Priority Queue Operations on EREW-PRAM

Mauricio Marín

PRG, Computing Laboratory, University of Oxford
Wolson Building, Parks Road, Oxford OX1 3QD, England, UK
E-mail: mmarin@comlab.ox.ac.uk

Abstract. Using EREW-PRAM algorithms on a tournament based complete binary tree we implement the *insert* and *extract-min* operations with $p = \log N$ processors at costs $O(1)$ and $O(\log \log N)$ respectively. Previous solutions [4, 7] under the PRAM model and identical assumptions attain $O(\log \log N)$ cost for both operations. We also improve on constant factors the asymptotic bound for *extract-min* since in it we reduce the use of communication demanding primitives. The tournament tree enables the design of parallel algorithms that are noticeably simple.

1 Tournament trees

Our data structure is a complete binary tree (CBT). Every item stored in the tree consists of a priority value and an identifier. We associate every leaf of the CBT with one item, and use the internal nodes to maintain a continuous binary tournament among the items. A match, at internal node n, consists of determining the item with higher priority (lesser numerical value) between the two children of n and writing the identifier of the winner in n. The tournament is made up of a set of matches played in every internal node located in each path from the leaves to the root. Every time we change the priority associated with a leaf l, the tournament is updated by performing matches along the unique path between l and the root of the tree. We call this last operation *update-cbt*. The operations *extract-min* and *insert* are implemented using *update-cbt* as a basic primitive.

The CBT can be represented implicitly in an array of $2N - 1$ tuples (i, x), where i is an item identifier and x its associated priority. (N is the number of items stored in the priority queue). A node at position n in the array CBT has its children at positions $2n$ and $2n + 1$. The parent of a node n is at position $\lfloor \frac{n}{2} \rfloor$. All internal nodes are stored between positions 1 and $N - 1$ of the CBT. We also use an array Leaf[1..N] of integers to map between items and leaves. To enable dynamic reusing of item identifiers in the PQ, the array Leaf is also used to maintain a single linked list of available item identifiers (initially this list is empty).

The results of every match performed in the internal nodes of the CBT are written as (i, x) in every internal node n, namely in n it is written the item identifier i in CBT[n].i and the priority x associated with i in CBT[n].x. The highest priority in the PQ is given by CBT[1].x, its identifier is CBT[1].i, and its associated leaf is at position Leaf[CBT[1].i].

Deletions in the CBT are performed by removing the child with lower priority between the children of the parent of the rightmost leaf, and exchanging it with the target leaf to be deleted. On the other hand, insertions are performed by appending a new rightmost leaf and updating the CBT. This is done by expanding in two leaves the first leaf of the tree.

2 Single operations in parallel

The cost of *extract-min* and *insert* is constant except for the cost of *update-cbt.* So we focus on the parallel implementation of *update-cbt.* We split this operation into two: *I-update-cbt* which is executed by *insert* and *E-update-cbt* executed by *extract-min.* In the description of these operations we are going to assume $p = \log N$ processors[1].

During an *insert*(x) operation and after creating a new leaf at position L in the CBT to hold the new item (i, x), the *I-update-cbt*(i, x, L) operation simultaneously compares x with all the priorities stored along the path from L to the root. This is made as shown in Figure 1. Note that this operation takes $O(1)$ parallel time if we use $\log N$ processors. Then its cost is $O(\frac{\log N}{p})$ for $p \leq \log N$.

```
procedure I-update-cbt($i, x, L$)
    $h$:= $\lfloor \log L \rfloor$;
    for $p \in \{1...h\}$ do in parallel
        $a$:= $L$ div $2^{h-p+1}$;
        if ( CBT[$a$].$x > x$ ) then
            CBT[$a$].$i$:= $i$;
            CBT[$a$].$x$:= $x$;
        endif
    endfor
end
```

Fig. 1. Pseudo-code for the parallel *I-update-cbt* operation.

