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 $k \leq L/\log_\Phi n$, 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 $\bar{p} = p_1, p_2, \ldots, p_n$ respectively. A code with lengths $\mathcal{L} = l_1, l_2, \ldots, l_n$ 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 $\bar{p} = p_1, p_2, \ldots, p_n$ if $Length(\mathcal{L}, \bar{p}) = \sum l_i p_i$ is minimal among all prefix-free codes. A code $\mathcal{L}$ is a *minimum redundancy length-restricted code* if $Length(\mathcal{L}, \bar{p})$ 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 $O(n 2^{O(\sqrt{\log L \log \log n})})$ 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.

Milidiu, Pessoa and Laber [MPL98] described an algorithm for length-restricted codes with error $1/F_{L - \lceil \log(n + \lceil \log n \rceil - L) \rceil + 1}$, 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 $wpl(T') \le wpl(T) + n^{-k}$ 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 [BKN02].

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 *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 $k \leq L/\log_\Phi n$ 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 $t_i^{j+1} = meld(t_{2i}^j, t_{2i+1}^j)$ and merging the list of new elements with a copy of the list S^1. Here t_i^j denotes the i-th item in the list S^j. An operation $meld(t', t'')$ 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^L. On Figure 1 we show how the algorithm **Package-Merge** works on the set of items with weights $\overline{p} = 1, 1, 3, 7, 11, 15$ for $L = 4$. The resulting code consists of codewords with lengths $\mathcal{L} = 4, 4, 3, 2, 2, 2$ respectively.

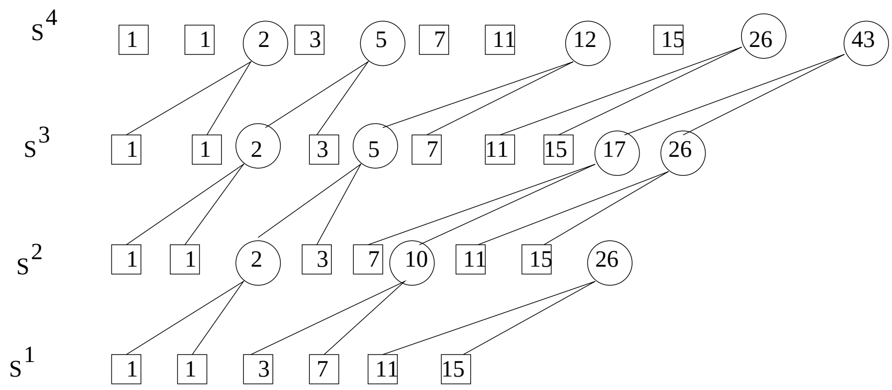

Figure 1: An example of **Package-Merge** for $L = 4$. Elements of S^1 are marked by squares, elements resulting from melding elements on the previous list are marked by circles.

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* $pred(e, i)$ 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 $AvLen(\mathcal{L}, \overline{p}) = Length(\mathcal{L}, \overline{p})/P$, where $P = \sum_{i=1}^{n} p_i$. We say that a length-restricted code $\mathcal{L}$ is almost-optimal with error ϵ, if $AvLen(\mathcal{L}, \overline{p}) \le AvLen(\mathcal{L}', \bar{p}) + \epsilon$ for all length-restricted codes $\mathcal{L}'$. Below we show how an almost-optimal length-restricted code with error $\frac{1}{n^k}$ 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

$O(n^k)$, using a code with error $1/n^k$ 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 $O(n^{k'})$ with $k' < k$, then a code with error $1/n^k$ is optimal.

To achieve this goal, we construct an optimal code for the "quantized" set of weights $\overline{p^{new}} = p_1^{new}, p_2^{new}, \ldots, p_n^{new}$. Before we define p_i^{new}, consider weights p_i^n, where $p_i^n = \lceil p_i/(\lceil P/n^k \rceil)\rceil(\lceil P/n^k \rceil)$ and $P = \sum_{i=1}^n p_i$. For any code $\mathcal{L}$, $\sum l_i p_i^n \le \sum l_i p_i + (P/n^k)\sum l_i \le \sum l_i p_i + P \cdot n^{-k+2}$, since $l_i \le n$. Hence $AvLen(\mathcal{L}, \overline{p}^n) \le Length(\mathcal{L}, \overline{p}^n)/P \le AvLen(\mathcal{L}, \overline{p}) + n^{-k+2}$.

