Algorithms for Construction of Optimal and Almost-Optimal Length-Restricted Codes

Marek Karpinski * Yakov Nekrich †

Abstract. In this paper we present new results on sequential and parallel construction of optimal and almost-optimal length-restricted prefix-free codes. We show that length-restricted prefix-free codes with error $1/n^k$ for any $k > 0$ can be constructed in $O(n \log n)$ time, or in $O(\log n)$ time with $n$ CREW processors. A length-restricted code with error $1/n^k$ for any $k \leq L/\log n$, where $\Phi = (1 + \sqrt{5})/2$, can be constructed in $O(\log n)$ time with $n/\log n$ CREW processors. We also describe an algorithm for the construction of optimal length-restricted codes with maximum codeword length $L$ that works in $O(L)$ time with $n$ CREW processors.

1 Introduction

Consider a list of items $e_1, e_2, \ldots, e_n$ with weights $\overline{p} = p_1, p_2, \ldots, p_n$ respectively. A code with lengths $L = l_1, l_2, \ldots, l_n$ is a prefix-free code if no codeword is a prefix of another one. A (prefix-free) code is a length-restricted (or length-limited) code for some integer $L$ if $l_i \leq L$ for all $1 \leq i \leq n$. A code

*Dept. of Computer Science, University of Bonn. Work partially supported by DFG grants, Max-Planck Research Prize, DIMACS, and IST grant 14036 (RAND-APX). Email marek@cs.uni-bonn.de.
†Dept. of Computer Science, University of Bonn. Work partially supported by IST grant 14036 (RAND-APX). Email yasha@cs.uni-bonn.de.
is called a minimum redundancy code or Huffman code for the set of items with weights \( \tilde{p} = p_1, p_2, \ldots, p_n \) if \( \text{Length}(L, \tilde{p}) = \sum p_i \) is minimal among all prefix-free codes. A code \( L \) is a minimum redundancy length-restricted code if \( \text{Length}(L, \tilde{p}) \) is minimal among all length-restricted prefix-free codes. The problem of length-restricted coding is motivated by practical implementations of coding algorithms. If a codeword does not fit into a machine word this can lead to less efficient decoding algorithms.

A Huffman code can be constructed in \( O(n \log n) \) time or in \( O(n) \) time if elements are sorted by weight (see, for instance [vL76], [MK95]). However, the construction of a length-restricted minimum redundancy code requires more time. Garey [G74] has described an algorithm for constructing length-restricted codes that runs in \( O(n^2 L) \) time. Larmore and Hirschberg [L87] described an algorithm that requires \( O(n^{3/2} L \log^{1/2} n) \) time. In [LH90] the same authors presented a \( O(nL) \) time sequential algorithm, based on the Package-Merge paradigm. Katajainen, Moffat and Turpin [KMT95] described an \( O(nL) \) time in-place implementation of the Package-Merge approach. In [LM02] Lidell and Moffat presented an algorithm that works in \( O((H - L + 1)n) \) time, where \( H \) is the height of the longest codeword in a Huffman code (without length restrictions). This leads to, e.g., a linear time algorithm for the case when \( L = H - c \), where \( c \) is a constant. Using the problem reduction due to Larmore and Przytycka (see [LP95]), Schieber [S95] has given an \( O(n^{2O(\sqrt{\log L \log \log n})}) \) algorithm for this problem. Although this algorithm is slightly asymptotically faster than [LH90] and [KMT95], we do not know of any practical implementations of this algorithm.

Miliou, Pessoa and Laber [MPL98] described an algorithm for length-restricted codes with error \( 1/F_i - \log[n + [\log n - L]] + 1 \), where \( F_i \) is the \( i \)-th Fibonacci number. Their algorithm runs in \( O(n) \) time for a sorted list of weights. In [MPL99] the same authors presented a heuristic solution and demonstrated its efficiency in practice.

