Parallel Algorithms for Priority Queue Operations*

Maria Cristina Pinotti
Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
Via S.Maria 46, I56100 Pisa, Italy

Geppino Pucci
Dipartimento di Elettronica e Informatica
Università di Padova
Via Gradenigo 6/A, I35131 Padova, Italy

Abstract

This paper presents parallel algorithms for priority queue operations on a p-processor EREW-PRAM. The algorithms are based on a new data structure, the Min-path Heap (MH), which is obtained as an extension of the traditional binary-heap organization. Using an MH, it is shown that insertion of a new item or deletion of the smallest item from a priority queue of n elements can be performed in $O(\frac{\log n}{p} + \log\log n)$ parallel time, while construction of an MH from a set of n items takes $O(\frac{n}{p} + \log n)$ time. The given algorithms for insertion and deletion achieve the best possible running time for any number of processors p, with $p \in O(\frac{\log n}{\log\log n})$, while the MH construction algorithm employs up to $\Theta(\frac{n}{\log n})$ processors optimally. The paper ends with a brief discussion of the applicability of MH's to the development of efficient parallel algorithms for some important combinatorial problems.

Keywords Analysis of Algorithms, Data Structures, Heaps, Parallel Algorithms.

*This work was supported in part by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM). Part of this research was done while the second author was visiting the International Computer Science Institute, Berkeley, California

1 Introduction

A *Priority Queue* (PQ) is an abstract data type storing a set of integer-valued items and providing operations such as insertion of a new item and deletion of the smallest stored item. In this note we introduce the *Min-path Heap* (MH) data structure. We employ this new structure to develop efficient parallel algorithms for the basic PQ operations of insertion, deletion and construction on the EREW-PRAM [5] model of computation.

Several parallel implementations of PQ's can be found in the literature. The first approach to date is due to Biswas and Browne [2], subsequently improved by Rao and Kumar in [7]. In their schemes, $p \in O(\log n)$ processors concurrently access a binary heap of n elements by acquiring locks on the nodes of the heap. Insertions and deletions are executed in a pipelined fashion, thus increasing the throughput of the structure from one to p simultaneous operations. However, the time requirement for a single insertion or deletion remains $O(\log n)$. More recent papers deal with the problem of speeding up a *single* heap operation. In this direction, optimal parallel algorithms for heap construction have been devised for the PRAM model in [6] and [8]. As to insertion and deletion, we are only aware of the implementation devised by Zhang in [9] which requires $O(\log n \log\log n)$ work and $O(\log\log n)$ time with $p = \log n$ PRAM processors. Note that the scheme fails to attain linear speedup by a factor of $O(\log\log n)$.

In the following sections we provide optimal parallel algorithms for PQ operations based on the MH data structure. Our results are the following. Let M be an MH of n elements stored in the shared memory of a p processor EREW-PRAM. We show how to insert a new item or delete the smallest item from M in parallel time $O\left(\frac{\log n}{p} + \log\log n\right)$. Moreover, we adapt the above referenced algorithms for parallel heap construction so that M can be built from a set of n elements in time $O\left(\frac{n}{p} + \log n\right)$. Our insertion and deletion algorithms achieve the best possible running time for any number of processors p, with $p \in O(\frac{\log n}{\log\log n})$, while the MH construction algorithm employs up to $\Theta(\frac{n}{\log n})$ processors optimally.

2 Parallel Algorithms for MH Operations

Our MH data structure is obtained as an extension of the traditional *binary heap* organization. Recall that a binary heap H is a complete binary tree (stored in vectorial form) where each node i contains an item, $H[i]$, whose value is less than the values $H[2i]$ and $H[2i+1]$ stored at its children $2i$ and $2i+1$[1]. For any node i of H, its *min-path* μ_i is the set of nodes defined

[1]For the sake of simplicity, we shall assume that all the items stored or to be inserted in an MH are distinct. The handling of duplicates requires some trivial modifications to the algorithms, whose complexities remain

by the following recurrence:

1. i belongs to μ_i.
2. Let a non leaf node j belong to μ_i. If j has only one child u, then u belongs to μ_i. Otherwise, if u and v are the children of j and $H[u] < H[v]$, then u belongs to μ_i.

