q to L with REARRANGE (π, 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. Then, 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 ({i1, . . . ,in}, Q):
    begin
      Let L be the leftmost vacant leaf of Q;
      EL := PARALLEL-SORT ({i1, . . . ,in});
      Build the path π from root to L;
      REARRANGE (π, L);
    end.
```

Since $|\pi| = h$, π can be built in time O(h). Hence, the time complexity C_I of INSERT is determined 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* μ of Q be recursively defined as:

1) the root of Q belongs to μ;

2) if a non leaf node P belongs to μ, then X belongs to μ, such that X, Y are the children of P, and $\max(E_X) \leq \max(E_Y)$, or Y is void.

We define a new procedure ADJUST on μ:

```
procedure ADJUST (μ):
    begin
     for each P∈ μ do
       if Brother(P) exists
          then    E := PARALLEL-MERGE (EP, EBrother(P));
                  Let E1, E2 be the left and right halves of E;
                  EP := E1;
                  EBrother(P) := E2;
       endif
     endfor
    end.
```

Note that ADJUST does not violate the extended heap order. A straightforward implementation of ADJUST on the PRAM takes time O (h loglog n). However, for

$h > (\log n)/\log\log n$, a better time complexity is achieved by assigning $n/\log n$ processors to each node of μ to perform the PARALLEL-MERGE operation between that node and its brother. The algorithm will then consist of $\lceil h/\log n \rceil$ phases, each one relative to at most log n nodes of μ. Each phase takes O(log n) time (from (2)). Therefore, ADJUST can be realized in time:

$$C_A \in O(\min\{h \log\log n, \max\{h, \log n\}\}). \quad (5)$$

Let now R and L denote, respectively, the root and the rightmost non vacant leaf of Q. Deletion is performed by the program DELETEMIN reported below. Recalling that the set E_R contains the n smallest elements of Q to be deleted, at first deletion returns E_R, and replaces the set in R with the elements in L. Then ADJUST and REARRANGE are called on the minimum path μ of Q to re-establish the extended heap order. Note that in the ordered sequence E_μ, 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 μ so that $\max(E_{P_i})$ (hence all the values of E_{P_i}) becomes smaller than the minimum value of Brother(P_i).

```
program DELETEMIN (Q):
    begin
      Let R, L be the root and the rightmost non vacant leaf of Q;
      Return E_R;
      E_R := E_L;
      Build 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:

$$C_D \in O(\min\{h+\log n, h \log\log n\}). \quad (6)$$

Note that, for $h \geq \log n$ (i.e., $m \geq n^2$) Deletemin takes time proportional to the nH height. The slowdown, for the case $h < \log n$ is due to the PARALLEL-MERGE algorithm, which cannot reach unit efficiency if the number of processors is "close" to the number of elements to be merged [2].

Construction. Let S be the set of elements to be stored in the new heap Q. We first build an nH tree whose nodes contain sorted items. Such 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 μ_T is built, adjusted and rearranged. The construction is performed by the following program:

```
program MAKEQUEUE (S, Q):
  begin
   for i:=0 to m/n - 1 do          -- Sorting phase
     S := S - {e1, ..., en};
     Q[i·n+1, . . . , (i+1)·n] := PARALLEL-SORT ({e1, . . . ,en});
   endfor;
   for k := h downto 1 do          -- Heapify phase, from the leaves
    for each P at level k do
      Let T be the nH rooted at P;
      Build the minimum-path μT;
      ADJUST (μT);
      REARRANGE (μT, P);
    endfor;
   endfor;
  end.
```

The correctness of the above algorithm can be easily proved by induction on the height of the nH. As to its time complexity, from (1) it follows that the sorting phase can be accomplished in parallel time O((m/n)log n). During the heapify phase, at each node P at 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:

$$O(\min\{k+\log n, k \log\log n\}),$$

for a total time of

$$O\left(\sum_{k=1}^{h} \frac{m}{n2^k} \min\{k+\log n, k \log\log n\}\right) = O((m/n) \log\log n).$$

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

$$C_{MQ} \in O((m/n)\log n). \qquad (7)$$

3.2 PPQ operations on nL

Due to their vectorial representation, nH cannot efficiently realize meldable PPQ. Melding is instead the basic operation for nL, since all the other PPQ operations of insertion, deletion, and heap construction can be implemented through melding.