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

Observe that instead of division by $\lceil P/n^k \rceil$ we can set $p_i^{new} = \lceil p_i/2^m \rceil$ for m such that $\lceil P/n^k \rceil \le 2^m \le 2\lceil P/n^k \rceil$. 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 $2j - i \le n$. Edge (i, j) has weight $w(i, j) = \sum_{k=1}^{2j-i} p_k$. Since $p_i^{new} < n^k$ 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 **Package-Merge** approach and "quantized" weights p_i^{new}. In [B93] it was shown that maximal codeword length of a Huffman code does not exceed $\min(\lceil -\log_\Phi p'_{\min} \rceil, n-1)$, where $p'_{\min} = p_{\min}/P$ is the minimal normalized weight. Since for the set of weights $\overline{p}^{new}$ $p'_{\min} \ge n^{-k}$, maximal codeword length is above bounded by $k \log_\Phi n$. A tighter upper bound is possible, but it is not necessary for our analysis.

If $L < k \log_\Phi n$, we can construct an almost-optimal code by applying **Package-Merge** to the set of weights $\overline{p^{new}}$ defined above. If $L > k \log_\Phi n$, we can construct an optimal (not length-restricted) code for weights $\overline{p^{new}}$. Since the maximum codeword length in this code does not exceed $k \log_\Phi n < L$, 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 $p_i^{new} < n^k$, 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 $k \le L/\log_\Phi n$, 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* $O(kn \log n)$ *time. If* $k \le L/\log_\Phi n$, *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_l^j, such that an element $e \in W_l^j$ iff $weight(e) \in [2^{l-1}, 2^l)$. 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_l^j are siblings. Then $t = meld(t_1, t_2)$ will belong to W_{l+1}^{j+1}. Therefore after melding elements of W_l^j will be merged with elements of W_{l+1}^1. The only exception may be an element from W_l^j whose sibling does not belong to W_l^j. However there is at most one such exception per class W_l^j and this exception can be inserted into a class W_l^j in constant time with $|W_l^j|$ processors.

The pseudocode 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 $exc[l]$ helps us to handle "exceptions" i.e. elements $e \in W_l^j$, such that $sibling(e) \notin W_l^j$. We denote by $length(W_l^j)$ the number of elements in W_l^j, m is the maximum number of classes W_i. Procedure $Meld(W_l^j)$ melds consecutive pairs of elements in W_l^j thus producing an array of length $|W_l^j|/2$, $first(W_l^j)$ and $last(W_l^j)$ denote the first and the last elements of W_l^j respectively.

The bottleneck of this algorithm is function Merge shown on line 10 of Figure 2. This function merges $\tilde{W}_l^j$ (the sorted list of elements from W_l^j sequentially melded in order of their weight) with the sorted list of elements from W_{l+1}^1. All other operations can be implemented in constant time with

```
for j := 1 to L do
   for ∀ l s.t. W_l ≠ ∅ pardo
      exc[l] := NULL
      if (sibling(first(W_l^j)) < 2^{l-1})
         exc[l] := meld(first(W_l^j), sibling(first(W_l^j)))
         W_l^j := W_l^j \ {first(W_l^j)}
      if (sibling(last(W_l^j)) ≥ 2^l)
         W_l^j := W_l^j \ {last(W_l^j)}
      W̃_l^j := Meld(W_l^j)
      W_{l+1}^{j+1} := Merge(W̃_l^j, W_{l+1}^1)
      if (exc[l] ≠ NULL)
         if (exc[l] ≥ 2^l)
            W_l^{j+1} := Merge(W_l^{j+1}, {exc[l]})
         else
            W_{l-1}^{j+1} := Merge(W_{l-1}^{j+1}, {exc[l]})
```

Figure 2: Parallel Implementation of **Package-Merge**

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 $t \in W_l^i$ is $weight(t) \cdot 2^{-l}$. If elements t_1 and t_2 belong to W_l^j 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_{l+1}^1, then the weight of t is bigger (smaller) than the weight of e.

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

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

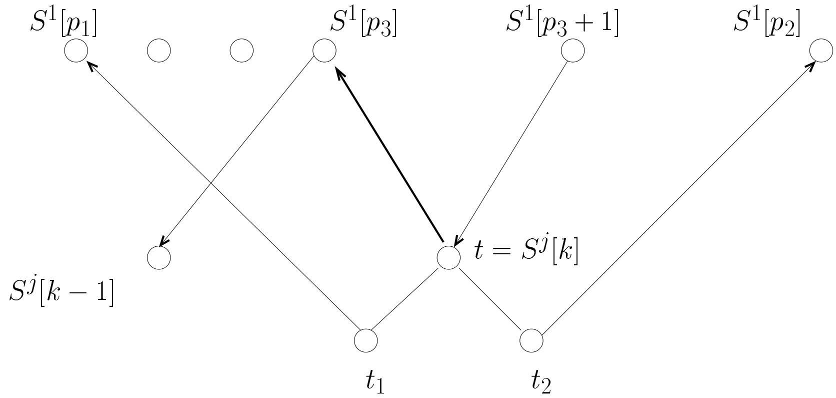

Figure 3: Computing $pred(t,i)$ if $pred(t_1,i) \neq pred(t_2,i)$.