The fastest \( n \)-processor algorithm for the construction of Huffman codes (without length restriction) is due to Larmore and Przytycka [LP95]. Their algorithm, based on a reduction of the Huffman tree construction problem to the concave least weight subsequence problem runs in \( O(\sqrt{n \log n}) \) time. An algorithm from [MPL99a] runs in \( O(H \log \log (n/H)) \) time with \( O(n) \) work, where \( H \) is the height of a Huffman tree. Kirkpatrick and Przytycka [KP96] introduced a problem of constructing so called almost optimal codes, i.e. the problem of finding a tree \( T' \) that is related to the Huffman tree \( T \) according to the formula \( \text{wpl}(T') \leq \text{wpl}(T) + n^{-k} \) for an arbitrary error parameter \( k \) (assuming \( \sum p_i = 1 \)). They presented an efficient parallel algorithm for the construction of almost optimal codes that works in \( O(k \log n \log^* n) \) time with \( n \) processors on a CREW PRAM, and an \( O(k^2 \log n) \) time algorithm
that works with \( n^2 \) processors on a CREW PRAM. These results were further improved in [BK N02].

In this paper we present a parallel algorithm for the construction of minimum-redundancy length-restricted codes that is based on the Package-Merge algorithm of Larmore and Hirschberg [LH90]. Our algorithm constructs a length-restricted code in \( O(L) \) time with \( n \) processors on a CREW PRAM. Thus our algorithm has the same time-processor product as the sequential algorithm of [LH90].

We also consider the problem of constructing the \textit{almost-optimal} length-restricted codes. We show that an almost-optimal code with error \( 1/n^k \) for any \( k > 0 \) can be constructed in \( O(kn \log n) \) time using a combination of results from [LP95] and [AST94]. We also describe an alternative algorithm based on Package-Merge that works with an error \( 1/n^k \) in \( O(k \log n) \) time with \( n \) processors on a CREW PRAM. Besides that, we present an algorithm that works sequentially in time \( O(n) \) or in logarithmic time with \( O(n/\log n) \) processors and constructs a code with error \( 1/n^k \), where \( k \leq L/\log_4 n \) and \( \Phi = (1 + \sqrt{5})/2 \).

The rest of this paper is structured as follows. In the next section we sketch the Package-Merge algorithm. In section 3 we describe algorithms for the construction of almost-optimal codes. In sections 4 and 5 we describe an efficient parallelization of Package-Merge. This parallelization leads to an \( O(L) \) time \( n \)-processor algorithm for minimum-redundancy length-limited codes, and to an \( O(\log n) \) time \( n \)-processor algorithm for almost-optimal length-limited codes with error \( 1/n^k \).

## 2 Package-Merge

In this section we give a sketch of Package-Merge. In the Package-Merge algorithm \( L \) lists of trees \( S_i \) are constructed. A list \( S^1 \) consists of \( n \) leaves with weights \( p_1, p_2, \ldots, p_n \), sorted according to their weight. The list \( S^{j+1} \) is created from the list \( S^j \) by forming new trees \( t_{i+1} = \text{meld}(t_i, t_{i+1}) \) and merging the list of new elements with a copy of the list \( S^1 \). Here \( t_i \) denotes the \( i \)-th item in the list \( S^j \). An operation \( \text{meld}(t', t'') \) creates a new tree \( t \) with two sons \( t' \) and \( t'' \), such that the weight of \( t \) equals to the sum of weights of its sons. By merging two sorted lists \( S_1 \) and \( S_2 \) we mean constructing a sorted list \( S_3 \) that consists of all elements from \( S_1 \) and \( S_2 \). The depth of the element \( p_i \) equals to the number of occurrences of \( p_i \) in the first \( 2n - 2 \) trees of the list \( S^k \). On Figure 1 we show how the algorithm Package-Merge works on the set of items with weights \( \mathbf{7} = 1, 1, 3, 7, 11, 15 \) for \( L = 4 \). The resulting code consists of codewords with lengths \( \mathbf{L} = 4, 4, 3, 2, 2, 2 \) respectively.
Figure 1: An example of Package-Merge for \( L = 4 \). Elements of \( S^1 \) are marked by squares, elements resulting from melding elements on the previous list are marked by circles.

