procedure E-update-cbt(L)
    \( h := \lfloor \log L \rfloor \);
    for \( j := h \) downto 1 do
        \( a := 2 \left( L \div 2^{h-j+1} \right) \);
        \( b := a + 1 \);
        \( x := \max\{\text{Prio}[\text{CBT}[a]]\} \);
        \( y := \max\{\text{Prio}[\text{CBT}[b]]\} \);
        if \( (x > y) \) then swap\((a, b)\);
        \( \text{CBT}[L \div 2^{h-j+1}] := a \);
        \( E := \text{MERGE}(\text{Prio}[a] \cup \text{Prio}[b]) \);
        Let \( E^1, E^2 \) be the left and right halves of \( E \).
        \( \text{Prio}[a] := E^1 \);
        \( \text{Prio}[b] := E^2 \);
    endfor
end

Figure 2: Extract-min update-cbt operation for parallel PQs.

Note that in our case it is necessary to execute one MERGE per update step whereas in [10] this operation is called twice. However, [10] claims \( O(h + \log \log n) \) complexity for extract-min by way of a parallel implementation of ADJUST (see [10] for details on this operation). Thus the implicit heap achieves better theoretical performance bound for extract-min than the CBT. The reason is that the CBT needs to update the complete path from leaf to root, in order to determine the new set of \( n \) “winners” of the binary tournament. In regard to the comparison with the sequential case, for a queue with \( m \) items, the sequential extract-min takes \( O(n \log m) \) time whereas the parallel takes \( O(\log (m/n) \log \log n) \) time (worse case bounds).

5 Parallel priority queues on BSP computers

Recently a BSP implementation of the PPQ introduced in [10] was proposed in [3]. This BSP-PPQ uses a novel mapping between nodes and processors, which reduces the amount of inter-processor communications needed to maintain the extended heap invariant. In addition [3], uses a parallel SELECT operation instead of MERGE and SORT which further increases the efficiency of the BSP-PPQ (with these modifications [3] claims better performance than previous approaches [10, 12, 14] improving on communication and slackness requirements).

In [3], the \( n \) items stored in every node of the implicit heap are randomly and evenly distributed among the \( p \) processors, about \( n/p \) per processor with \( n/p > 1 \), and the reduction in communication comes from the fact that the administration of these items is essentially made by using a parallel SELECT operation.
SELECT(S, c) determines the c-th smallest item in \( S \) and requires \( T_{\text{sel}}(|S|, c, p) \approx O\left((|S| + c)/p + (|S|/p)g + (\log p) \ell \right) \) BSP parallel time. In addition to selection, [3] uses a parallel prefix operation with cost \( T_{\text{ppf}}(p) = O((\log p)(1 + g + \ell)) \) (see [3] for details on \( T_{\text{sel}} \) and \( T_{\text{ppf}} \)). Then for a 2-ary parallel heap, [3] implements the operations \textit{insert} and \textit{extract-min} at costs

\[
hT_{\text{sel}}(2n, n, p) + O(h)
\]
and

\[
h(2T_{\text{sel}}(2n, n, p) + 2T_{\text{ppf}}(p) + O(1))
\]
respectively, where \( h = \log\left(\frac{m}{n}\right) \) is the depth of the heap.

Using the CBT, we improve these bounds on constant factors by reducing the number of parallel primitive operations (and thereby the amount of inter-processor communication and synchronisation) executed in \textit{insert} and \textit{extract-min}. Our performance bounds are

\[
h^*T_{\text{sel}}(2n, n, p) + O(h)
\]
and

\[
h(T_{\text{sel}}(2n, n, p) + T_{\text{ppf}}(p) + O(1))
\]
for \textit{insert} and \textit{extract-min} respectively. We use \( h^* \leq h \) to imply that on average \( h^* \) is less than \( h \) (see section 3), where \( h^* \) is the average number of different item identifiers found in the path from any leaf to the root of the CBT. In the worst case we have \( h^* = h \), but on the average case this occurs only with very low probability \( 1/N \).

We build upon the same basic ideas of [10] and [3] but using the CBT and bottom-top tournaments. We maintain the arrays CBT and Leaf in each processor. We also maintain an instance of array Prio in every processor but the elements of Prio are randomly and evenly distributed among the \( p \) processors. For any item \( i \) we maintain in each processor \( k \) a total of \( o(n/p) \) priorities associated with \( i \) in the respective segment of Prio[\( i \)] stored in processor \( k \). Similar to [3], we use the SELECT operation to redistribute the contents of Prio[\( i \)] associated with every item \( i \) along the path from a leaf \( L \) to the root of the CBT. In addition we use a parallel operation MAX(S), with cost \( O(|S|/p) + T_{\text{ppf}}(p) \), which obtains the maximum priority value among a set \( S \) of \( |S| \) priorities distributed in \( p \) processors (after executing MAX the maximum is "known" by all the processors).

The insertion of a set \( S \) of \( n \) priorities associated with a new item \( i \) is made by transforming into an internal node with two leaf children the leaf located at position \( N = m/n \) of the CBT. During an \textit{insert} and after setting CBT[\( 2N \)]=CBT[\( N \)], CBT[\( 2N + 1 \)]=\( i \), and Leaf[\( i \)]=\( 2N + 1 \) the operation \textit{Insert bsp-update-cbt}(\( i \), \( S \), Leaf[\( i \)]) is executed in each processor as shown in figure 3. The construction of arrays \( D \) and \( I \), and the updating of array Prio[\( k \)] (by using SELECT [3]) takes time \( O(h) \). The worst case for the size of array \( D \) is \( h \) (i.e., on average \( |D| = h^* \)), and the initial random distribution of \( S \) among the \( p \) processors takes time \( o(n/p)g + \ell \). Thus the insertion cost is dominated by the \( h^* \) calls to SELECT which leads to the above presented performance bounds for BSP \textit{insert}.

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 \( i \).