Informally, μ_i is the unique path in H from i to a *target leaf* of address L_i, such that each internal node on the path stores an item whose value is less than the one stored at its sibling. Note that since H is stored as a vector, we can easily determine the addresses of all the nodes on μ_i from L_i. More precisely, if $h \geq 1$ is the height of H and i is at level k, $1 \leq k \leq h$, the nodes on μ_i have addresses L_i **div** 2^j, with $0 \leq j \leq h-k$. The importance of min-paths for the realization of fast parallel algorithms for PQ operations will be made clear in subsection 2.2.

We have just shown that in order to have fast access to min-path information it suffices to maintain, for each node i of H, the address of its target leaf L_i. Therefore, we define a *Min-path Heap* M to be a data structure whose representation consists of two vectors:

- M_H, where the items are stored in a binary heap fashion.
- M_L, with $M_L[i]$ storing L_i, that is, the address of the target leaf of node i.

In addition to restoring the heap order on M_H, insertion and deletion algorithms for an MH M have to update the min-path information stored in M_L. We also associate M with an integer variable, N_M, denoting the number of nodes currently stored in M. Note that an MH induces only a constant factor increase in space over the traditional binary heap organization. Moreover, its representation is simple and compact (compare it with the structure in [9], where some nodes have to store a table of $O(\log\log n)$ entries).

In the following sections we will sometimes refer to M, M_H and M_L using the classical binary tree terminology. For instance, we will say that $h = \lfloor \log N_M \rfloor + 1$ is the *height* of M, M_H or M_L and will use terms like *leaf* or *sibling* to denote particular locations in the vectors.

In order to describe the EREW-PRAM algorithms for the basic operations on an MH, we introduce the following conventions. Let S be a set of processor indices. The statement

for $i \in S$ **do in parallel** *statement list* **endfor**

---

unaltered.

denotes $|S|$ parallelizable executions of the statement list, one execution for each index $i \in S$. If a statement is not within the scope of a **for** ... **endfor** construct, it is executed in parallel by all the active processors. Within a **for** ... **endfor** construct, the statement

$$P_i : \text{BROADCAST}(X = x)$$

denotes the computation needed to broadcast the value x from the processor P_i in charge of the i^{th} instance of the statement list to all the active processors. These processors will store the received value in their local variable X. This operation can be realized in time $O(\log p)$ on an EREW-PRAM, where p is the number of active processors. Finally, if M is an MH of height h and L is the address of one of its leaves, the function

$$\text{SIBLING}(M, L, i) = \textbf{ let } K = L \textbf{ div } 2^{h-i} \textbf{ in } K + (-1)^{K \textbf{ mod } 2}$$

returns the address of the sibling of the i^{th} node on the path in M from the root to L.

2.1 Insertion

The algorithm for inserting a new item I in an MH M of height h proceeds in two phases:

1. The processors determine the position that item I has to occupy in the *insertion path* μ_I from the root to the first vacant leaf of M_H (which has address $N'_M = N_M + 1$) so that the heap order in M_H is not violated. Note that the i^{th} node on the insertion path, starting from the root, has address $n_i = N'_M$ **div** 2^{h-i} for $1 \leq i \leq h$. Once such position n_k has been determined, all the items stored in $M_H[n_j]$, $k \leq j \leq h-1$ are shifted to position $M_H[n_{j+1}]$ and I is stored in $M_H[n_k]$.

2. The processors recompute the new target leaves for the nodes of μ_I. For nodes n_j, $k \leq j \leq h$, it must be $M_L[n_j] = N'_M$, as in the updated M_H we have $M_H[\text{SIBLING}(M, N'_M, j)] > M_H[n_j]$, for $k+1 \leq j \leq h$, because of the heap property and the shift performed on μ_I. For nodes n_j, $1 \leq j \leq k-1$, the target leafs may be different only if their min-path μ_{n_j}, prior to the insertion, went through n_k or $\text{SIBLING}(M, N'_M, k)$ (this can be easily checked by comparing $M_L[n_j]$ with $M_L[n_k]$ and $M_L[\text{SIBLING}(M, N'_M, k)]$). The new target leaf for such nodes will be N'_M if $I < M_H[\text{SIBLING}(M, N'_M, k)]$ and $M_L[\text{SIBLING}(M, N'_M, k)]$ otherwise.