When list \( S^L \) is constructed, we can compute depths of all elements in an optimal code in \( O(L) \) time with \( n \) processors. Indeed, \( S^L \) consists of \( 2n - 1 \) trees, and these trees have in total at most \( n \) leaves on every tree level. These leaves correspond to elements \( p_1, \ldots, p_n \). We can mark all nodes in the biggest tree in \( S^L \) and then compute all occurrences of \( p_i \) in the \( 2n - 2 \) smallest trees in time \( O(L) \).

In sections 4 and 5 we describe parallel algorithms for the construction of \( S^L \). We will see in section 4 that the most time-consuming operation is the merging of two lists. We show how after a certain pre-processing stage a logarithmic number of merge operations can be performed in logarithmic time with \( n \log n \) processors. During this pre-processing stage we compute the predecessor values \( \text{pred}(e, i) \) for every element \( e \) and every list \( S^j \). These values can be efficiently re-computed after a meld operation and they will allow us to merge arrays in constant time. In section 5 we show how the number of processors can be reduced from \( n \log n \) to \( n \).

## 3 Almost-optimal length-restricted codes

We define average length of a code \( \mathcal{L} \) as \( \text{AvLen}(\mathcal{L}, \bar{p}) = \frac{\text{Length}(\mathcal{L}, \bar{p})}{P} \), where \( P = \sum_{i=1}^{n} p_i \). We say that a length-restricted code \( \mathcal{L} \) is almost-optimal with error \( \epsilon \), if \( \text{AvLen}(\mathcal{L}, \bar{p}) \leq \text{AvLen}(\mathcal{L}', \bar{p}) + \epsilon \) for all length-restricted codes \( \mathcal{L}' \). Below we show how an almost-optimal length-restricted code with error \( \frac{1}{n^\epsilon} \) can be sequentially constructed in time \( O(n \log n) \). Observe that \( P = \sum p_i \) is the length of the message, and coding error equals to the average compression loss per symbol. Therefore, if we want to compress the message of length
Page 5

\[ O(n^k), \text{ using a code with error } 1/n^k \text{ instead of an optimal length-limited code would lead to only a constant increase in length of the compressed message. Besides that, if message length is } O(n^{k'}) \text{ with } k' < k, \text{ then a code with error } 1/n^k \text{ is optimal.} \]

To achieve this goal, we construct an optimal code for the “quantized” set of weights \( \overline{p}_{\text{new}} = p^{\text{new}}_1, p^{\text{new}}_2, \ldots, p^{\text{new}}_n \). Before we define \( p^{\text{new}}_i \), consider weights \( p_i \), where \( p_i = \left\lfloor P/(n^k) \right\rfloor \left( \left\lfloor P/n^k \right\rfloor \right) \). For any code \( \mathcal{L} \), \( \sum l_i p_i \leq \sum l_i (p_i + (P/n^k)) \sum l_i p_i + P \cdot n^{-k+2} \), since \( l_i \leq n \). Hence \( \text{AvLen}(\mathcal{L}, \overline{p}) \leq \text{Length}(\mathcal{L}, \overline{p})/P \leq \text{AvLen}(\mathcal{L}, \mathcal{P}) + n^{-k+2} \).

Let \( \mathcal{L}^* \) be an optimal length-restricted code for \( \overline{p} \), and \( \mathcal{L}^A \) be an optimal length-restricted code for \( \overline{p} \). Then \( \text{AvLen}(\mathcal{L}^A, \overline{p}) \leq \text{AvLen}(\mathcal{L}^*, \overline{p}) \leq \text{AvLen}(\mathcal{L}^*, \mathcal{P}) \leq \text{AvLen}(\mathcal{L}^*, \mathcal{P}) \leq n^{-k+2} \). Therefore we can construct an optimal code for weights \( p_i \), then replace \( p_i \) with \( p_i \) and the resulting code will have an error at most \( n^{-k+2} \). All weights \( p_i \) are divisible by \( [P/n^k] \). We define \( p^{\text{new}}_i = p_i/([P/n^k]) = p_i/([P/n^k]) \). An optimal code for weights \( p^{\text{new}}_i \) is also an optimal code for \( p_i \). Hence we can construct an optimal code for \( p^{\text{new}}_i \), then replace \( p^{\text{new}}_i \) with \( p_i \), and the resulting code will also have an error at most \( n^{-k+2} \). Since \( p_i < P \), all weights \( p^{\text{new}}_i < n^k \) for all \( i \).

