exposes the recursive relation of the frequency in the number of item identifiers found in all the paths from leaves to the root (this relation remains unchanged for any other enumeration which gives a valid tournament tree).

Assuming that we can select any of the $N$ leaves at level $k$ as the target leaf for an insertion, we define $T(k, i)$ as the total number of paths with $i$ distinct item identifiers. The number of paths with $i$ identifiers will be the numbers of paths at level $k - 1$ that include the identifier in the $k$-th level plus the number of paths at level $k - 1$ not including the $k$-level identifier (note that both sets are disjoint). That is:

$$T(k, i) = \begin{cases} 
0 & i < 1 \text{ or } i > k \\
1 & k = 1 \text{ or } k = i \\
T(k - 1, i - 1) + T(k - 1, i) & \text{otherwise}
\end{cases}$$

It is easy to verify that $T(k, i) = \binom{k - 1}{i - 1}$ by induction using the well known property that $\binom{k}{i} = \binom{k - 1}{i - 1} + \binom{k - 1}{i}$. Then the average number of item identifiers in a path is given by

$$h^* = \frac{1}{2^{k-1}} \sum_{i=1}^{k} \frac{k}{i} \cdot T(k, i)$$

$$= \frac{1}{2^{k-1}} \sum_{i=1}^{k} \frac{k}{i} \binom{k - 1}{i - 1}$$

$$= \frac{1}{2^{k-1}} ((k - 1)2^{k-2} + 2^{k-1})$$

$$= (k + 1)/2 \leq \log N - 1 + h^*/2 + 1 = h/2 + 1$$

4 Parallel priority queues on a CREW-PRAM

Parallel priority queues (PPQs) were first introduced in [10]. In [10] an implicit heap based data structure is proposed to implement PPQs. Given a PPQ with $m = n \cdot N$ items, $h = \log(m/n)$ and an $n$-processor CREW-PRAM, the structure in [10] supports the insertion and deletion of $n$ items in $O(h + \log n)$ and $O(h \log \log n)$ parallel time respectively (we assume the practical implementation of ADJUST procedure described in [10]). These performance bounds are achieved by using a standard implicit heap in which every node is able to store $n$ items. The heap invariant is extended by considering that the item with lowest priority among the $n$ items stored in a node $d$ has higher priority than all the items stored in the descendants of $d$.

In this section we implement PPQs using the tournament based CBT. We assume that every item consists of $n$ sub-items or priorities (i.e., there is a total of $N$ items holding a total of $n \cdot N$ priority values with their associated data). Our data structure is similar to that defined in section 3 but with the difference that now the array Prio is able to store for every item a set of $n$ priorities. For each item $i$ its $n$ priority values are maintained sorted in Prio[i]. The winner of a match among
procedure \textit{I-update-cbt}(i, S, L)
\begin{align*}
h &:= \lfloor \log L \rfloor;\\
\text{for } p &\in \{1..h\} \text{ do in parallel}\\
&I[p] := CBT[L \div 2^{h-p+1}];\\
\text{end for}\\
\text{Build up array } D \text{ from } I \text{ without duplicates;}\\
\text{Set } E := \cup \text{ Prio}[k] \text{ with } k \in D;\\
T &:= \text{SORT}(S);\\
P &:= \text{MERGE}(E, T);\\
\text{Set } \text{Prio}[k] \cup \text{Prio}[i] := P \text{ with } k \in D;\\
\end{align*}

end

Figure 1: Insert \textit{update-cbt} operation for parallel PQs.

two sibling nodes is the node \(d\) whose \(n\) associated priority values are all less (higher priority) than its sibling.

Similar to \cite{10} we use the parallel primitives \text{SORT}(S) \text{, with } |S| = n, \text{ and } \text{MERGE}(E_1, E_2) \text{ which on an } n\text{-processor CREW-PRAM have complexities } O(\log n) \text{ and } O\left(\frac{|E_1|+|E_2|}{n}\log \log n\right) \text{ respectively. Notice that we could replace these expensive operations by a parallel SELECT as it is proposed in [3] (see next section).}

As in the sequential case, the insertion of a set \(S\) of \(n\) priorities associated with a new item \(i\) is accomplished by transforming into an internal node with two leaf children the leaf located at position \(N\) of the CBT. During an \textit{insert} and after setting CBT[2 \(N\)] := CBT[N], CBT[2 \(N\) + 1] := \(i\), and Leaf[\(i\)] := 2 \(N\) + 1 the \textit{update-cbt}(i, S, Leaf[\(i\)]) operation executes the steps shown in figure 1.

In the figure, the construction of array \(I\) takes \(O(1)\) parallel time, array \(D\) takes \(O(h)\) sequential time (duplicated values in \(I\) are always in successive positions), and set \(E\) involves \(O(h^*)\) parallel time using \(n\) processors with \(h^* = |D| \leq h\) (see section 3). The parallel merge takes time \(O\left(\frac{nh^* + n}{n}\log \log n\right) = O(h^* + \log \log n)\) whereas the parallel sort takes time \(O(\log n)\). Finally, updating array \text{Prio} takes \(O(h^*)\) parallel time. Then the cost of \textit{insert} is \(O(h^* + \log n)\).

As a comparison with the sequential case and assuming that there are a total of \(m\) items in the parallel and sequential queue, the sequential insert takes \(O(n \log m)\) time whereas the parallel insert \(O(\log(m/n) + \log n)\) time (worst case bounds).

The extraction of the set \(S\) containing the \(n\) highest priorities in the CBT is performed as follows. Let us assume that \(L\) is the position of the leaf that holds the item which contains \(S\) and \(i\) is the item selected to be stored in \(L\). During an \textit{extract-min} and after setting Leaf[\(i\)] := \(L\) and CBT[\(L\)] := \(i\), the \textit{Extract-min update-cbt}(\(L\)) operation execute the steps shown in figure 2. Every step of this operation is dominated by the complexity of \text{MERGE}. Then this operation takes \(O(h \log \log n)\) parallel time.