Note that the above strategy still yields a valid MH when we insert an item starting from any empty leaf of address different from N'_M. In particular, the deletion algorithm described

in the following subsection creates a "hole" in the structure at the target leaf of the root, $L = M_L[1]$. The hole is then refilled by performing an insertion starting from L.

The following procedure INSERT implements the above ideas. Parameter L is the address of a leaf, while parameter I is the item to be inserted. INSERT uses the auxiliary vectors V_H and V_L to perform operations on the elements stored along the insertion path μ_I and their target leaves.

procedure INSERT(L, I):
  $h := \lfloor \log L \rfloor + 1$; { height of M_H}
  for $i \in \{1, \ldots, h-1\}$ **do in parallel**
    $V_H[i] := M_H[L$ **div** $2^{h-i}]$;
    $V_L[i] := M_L[L$ **div** $2^{h-i}]$;
    {copy the elements and target leaves stored along μ_I}
  endfor;
  for $i \in \{1\}$ **do in parallel**
    $V_H[0] := -\infty$
    $V_H[h] := +\infty$
    {these two dummy elements are needed for the next parallel steps}
  endfor;
  for $i \in \{1, \ldots, h\}$ **do in parallel**
    if $V_H[i] > I$ **and** $V_H[i-1] < I$
      then P_i : BROADCAST(Pos $= i$)
      {i is the position where I must be inserted}
    fi
  endfor;
  for $i \in \{\text{Pos}, \ldots, h-1\}$ **do in parallel**
    TEMP $:= V_H[i]$; $V_H[i+1] := TEMP$;
    { shift the items greater than $I \ldots$ }
    $V_L[i+1] := L$;
    { ...and set their target leaves to L}
  endfor;
  for $i \in \{\text{Pos}\}$ **do in parallel**
    if $i > 1$ **then**
      if $V_H[i] \geq M_H[\text{SIBLING}(M, L, i)]$
        then P_i : BROADCAST$(L_1 = M_L[\text{SIBLING}(M, L, i)])$
        else P_i : BROADCAST$(L_1 = V_L[i])$
      fi;
      { if $V_L[j] = L_1$ for $j <$Pos, then $V_L[j]$ must be set...}
      if $I < M_H[\text{SIBLING}(M, L, i)]$
        then P_i : BROADCAST$(L_2 = L)$
        else P_i : BROADCAST$(L_2 = M_L[\text{SIBLING}(M, L, i)])$
      fi
      { ... to L_2}
    fi;
    $V_H[i] := I$; $V_L[i] := L$;
  endfor;

```
    for i ∈ {1, ..., Pos − 1} do in parallel
        if V_L[i] = L_1
          then V_L[i] := L_2
        fi
    endfor;
    for i ∈ {1, ..., h} do in parallel
        M_H[L div 2^{h−i}] := V_H[i];
        M_L[L div 2^{h−i}] := V_L[i];
        { copy the updated path back}
    endfor
end INSERT.
```

To insert a new item I, we first increment N_M and then call INSERT(N_M,I). The time complexity of INSERT on an MH of n elements and with $p \leq \log n$ processors is determined by the BROADCAST operations (time $O(\log\log n)$) and by the parallel execution of constant-time operations on at most $O(\log n)$ nodes (time $O\left(\frac{\log n}{p}\right)$) for a total time

$$C_I \in O\left(\frac{\log n}{p} + \log\log n\right)$$

It should be noted that the above procedure INSERT can be employed to provide an implementation of the useful *decrease-key* operation [4]. In MH terms, *decrease-key* is given a pointer to a node k of M_H and a value v smaller than $M_H[k]$. The algorithm sets $M_H[k]$ to v and then re-establishes the heap property on M_H. Such readjustment can be obtained by simply considering k as a leaf node and running a slight variant of INSERT with parameters k and v. The details of the algorithm are omitted for the sake of brevity.