Observe that instead of division by \([P/n^k]\) we can set \( p^{\text{new}}_i = [P/2^m] \) for \( m \) such that \( [P/n^k] \leq 2^m \leq 2[P/n^k] \). This would increase coding error by at most a factor of 2 and allow us to construct the new set of weights using only bit operations, since division by a power of 2 can be implemented as a right bit shift.

The construction of a length-restricted code with maximum codeword length \( L \) can be reduced to finding a minimum-weight \( L \)-link path in a graph with the concave Monge property (see [LP95]). The last problem can be solved in \( O(n \log U) \) time, where \( U \) is the maximum absolute value of the edge weights in a graph ([AST94]). The graph described in [LP95] has \( n \) nodes and edges \((i, j)\), s.t. \( i < j \) and \( 2j - i \leq n \). Edge \((i, j)\) has weight \( w(i, j) = \sum_{k=1}^{2j-i} p_k \). Since \( p^{\text{new}}_i < n^k \) for all \( i \), \( w(i, j) < n^{k+1} \) \( \forall i, j \), and \( U < n^{k+1} \). Hence, we can construct an almost optimal code with error \( 1/n^k \) in \( O(kn \log n) \) time.

We can also construct a length-restricted code with error \( 1/n^k \) in logarithmic parallel time with \( n \log n \) operations using the \textbf{Package-Merge} approach and “quantized” weights \( p^{\text{new}}_i \). In [B93] it was shown that maximal codeword length of a Huffman code does not exceed \( \min\left(\left\lceil -\log_2 p^{\text{min}}_i \right\rceil, n-1 \right) \), where \( p^{\text{min}}_i = p^{\text{min}} / P \) is the minimal normalized weight. Since for the set of weights \( \overline{p}_{\text{new}} \), \( \overline{p}^{\text{min}}_i \geq n^{-k} \), maximal codeword length is above bounded by \( k \log_2 n \). A tighter upper bound is possible, but it is not necessary for our analysis.

If \( L < k \log_2 n \), we can construct an almost-optimal code by applying \textbf{Package-Merge} to the set of weights \( \overline{p}_{\text{new}} \) defined above. If \( L > k \log_2 n \), we
can construct an optimal (not length-restricted) code for weights $p^{new}$. Since
the maximum codeword length in this code does not exceed $k \log_2 n < L$,
this code is also an optimal length-restricted code. An optimal code can
be constructed in time $O(n)$, or in time $O(k \log n)$ with $n / \log n$ processors
(see [BKN02]), if elements are sorted by weight. Since $p^{new}_n < n^k$, elements
can be sorted in $O(n)$ time, or, under certain conditions, in $O(\log n)$ time
with $n / \log n$ processors. Thus an almost-optimal length-restricted code with
error $1/n^k$, such that $k \leq L / \log_2 n$, can be sequentially constructed in linear
time, or in parallel time $O(k \log n)$ with $n / \log n$ processors.

In general case, we can construct an almost-optimal length-restricted code
with error $1/n^k$ in $O(k \log n)$ time with $n$ processors. We sum up the results
of this section in the following

**Theorem 1** A length-restricted code with error $1/n^k$ for any $k > 0$ can be
constructed in $O(k n \log n)$ time. If $k \leq L / \log_2 n$, a length-restricted code
with error $1/n^k$ can be constructed in $O(n)$ time or in $O(k \log n)$ time with
$n / \log n$ CREW processors.

4 A Parallelization of the Package-Merge

We divide elements of $S^j$ into classes $W^j_i$, such that an element $e \in W^j_i$ iff
weight($e$) $\in [2^{i-1}, 2^i)$. We will say that elements $t_1, t_2$ from $S^j$ are siblings if
at the $j$-th stage of the algorithm $t_1$ will be melded with $t_2$.