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

Proof: First we construct arrays $R_l = W^j_{l\log n+1} \cup W^j_{l\log n+2} \cup \ldots \cup W^j_{l\log n+\log n} \cup W^1_{l\log n+1} \cup W^1_{l\log n+2} \cup \ldots \cup W^1_{l\log n+2\log n}$ for $l = 0, \ldots, m/\log n - 1$ and sort elements of R_l according to their relative weights. Next we construct arrays $C_{l,k}$, $k = 1, \ldots, 2\log n$ so that elements of $C_{l,k}$ correspond to elements of R_l and $C_{l,k}[i] = 1$ if $R_{l\cdot\log n}[i] \in W^1_{l\log n+k}$ and $C_{l,k}[i] = 0$ otherwise. We compute prefix sums $P_{l,k}[i] = \sum_{m=1}^{i} C_{l,k}[i]$ for all arrays $C_{l,k}$. 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 $C_{l,k}$ 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 $pred(e,i)$ can be computed from $C_{l,k}$ as follows. Suppose $e \in W^j_l$. Let $k' = i - l\log n$. Let s be the index of e in R_l and let v be $P_{l,k'}[s]$. Then $pred(e,i)$ equals to v. Values of $pred'(e,i)$ can be computed in the same way.
□

On Fig. 4 an algorithm for updating $pred$ and $pred'$ after $Meld(W^j_l)$ is shown. We use some additional notation on Fig. 4. If $e \in W^j_l$ then $class(e) = l$ and if $e = meld(e_1, e_2)$ then $left(e) = e_1$. Suppose that $pred'(e,l) = k$ for some $e \in S^1$, $S^j[k] \in W^j_l$. Then it is easy to see that the predecessor of e in $\tilde{W}^j_l$ is either $t = meld(S^j[k], sibling(S^j[k]))$ or the element preceding t in $\tilde{W}^j_l$ (see lines 1-6 of Fig. 4). If $t = meld(t_1, t_2)$ we tentatively set $pred(t,i) = pred(t_1,i)$ (lines 7-9). The value of $pred(t,i)$ is correct only if $pred(t_1,i) = pred(t_2,i)$. If $pred(t_1,i) = p_1$, $pred(t_2,i) = p_2$, and $p_1 \neq p_2$, then $pred(t,i) = p_3$ such that $p_1 \leq p_3 \leq p_2$. Otherwise the correct value of $pred(t_1,i)$ can be found as follows. Let k be the index of t in S^j. It is

```
1    for ∀e ∈ S^1 pardo
2      for class(e) − log n ≤ l ≤ class(e) pardo
3        c := ⌈pred'(e, l)/2⌉
4        if (r(e) < r(S^j[c]))
5          c := c − 1
6        pred'(e, l) := c
7    for ∀e ∈ S^j pardo
8      for class(e) ≤ l ≤ class(e) + log n pardo
9        pred(e, l) := pred(left(e), l)
10   for 1 ≤ s ≤ |S^1| pardo
11     for class(S^1[s]) − log n ≤ l ≤ class(S^1[s]) pardo
12       k := pred'(S^1[s], l)
13       if (r(S^j[k]) < r(S^1[s])) AND
           (r(S^1[s + 1]) > r(S^j[k + 1]))
14         pred(S^j[k + 1], l) := s
15       if (r(S^j[k]) = r(S^1[s])) AND
           (r(S^1[s + 1]) > r(S^j[k]))
16         pred(S^j[k], l) := s
```

Figure 4: Recomputing $pred(e, i)$ and $pred'(e, i)$ after $Meld(W_l^j)$

easy to see that for $\forall\ p\ p_1 < p \le p_3\ pred'(S^1[p], i)$ is either k or $k-1$. If $pred(t, i) = p_3$ and $r(S^1[p_3]) < r(t)$, then $pred'(S^1[p_3], i) = k-1$, $r(S^1[p_3]) > r(S^j[k-1])$, and $r(S^1[p_3+1]) > r(S^1[k])$ (see Fig. 3). If $pred(t, i) = p_3$ and $r(S^1[p_3]) = r(t)$, then $pred'(S^1[p_3], i) = k$, $r(S^1[p_3]) = r(S^j[k])$, 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 $pred(t, i)$ in case $pred(t_1, i) \ne pred(t_2, i)$.