2.2 Deletion

The algorithm for deleting the root of an MH M proceeds in three phases:

1. Let $\mu_1 = \{n_1 = 1, \ldots, n_h = L = M_L[1]\}$ be the min-path of the root of M. The root item $M_H[1]$ is returned and the target leaf of the root $L = M_L[1]$ is broadcast to all the processors.

2. Nodes $n_2, \ldots, n_h$ are shifted one position above, that is, $M_H[n_i] := M_H[n_{i+1}]$ (for technical reasons, we set $M[n_h] = +\infty$). Note that this operation restores the heap order in M_H, but disrupts the target leaf information in M_L for the nodes in μ_1.

3. The target leaves for the nodes on μ_1 are recomputed and the "hole" in position L is filled by invoking INSERT(L,$M_H[N_M]$). Finally, N_M is decremented.

It remains to explain how to recompute $M_L[n_i]$, $1 \leq i \leq h$, once μ_1 is shifted upwards. Consider the new min-paths for nodes n_i in the updated structure. Starting from the root and proceeding along the nodes of μ_1, the min-path will follow the same route as before if the new values stored at nodes n_i are still smaller than the ones stored at their siblings. However, whenever we reach a node n_k whose sibling SIBLING(M, L, k) contains now a smaller value, the min-path "deviates" and reaches the target leaf M_L[SIBLING(M, L, k)]. This observation suggests the following strategy to rebuild M_L efficiently in parallel. In the following, RANK1, RANK2, RANK3 and RANK4 are auxiliary vectors of h positions.

1. Each value $M_H[n_i]$, $2 \leq i \leq h$ is compared with its sibling value M_H[SIBLING(M, L, i)]. If $M_H[n_i]$ is smaller, then RANK1$[i]$ is set to 0, otherwise RANK1$[i]$ is set to 1 (the values 1 indicate a "deviation" of the min-path). Note that at least RANK1$[h]$ will be initialized to 1, as we set $M_H[L]$ to $+\infty$.

2. Prefix sums are computed on input RANK1. The results are stored in RANK2.

3. (*compaction*) For each position i, if RANK1$[i] = 1$ (i.e., the min-path deviates at n_i) and RANK2$[i] = j$ then RANK3$[j]$ is set to i (RANK3$[j]$ is the address of the j^{th} deviation).

4. (*target leaf assignment*) For $1 \leq i \leq h-1$ let j_i =RANK2$[i] + 1$ and k_i =RANK3$[j_i]$. $M_L[n_i]$ is set to M_L[SIBLING(M, L, k_i)], which is the target leaf of the sibling of the first node n_{k_i}, following n_i, where the the min-path deviates.

Note that in step 4 the same cell of vector RANK3 could be accessed concurrently by two processors associated to nodes n_i and n_j with RANK2$[i]$ = RANK2$[j]$. This problem is easily overcome by computing the vector RANK4$[i]$ =RANK3[RANK2$[i] + 1$] by means of a simple prefix operation, whose description is omitted for the sake of brevity.

The following procedure DELETEMIN implements the above strategy. In the procedure, we use the statements

PREFIX-SUMS(RANK1,RANK2) and COMPUTE(RANK2,RANK3,RANK4)

respectively to denote the prefix-sums computation with input RANK1 and output RANK2 and the creation of vector RANK4$[i]$ =RANK3[RANK2$[i]$+1]. These operations can be realized on an EREW-PRAM in $O\left(\frac{\log n}{p} + \log\log n\right)$ time [5].