Suppose that two elements $t_1, t_2$ from $W^j_i$ are siblings. Then $t = meld(t_1, t_2)$
will belong to $W^j_{i+1}$. Therefore after melding elements of $W^j_i$ will be merged
with elements of $W^j_{i+1}$. The only exception may be an element from $W^j_i$
whose sibling does not belong to $W^j_i$. However there is at most one such
exception per class $W^j_i$ and this exception can be inserted into a class $W^j_i$ in
constant time with $|W^j_i|$ processors.

The pseudo-code description of the parallel algorithm is shown on Figure
2. We say $e < a$ for an element $e$ and a number $a$ whenever weight($e$) $< a$.
An array $\text{exc}[i]$ helps us to handle “exceptions” i.e. elements $e \in W^j_i$, such
that sibling($e$) $\not\in W^j_i$. We denote by length($W^j_i$) the number of elements
in $W^j_i$, $m$ is the maximum number of classes $W_i$. Procedure $\text{Meld}(W^j_i)$
melds consecutive pairs of elements in $W^j_i$ thus producing an array of length
$\lfloor |W^j_i|/2 \rfloor$, $\text{first}(W^j_i)$ and $\text{last}(W^j_i)$ denote the first and the last elements of $W^j_i$
respectively.

The bottleneck of this algorithm is function $\text{Merge}$ shown on line 10 of
Figure 2. This function merges $\hat{W}^j_i$ (the sorted list of elements from $W^j_i$
sequentially melded in order of their weight) with the sorted list of elements
from $W^j_{i+1}$. All other operations can be implemented in constant time with
\begin{verbatim}
for j := 1 to L do 
    for \forall l s.t. W_l \neq \emptyset 
        pardo 
            exc[l] := NULL 
            if (\text{siblings}(first(W_j^l)) < 2^{i-1}) 
                exc[l] := meld(first(W_j^l), \text{siblings}(first(W_j^l))) 
            W_j^l := W_j^l \ {\text{first}(W_j^l)} 
            if (\text{siblings}(last(W_j^l)) \geq 2^i) 
                W_j^l := W_j^l \ {\text{last}(W_j^l)} 
            \tilde{W}_j^l := \text{Meld}(W_j^l) 
            W_{j+1}^l := \text{Merge}(\tilde{W}_j^l, W_{i+1}^l) 
            if (exc[l] \neq \text{NULL}) 
                if (exc[l] \geq 2^i) 
                    W_{j+1}^l := \text{Merge}(W_{j+1}^l, \{exc[l]\}) 
                else 
                    W_{j+1}^l := \text{Merge}(W_{j+1}^l, \{exc[l]\}) 
\end{verbatim}

Figure 2: Parallel Implementation of \textbf{Package-Merge}

\textit{n} processors. We will show below how arrays can be merged efficiently in average constant time per iteration. First we will show how this algorithm can be implemented to work in \(O(L)\) time with \(n \log n\) processors. In the next section we will reduce the number of processors to \(n\).

We will use the following notation. Relative weight \(r(t)\) of an element \(t \in W_j^l\) is \(\text{weight}(t) \cdot 2^{-i}\). If elements \(t_1\) and \(t_2\) belong to \(W_j^l\) and \(t\) is the result of melding two elements \(t_1\) and \(t_2\), such that \(r(t_1) > r(e)\) and \(r(t_2) > r(e)\) \((r(t_1) < r(e)\) and \(r(t_2) < r(e)\)), where \(e\) is an element from \(W_{i+1}^l\), then the weight of \(t\) is bigger (smaller) than the weight of \(e\).