Melding. To meld two nL, Q_1 and Q_2, into a nL Q, we combine their rightmost paths ρ_1 and ρ_2 into a single path ρ of length $h = h_1+h_2$, where $h_1=|\rho_1|$ and $h_2=|\rho_2|$. 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 in Q.

We represent a node P of an nL by a record P = (E, left, right, rank), where P.E is the sorted sequence of n values stored in P (denoted by E_P is 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 P.E stored at the nodes of path ρ_i. Similarly, we define $L\rho_i$ to be the sequence of the pairs relative to the nodes of ρ_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 reported below. (Note the use of the Pascal-like notation new(P) and P^, to denote a new record to be created, and its fields).

```
program MELD (Q1, Q2, Q):
  begin
    Let ρ1 and ρ2 be the rightmost paths of Q1 and Q2, respectively; and let
    h1 = |ρ1| and h2 = |ρ2|.
    E := PARALLEL-MERGE (Eρ1, Eρ2);
    L := PARALLEL-MERGE (Lρ1, Lρ2);
    Let E^1, ..., E^(h1+h2) be the (h1+h2) consecutive subsequences of E with
    cardinality n;
    Let L^1, ..., L^(h1+h2) be the (h1+h2) 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, 0);
      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.
```

The merge operation on the sequences $L\rho_1$ and $L\rho_2$ is based on the total order defined over the second field of the pairs. Moreover, 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 ρ guarantees the extended heap order as, for each left subtree T pointed by a node P in ρ, the values in P.E can only be less or equal to the ones in the node of ρ_1 or ρ_2 that originally pointed to T.

Finally, it can be easily seen that the time complexity C_M of MELD is dominated by the PARALLEL–MERGE step needed to create E and L, while all the other phases are performed in O(h) time. From (2), we have:

$$C_M \in O(\min\{h_1+h_2+\log n, (h_1+h_2)\,\mathrm{loglog}\, n\}). \qquad (8)$$

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

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

4. Conclusions

The PPQ data type introduced in this paper is based on the idea of extending sequential Priority Queue structures to a parallel context. Adopting the CREW-PRAM model of parallel computation, we have defined two PPQ data structures, the nH and the nL, respectively realizing unmeldable and meldable queues. We have then employed optimal algorithms of 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 operation followed by m/n Deletemins, where m

and n are the elements to be sorted, and the number of processors, respectively. Letting $C_{HS}(m, n)$ be the time complexity of Heapsort, from (4) and (5), we have:

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

that is:

$$C_{HS}(m,n) \in O((m/n \log m))$$

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 $m > n \cdot \log\log n$, we can slightly modify the sorting phase of makequeue to use simpler sorting algorithms (as in [7]) 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.

Acknowledgement

We wish to thank Fabrizio Luccio and Andrea Pietracaprina for helpful suggestions.

References

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

[2] A. Borodin and J.E. Hopcroft, Routing, Merging and Sorting on Parallel Models of Computation, *J. Comput. Syst. Sci.* **30** (1985) 130-145.

[3] R. Cole, Parallel Merge Sort, *SIAM J. Comput.* **17** (4) (1988) 770-785.

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

[5] R.M. Karp, M. Saks and A. Wigderson, On a Search Problem Related to Branch-and-Bound procedures, *Proc. 27th Ann. IEEE Symp. on Foundations of Computer Science* (1986) 19-28.

[6] R.E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[7] L.G. Valiant, Parallelism in Comparison Problems, *SIAM. J. Comput.* **4** (3) (1975) 348-355.

[8] J.W.J. Williams, Algorithm 232: Heapsort, *Comm. ACM* **7** (6) (1964) 347-348.