procedure DELETEMIN:
for $i \in \{1\}$ **do in parallel**

return $M_H[1]$;
P_i : BROADCAST(L = $M_L[1]$)
endfor;
{ return the min value and distribute the address of the target leaf of the root}
$h := \lfloor \log L \rfloor + 1$; { length of μ_1}
for $i \in \{1, \ldots, h-1\}$ **do in parallel**
$V_H[i] := M_H[L$ **div** $2^{h-i-1}]$;
{copy and shift the elements stored along μ_1}
endfor;
for $i \in \{h\}$ **do in parallel**
$V_H[h] := +\infty$; RANK1[1]:= 0;
endfor;
for $i \in \{2, \ldots, h\}$ **do in parallel**
if $V_H[i] < M_H$[SIBLING(M, L, i)]
then RANK1[i] := 0
else RANK1[i] := 1
fi;
PREFIX-SUMS(RANK1,RANK2);
if RANK1[i] = 1
then RANK3[RANK2[i]] := i
fi;
{compact the indices of the nodes where the min-path deviates}
endfor;
for $i \in \{1 \ldots, h-1\}$ **do in parallel**
COMPUTE(RANK2,RANK3,RANK4);
{RANK4[i] =RANK3[RANK2[i] + 1]}
$V_L[i] := M_L$[SIBLING$(M, L$,RANK4[i])]
{these are the new addresses of the target leaves of nodes in μ_1}
$M_H[L$ **div** $2^{h-i}] := V_H[i]$;
$M_L[L$ **div** $2^{h-i}] := V_L[i]$;
{ copy the updated path back}
endfor;
for $i \in \{1\}$ **do in parallel**
$I := M_H[N_M]$;
$M_H[N_M] := M_L[N_M] := +\infty$
endfor;
INSERT(L, I);
{fill the hole in $M_H[L]$}
$N_M := N_M - 1$
end DELETEMIN.

The time complexity of DELETEMIN on an MH of n elements is determined by the broadcast and prefix steps, and by the parallel execution of constant-time operations on $O(\log n)$ nodes, for a total time complexity

$$C_D \in O\left(\frac{\log n}{p} + \log\log n\right)$$

2.3 Construction

We are finally left with implementing MH construction. Let $S[1,\ldots,n]$ be the set of n elements to be stored in an MH M. We first build the vector M_H by applying one of the optimal parallel algorithms for heap construction proposed in the literature (see [6] and [8]). The vector M_L is subsequently created by first initializing $M_L[i] = i$ for each leaf i and then computing $M_L[j]$ for any internal node j. More precisely, if $M_L[2j]$ and $M_L[2j+1]$ have been computed, $M_L[j]$ is set to $M_L[k]$, where $k \in \{2j, 2j+1\}$ is such that $M_H[k] = \min\{M_H[2j], M_H[2j+1]\}$.

Construction is performed by the following procedure CONSTRUCT_MH(S, M). In the procedure, the statement

$$\text{BUILD}(S, M_H)$$

denotes the invocation of an optimal parallel heap construction scheme which builds M_H out of S.

```
procedure CONSTRUCT_MH(S, M):
    N_M := |S|; h := ⌊log N_M⌋ + 1;

    {M_H construction:}
    BUILD(S, M_H);

    {M_L construction:}
    count:=2;
    while count> 0 do
        {determine the target leaves of the nodes in the last two levels}
        if h > 0
          then for i ∈ {2^{h-1}, ..., 2^h − 1} do in parallel
                  if 2i ≤ N_M
                    then if 2i + 1 ≤ N_M
                            then if M_H[2i] < M_H[2i + 1]
                                    then M_L[i] := M_L[2i]
                                    else M_L[i] := M_L[2i + 1]
                                 fi
                            else M_L[i] := 2i
                         fi
                    else M_L[i] := i
                  fi
               endfor
        fi;
        h := h − 1; count:=count−1
    endwhile;
    while h > 0 do
        for i ∈ {2^{h-1}, ..., 2^h − 1} do in parallel
```

```
            if M_H[2i] < M_H[2i + 1]
              then M_L[i] := M_L[2i]
              else M_L[i] := M_L[2i + 1]
            fi
        endfor;
        h := h − 1
    endwhile
end CONSTRUCT_MH.
```