We compute for every item \(e \in W_j^l\) and every \(i\), \(l \leq i \leq l + \log n\) the value of \(\text{pred}(e, i) = k\), s.t. \(S^l[k] \in W_j^l\) and \(r(S^l[k]) \leq r(e) < r(S^l[k + 1])\). In other words, \(\text{pred}(e, i)\) is the index of the biggest element in a class \(W_j^l\), whose relative weight is smaller than or equal to \(r(e)\). We also need values of \(\text{pred}'(e, l)\) for all \(e \in S^l\) and all \(l \in [i - \log n, i]\) if \(e \in W_j^l\), where \(\text{pred}'(e, l)\) is the index of the biggest element in \(W_j^l\) whose relative weight is smaller than or equal to \(r(e)\). Obviously, if \(\text{pred}(t, i) = j\) and \(t \in W_j^l\), then there are exactly \(j\) elements in \(S^l\) whose weight is less than or equal to the weight of \(t\). Thus, if \(\text{pred}\) and \(\text{pred}'\) are known \(\text{Merge}(\tilde{W}_j^l, W_{i+1}^l)\) can be performed in constant time.

It remains to show how \(\text{pred}(e, i)\) and \(\text{pred}'(e, i)\) can be computed and updated after each iteration.

7
Figure 3: Computing \( \text{pred}(t, i) \) if \( \text{pred}(t_1, i) \neq \text{pred}(t_2, i) \).

**Statement 1** The values of \( \text{pred}(e, i) \) for \( e \in S^j \) and \( \text{pred}'(e, i) \) for \( e \in S^1 \) can be computed in \( O(\log n) \) time with \( n \) processors.

**Proof**: First we construct arrays \( R_l = W^j_{l_{\log n+1}} \cup W^j_{l_{\log n+2}} \cup \ldots \cup W^j_{l_{\log n+m}} \cup W^j_{l_{\log n+1}} \cup W^j_{l_{\log n+2}} \cup \ldots \cup W^j_{l_{\log n+2\log n}} \) for \( l = 0, \ldots, m/\log n - 1 \) and sort elements of \( R_l \) according to their relative weights. Next we construct arrays \( C_{i,k}, k = 1, \ldots, 2\log n \) so that elements of \( C_{i,k} \) correspond to elements of \( R_l \) and \( C_{i,k}[i] = 1 \) if \( R_{l_{\log n}}[i] \in W^j_{l_{\log n+k}} \) and \( C_{i,k}[i] = 0 \) otherwise. We compute prefix sums \( P_{i,k}[i] = \sum_{m=1}^i C_{i,k}[i] \) for all arrays \( C_{i,k} \). One such prefix sum can be computed in \( O(\log n) \) time with \( |R_l|/\log n \) processors. Since the total number of elements in all arrays \( C_{i,k} \) is \( O(n \log n) \), we can allocate processors in appropriate way in logarithmic time and then compute all prefix sums also in logarithmic time.

The values of \( \text{pred}(e, i) \) can be computed from \( C_{i,k} \) as follows. Suppose \( e \in S^j \). Let \( k' = i - l \log n \). Let \( s \) be the index of \( e \) in \( R_l \) and let \( v \) be \( P_{i,k'}[s] \). Then \( \text{pred}(e, i) \) equals \( v \). Values of \( \text{pred}'(e, i) \) can be computed in the same way.

\( \Box \)