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

In this way we can perform $\log n$ iterations of **Package-Merge** in constant time per iteration. After this we have to compute $pred(e, i)$ and $pred'(e, i)$ 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 **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 $pred$ and $pred'$ must be recomputed. But the number of processors can by reduced by a logarithmic factor, since not all values $pred$ and $pred'$ are necessary at each iteration. In fact, if we know values of $pred(e,i)$ for the next class W_i^1, if $e \in W_{i-1}^j$ for all $e \in S^j$ and values of $pred'(e,i)$ for the previous class W_i^j, if $e \in W_{i+1}^1$ for all $e \in S^1$ then merging can be performed in constant time. Therefore we will use functions $\overline{pred}$ and $\overline{pred'}$ instead of $pred$ and $pred'$ such that this information is available at each iteration, but the total number of values in $\overline{pred}$ and $\overline{pred'}$ is limited by $O(n)$. We must also be able to recompute values of $\overline{pred}$ and $\overline{pred'}$ in constant time after each iteration.

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

After procedure $Meld$ predecessors must be recomputed and "refined". That is, for every $e \in sample_{i-l-1}(\tilde{W}_l^j)$ its predecessor from $sample_{i-l-1}(W_i^1)$ is known. However $\tilde{W}_l^j$ will be merged with W_{l+1}^1 into W_{l+1}^{j+1}. Therefore for $e \in sample_{i-l-2}(\tilde{W}_l^j)$ its predecessor from $sample_{i-l-2}(W_i^1)$ must be computed. Recomputing and "refining" $pred$ and $pred'$ 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 $\overline{pred}$ and $\overline{pred'}$, 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 $k \leq L/\log_\Phi n$ 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

[AST94] Aggarwal, A., Schieber, B., Tokuyama, T., *Finding a Minimu-Weight k-Link Path in Graphs with the Concave Monge Property*, Journal on Discrete & Computational Geometry 12 (1994), pp. 263–280.

[BKN02] Berman, P., Karpinski, M., Nekritch, Y., *Approximating Huffman Codes in Parallel*, Proc. 29th ICALP (2002).

[B93] Buro, M., *On the Maximum Length of Huffman Codes*, Information Processing Letters 45(1993), pp. 219-223.

[G74] Garey, M., *Optimal binary search trees with restricted maximal depth*, SIAM Journal on Computing 3 (1974), pp. 101–110.

[KMT95] Katajainen,J., Moffat, A. , Turpin, A. , *A Fast and Space-economical Algorithm for Length-Limited Coding*, Proc. International Symposium on Algorithms and Computation (1995), pp. 12-21.

[KP96] Kirkpatrick, D., Przytycka, T., *Parallel Construction of Binary Trees with Near Optimal Weighted Path Length,* Algorithmica (1996), pp. 172–192.

[L87] Larmore, L., *Height-restricted optimal binary trees*, SIAM Journal on Computing 16 (1987), pp. 1115–1123.

[LH90] Larmore, L., Hirschberg, D., *A Fast Algorithm for Optimal Length-Limited Huffman Codes,* Journal of the ACM 37(3) (1990), pp. 464–473.

[LPW93] Larmore, L. L., Przytycka, T.,Rytter, W., *Parallel Construction of Optimal Alphabetic Trees*, Proc. 5th ACM Symposium on Parallel Algorithms and Architectures (1993), pp. 214–223.

[LP95] Larmore, L., Przytycka, T., *Constructing Huffman trees in parallel*, SIAM Journal on Computing 24(6) (1995), pp. 1163–1169.

[LM02] Liddell, M., Moffat, A., *Incremental Calculation of Minimum-Redundancy Length-Restricted Codes,* Proc. Data Compression Conference (2002), pp. 182-191.

[MK95] Moffat, A., Katajainen, J., *In-Place Calculation of Minimum-Redundancy Codes*, Proc. WADS (1995), pp. 393-402

[MPL98] Milidiu, R. L., Pessoa, A. A., Laber, E.S., *In-Place Length-Restricted Prefix Coding*, Proc. String Processing and Information Retrieval: A South American Symposium (1998), pp. 50-59.

[MPL99] Milidiu, R. L., Pessoa, A. A., Laber, E.S., *Efficient Implementation of the WARM-UP Algorithm for the Construction of Length-Restricted Prefix Codes*, Proc. ALENEX(1999), pp. 1-17.

[MPL99a] Milidiu, R. L., Pessoa, A. A., Laber, E.S., *A Work Efficient Parallel Algorithm for Constructing Huffman Codes,* Proc. Data Compression Conference (1999), pp. 277-286.

[S95] Schieber, B., *Computing a minimum-weight k-link path in graphs with concave Monge property,* Proc. the 6th Annual Symposium on Discrete Algorithms (1995), pp. 405–411.

[vL76] van Leeuwen, J., *On the construction of Huffman trees,* Proc. 3rd Int. Colloquium on Automata, Language s and Programming (1976), pp. 382–410.