Let us analyze the running time of the above procedure. By using the previously referenced schemes in [6] and [8], the M_H construction can be executed in $O\left(\frac{n}{p} + \log n\right)$ time. As to the M_L construction, the algorithm is essentially a min-computation performed along a complete binary tree of n nodes, thus requiring $O\left(\frac{n}{p} + \log n\right)$ time. Therefore, the overall time complexity of the procedure is

$$C_C \in O\left(\frac{n}{p} + \log n\right)$$

3 Conclusions

The *Min-path Heap* (MH) data structure introduced in the previous sections provides an optimal implementation of priority queues on a p-processor EREW-PRAM. We have devised insertion and deletion algorithms for an MH M of n elements which require $O\left(\frac{\log n}{p} + \log\log n\right)$ time and have adapted known parallel heap-construction schemes to build M in $O\left(\frac{n}{p} + \log n\right)$ time. All the algorithms are extremely simple and the orders of magnitude do not hide "big" constants. Moreover, the space requirement of M is only $2n$ memory cells, arranged in two vectors of n locations each.

It has to be noted that the number of processors that can be profitably exploited by our algorithms is (necessarily) small ($p \in O(\log n)$). However, the current (or even foreseeable) technology for the construction of parallel machines with shared memory is applicable only to systems with "few" processors [5]. Our simple and efficient algorithms are suitable for an optimal exploitation of such "coarse grain" parallelism.

For the above systems, MH structures can be employed to optimally speed up those sequential applications which make use of binary heaps and whose time complexity is determined by the cost of the heap operations. Consider, for instance, *Heap-Sort* or the implementation of the *LPT* heuristic for scheduling [3]. The use of MH's yields optimal $O\left(\frac{n \log n}{p}\right)$ time parallel algorithms for the above problems, for any number $p \in O\left(\frac{\log n}{\log\log n}\right)$ of processors. As a final example, consider the straightforward parallelization of Djikstra's algorithm for computing a rooted *Shortest Path Tree* (SPT) of a weighted directed graph. The complexity of the algo-

rithm is dominated by the time needed for n deletions and $O(m)$ *decrease-key* operations on a priority queue of $O(n)$ elements [4]. The use of an MH yields a parallel SPT algorithm with running time $O(\frac{m \log n}{p})$ with $p \in O(\frac{\log n}{\log \log n})$. For this range of processors and $m \in O(\frac{n \log n}{\log \log n})$ this simple algorithm achieves a better processor-time product than the other parallel SPT algorithms in the literature [1, 4].

References

[1] B.Auerbuch and Y.Shiloach, New Connectivity and MSF Algorithms for Ultracomputer and PRAM, in: *Proc. of the 1983 Int. Conf. on Parallel Processing* (1983) 298-319.

[2] J.Biswas and J.C.Browne, Simultaneous Update of Priority Structures, in: *Proc. of the 1987 Int. Conf. on Parallel Processing* (1987) 124-131.

[3] T.H.Cormen, C.E.Leiserson and R.L.Rivest, *Introduction to Algorithms* (MIT Press, Cambridge Mass., 1990).

[4] J.M.Driscoll, H.V.Gabow, R.Shrairman and R.E.Tarjan, Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation, *Communications of the ACM* **31**(11) (1988) 1343-1354.

[5] R.M.Karp and V.Ramachandran, Parallel Algorithms for Shared-Memory Machines, in: J.van Leeuween, ed., *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity* (Elsevier, Amsterdam, 1990) 870-941.

[6] S.Olariu and Z.Wen, An Optimal Parallel Construction Scheme for Heap-like Structures, in: *Proc. Twenty-eight Allerton Conf. on Communication, Control, and Computing* (1990) 936-937.

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

[8] V.N.Rao and W.Zhang, Building Heaps in Parallel, *Information Processing Letters* **37** (1991) 355-358.

[9] W.Zhang and R.Korf, Parallel Heap Operations on EREW PRAM, To appear in: *Sixth Int. Parallel Processing Symp.(IPPS'92)* (1992).