On Fig. 4 an algorithm for updating \( \text{pred} \) and \( \text{pred}' \) after \( \text{Meld}(W^j_i) \) is shown. We use some additional notation on Fig. 4. If \( e \in W^j_i \) then \( \text{class}(e) = l \) and if \( e = \text{melt}(e_1, e_2) \) then \( \text{left}(e) = e_1 \). Suppose that \( \text{pred}'(e, l) = k \) for some \( e \in S^1, S^j[k] \in W^j_i \). Then it is easy to see that the predecessor of \( e \) in \( \tilde{W}^j_i \) is either \( t = \text{meld}(S^j[k], \text{siblings}(S^j[k])) \) or the element preceding \( t \) in \( \tilde{W}^j_i \) (see lines 1-6 of Fig. 4). If \( t = \text{meld}(t_1, t_2) \) we tentatively set \( \text{pred}(t, i) = \text{pred}(t_1, i) \) (lines 7-9). The value of \( \text{pred}(t, i) \) is correct only if \( \text{pred}(t_1, i) = \text{pred}(t_2, i) \). If \( \text{pred}(t_1, i) = p_1, \text{pred}(t_2, i) = p_2, \) and \( p_1 \neq p_2 \), then \( \text{pred}(t, i) = p_3 \) such that \( p_1 \leq p_3 \leq p_2 \). Otherwise the correct value of \( \text{pred}(t_1, i) \) can be found as follows. Let \( k \) be the index of \( t \) in \( S^j \). It is
for \( \forall e \in S^1 \) pardo
   for class\( (e) - \log n \leq l \leq \text{class}(e) \) pardo
      \[
      c := \left\lfloor \frac{\text{pred}'(e, l)}{2} \right\rfloor
      \]
      if \( r(e) < r(S^j[c]) \)
      \[
      c := c - 1
      \]
      \[
      \text{pred}'(e, l) := c
      \]
   for \( \forall e \in S^j \) pardo
   for class\( (e) \leq l \leq \text{class}(e) + \log n \) pardo
   \[
   \text{pred}(e, l) := \text{pred}(\text{left}(e), l)
   \]
   for \( 1 \leq s \leq |S^1| \) pardo
      \[
      k := \text{pred}'(S^1[s], l)
      \]
      if \( r(S^j[k]) < r(S^1[s]) \) AND
      \[
      (r(S^1[s + 1]) > r(S^j[k + 1]))
      \]
      \[
      \text{pred}(S^j[k + 1], l) := s
      \]
      if \( r(S^j[k]) = r(S^1[s]) \) AND
      \[
      (r(S^1[s + 1]) > r(S^j[k]))
      \]
      \[
      \text{pred}(S^j[k], l) := s
      \]

Figure 4: Recomputing \( \text{pred}(e, i) \) and \( \text{pred}'(e, i) \) after \( \text{Meld}(W^j_i) \)

It is easy to see that for \( \forall p \ p_1 < p \leq p_3 \) \( \text{pred}'(S^1[p], i) \) is either \( k \) or \( k - 1 \). If \( \text{pred}(t, i) = p_3 \) and \( r(S^1[p_3]) < r(t) \), then \( \text{pred}'(S^1[p_3], i) = k - 1, r(S^1[p_3]) > r(S^j[k - 1]) \), and \( r(S^1[p_3 + 1]) > r(S^1[k]) \) (see Fig. 3). If \( \text{pred}(t, i) = p_3 \) and \( r(S^1[p_3]) = r(t) \), then \( \text{pred}'(S^1[p_3], i) = k, r(S^1[p_3]) = r(S^j[k]), \) and \( r(S^1[p_3 + 1]) > r(S^1[k]) \). We check for this condition on lines 10-16 of Fig. 4 and compute the correct values of \( \text{pred}(t, i) \) in case \( \text{pred}(t_1, i) \neq \text{pred}(t_2, i) \).

When the elements of \( W^j_i \) are melded and predecessor values \( \text{pred}(e, i) \) are recomputed \( \text{pred}(W^j_i[t], i - 1) \) equals to the number of elements in \( W^1_{i-1} \) that are smaller than or equal to \( W^j_i[t] \) and \( \text{pred}'(W^1_{i-1}[t], i) \) equals to the number of elements in \( W^j_i \) that are smaller than or equal to \( W^1_{i-1}[t] \). Therefore indices of all elements in the merged array can be computed in constant time. When \( S^j \) and \( S^1 \) are merged \( \text{pred} \) and \( \text{pred}' \) can be recomputed in constant time.

In this way we can perform \( \log n \) iterations of \textbf{Package-Merge} in constant time per iteration. After this we have to compute \( \text{pred}(e, i) \) and \( \text{pred}'(e, i) \) for \( S^1 \) and \( S^{\log n} \) as described in Statement 1. Then we will be able to perform the next \( \log n \) iterations in the same way. Therefore every \( \log n \) iterations of \textbf{Package-Merge} can be performed in \( O(\log n) \) time with \( n \log n \) processors.
and we have proven