The details for the *extract-min* operation are as follows. Let us define j= CBT[1].i, y= CBT[1].x, L= Leaf[j], and let (i, x) be the tuple selected to replace (j, y) in leaf L. Before making effective this replacement and after broadcasting (i, x, L), the *E-update-cbt* operation proceeds in three main steps. Firstly, in every internal node n along the path from leaf L till the root we write the

[1] Note that, similar to [4], we are not considering here the $O(\log p)$ cost of broadcasting the key to all the processors, in which case our *insert* operation has the same asymptotic cost than [4] but with much simpler algorithms and without the cost of additional broadcasts.

tuple (k, z) stored in one of the children of n so that $k \neq j$ (recall that item j is duplicated along all this path). Secondly, a parallel prefix operation (with operator $\oplus$ = *min*) is performed among all the nodes in the path from L to the root in order to re-establish the binary tournament invariant (similar to [4] this operation calculates all the prefixes $a_i = a_1 \oplus a_2 \oplus a_3 \cdots \oplus a_i$ for each $i = 1, 2, \cdots, h$ where h is the height of the CBT). Finally, an attempt for storing (i, x) in each node n in the path from L to the root is made as we do in Figure 1, and then the replacement of (j, y) by (i, x) is performed in the leaf at L. The *E-update-cbt*(i, x, L) operation is described in Figure 2 (similar to [4] we use two auxiliary arrays, I and X, of size $\log N$ to perform the *parallel prefix* operation).

Then the cost of this operation is dominated by the cost of *parallel prefix* which is $O(\log\log N)$ when we use $p = \log N$ processors. Then the cost of *extract-min* is $O(\frac{\log N}{p} + \log\log N)$ with $p \leq \log N$.

```
procedure E-update-cbt($i$,$x$, $L$)
    $h$:= $\lfloor \log L \rfloor$;
    for $p \in \{1...h\}$ do in parallel
        $a$:= $L$ div $2^{h-p+1}$; $j$:= CBT[$a$].$i$;
        if ( CBT[$2a$].$i \neq j$ ) then $b$:= $2a$;
          else $b$:= $2a + 1$;
        $I[p]$:= CBT[$b$].$i$; $X[p]$:= CBT[$b$].$x$;
    endfor
    ALL-PREFIX-MIN($I, X, h$);
    for $p \in \{1...h\}$ do in parallel
        if ( $X[p] > x$ ) then
            $X[p]$:= $x$; $I[p]$:= $i$;
        endif
    endfor
    for $p \in \{1...h\}$ do in parallel
        $a$:= $L$ div $2^{h-p+1}$;
        CBT[$a$].$i$:= $I[p]$; CBT[$a$].$x$:= $X[p]$;
    endfor
end
```

Fig. 2. Pseudo-code for the parallel *E-update-cbt* operation.

Similar to other approaches [1, 3, 4, 5, 7], the construction of the CBT from a set of N priority values takes $O(\frac{N}{p} + \log N)$ parallel time: in parallel each processor takes a subset of $\frac{N}{p}$ elements, and then level by level from leaves to root the matches are simultaneously performed in the internal nodes.

3 Final comments

We have described parallel algorithms for speeding up single-item priority queue operations on a p-processor EREW-PRAM. Our results improve previous solutions. This improvement comes from the use of a tournament based complete binary tree (CBT) rather than variations to the standard implicit heap. This data structure has also enabled improved implementations of multiple-item priority queue operations on the PRAM and BSP models [2].

As we showed in this paper, the CBT enables a more efficient implementation of the *extract-min* and *insert* operations with the additional advantage that the algorithms involved are noticeably simpler than the ones proposed in previous approaches. One disvantage of the CBT, however, is its higher requirement in memory used. In particular, for a PQ with N items we needed to use an array of $2N - 1$ tuples (i, x) which is equivalent to duplicate the number of items in the PQ. Nevertheless, this memory overhead might be reduced by, for example, halving the number of leaves in the tree since the winner between two leaves can be actually maintained in their common father.

References

1. C. Luchetti and M.C. Pinotti. "Some comments on building heaps in parallel". *Inf. Proc. Letters*, 47:145–148, 1993.
2. M. Marin. "Binary tournaments and priority queues: PRAM and BSP". Technical report PRG-TR-7-97, Oxford University, Jan. 1997.
3. S. Olariu and Z. Wen. "Optimal parallel initialization algorithms for a class of priority queues". *IEEE Trans. Parallel Distrib. Systems*, 2:423–429, 1991.
4. M.C. Pinotti and G. Pucci. "Parallel algorithms for priority queue operations". In *SWAT'92 LNCS 621*, pages 130–139, 1992.
5. N.S. Rao and W. Zhang. "Building heaps in parallel". *Inf. Proc. Letters*, 37:355–358, 1991.
6. V.N. Rao and V. Kumar. "Concurrent access of priority queues". *IEEE Trans. Comput.*, 37(12):1657–1665, 1988.
7. W. Zhang and R. Korf. "Parallel heap operations on EREW PRAM". In *6th Int. Parallel Processing Symposium*, pages 315–318, 1992.