**Theorem 2** The algorithm Package-Merge can be implemented in $O(L)$ time with $n \log n$ processors on CREW PRAM.

5 An $O(nL)$ work algorithm

The algorithm described in the previous section requires $n \log n$ processors to work in $O(L)$ time, because at every step $2n \log n$ values of $\text{pred}$ and $\text{pred}'$ must be recomputed. But the number of processors can by reduced by a logarithmic factor, since not all values $\text{pred}$ and $\text{pred}'$ are necessary at each iteration. In fact, if we know values of $\text{pred}(e, i)$ for the next class $W^i_1$, if $e \in W^j_{i-1}$ for all $e \in S^j$ and values of $\text{pred}'(e, i)$ for the previous class $W^j_1$, if $e \in W^j_{i+1}$ for all $e \in S^j$ then merging can be performed in constant time. Therefore we will use functions $\text{pred}$ and $\text{pred}'$ instead of $\text{pred}$ and $\text{pred}'$ such that this information is available at each iteration, but the total number of values in $\text{pred}$ and $\text{pred}'$ is limited by $O(n)$. We must also be able to recompute values of $\text{pred}$ and $\text{pred}'$ in constant time after each iteration.

For an array $R$ we will denote by $\text{sample}_k(R)$ a subarray of $R$ that consists of every $2^k$-th element of $R$. We define $\text{pred}(e, i)$ for $e \in W^j_i$ as index of the biggest element $\hat{e}$ in $\text{sample}_{i-1}(W^j_i)$, such that $r(\hat{e}) \leq r(e)$. Besides that, we maintain the values of $\text{pred}(e, i)$ only for $e \in \text{sample}_{i-1}(W^j_i)$. In other words, for every $2^{i-1}$-th element of $W^j_i$ we know its predecessor with precision up to $2^{i-1}$ elements. We define $\text{pred}'(e, l)$ for $e \in \text{sample}_{i-1}(W^j_i)$ as the index of the biggest element $\hat{e}$ in $\text{sample}_{i-1}(W^j_i)$, such that $r(\hat{e}) \leq r(e)$. Obviously, the total number of values in $\text{pred}$ and $\text{pred}'$ is $O(n)$.

After procedure Meld predecessors must be recomputed and “refined”. That is, for every $e \in \text{sample}_{i-1}(W^j_i)$ its predecessor from $\text{sample}_{i-1}(W^j_i)$ is known. However $W^j_i$ will be merged with $W^j_{i+1}$ into $W^j_{i+1}$. Therefore for $e \in \text{sample}_{i-1}(W^j_i)$ its predecessor from $\text{sample}_{i-1}(W^j_i)$ must be computed. Recomputing and “refining” $\text{pred}$ and $\text{pred}'$ after Meld is similar in spirit to the algorithm described in the previous section. A detailed description will be given in the full version of this paper.

Using the values of $\text{pred}$ and $\text{pred}'$, we can merge $S^1$ and $S^j$ in a constant time.

Thus we can perform $\log n$ iterations of Package-Merge in logarithmic time. Combining this fact with Statement 1 we get

**Theorem 3** The algorithm Package-Merge can be implemented in $O(L)$ time with $n$ CREW processors.
Corollary 1 An optimal length-restricted code with maximum codeword length \( L \) can be constructed in \( O(L) \) time with \( n \) CREW processors. An almost optimal length-restricted code with maximum codeword length \( L \) and error \( 1/n^k \) can be constructed in \( O(k \log n) \) time with \( n \) CREW processors.

6 Conclusion

We described an algorithm for the construction of almost-optimal length-restricted codes with error \( 1/n^k \) for any \( k > 0 \) that works in \( O(n \log n) \) time. We show that this algorithm can be parallelized to work in time \( O(\log n) \) with \( n \) CREW processors. We also showed that an almost-optimal length-restricted code with error \( 1/n^k \) for any \( k \leq L/\log^k n \) can be constructed in \( O(kn) \) time or in \( O(k \log n) \) time with \( n/\log n \) CREW processors. Our algorithms use only comparison, addition, and bit shift operations.

Acknowledgements

We thank Piotr Berman and Larry Larmore for stimulating remarks and discussions.

References


