Electronic Colloquium on Computational Complexity, Report No. 29 (2002)

Parallel Construction of Minimum Redundancy Length-Limited Codes

Marek Karpinski * Yakov Nekrich †

Abstract. This paper presents new results on parallel constructions of the length-limited prefix-free codes with the minimum redundancy. We describe an algorithm for the construction of length-limited codes that works in $O(L)$ time with n processors for L the maximal codeword length. We also describe an algorithm for a construction of *almost optimal length-limited codes* that works in $O(\log n)$ time with n processors. This is an optimal parallelization of the best known up to date sequential algorithm.

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. We say that an integer list $\mathcal{L} = l_1, l_2, \ldots, l_n$ is a *prefix-free code* if $\sum 2^{-l_i} \leq 1$. A (prefix-free) code is *length-limited* for some integer L if $l_i \leq L$ for all $1 \leq i \leq n$.

A code 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-limited code* if $Length(\mathcal{L}, \bar{p})$ is minimal among all length-limited prefix-free codes. The problem of length-limited coding is motivated by practical implementations of coding algorithms since every codeword must fit into a machine register of fixed width.

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]). However the construction of a length-limited minimum redundancy code requires more time. Garey [G74] has described an algorithm for constructing length-limited codes

*Supported in part by DFG grants, DIMACS, PROCOPE Project 31022, and IST grant 14036 (RAND-APX). Email marek@cs.uni-bonn.de.

†Work partially supported by PROCOPE Project 31022, and IST grant 14036 (RAND-APX). Email yasha@cs.uni-bonn.de.

ISSN 1433-8092

that runs in $O(n^2L)$ 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 problem reduction due to Larmore and Przytycka (see [LP95]) Schieber [S95] has given an $O(n2^{O(\sqrt{\log L \log\log n})})$ algorithm for this problem.

The fastest n-processor algorithm for the construction of Huffman codes is due to Larmore and Przytycka [LP95]. Their algorithm, based on reduction of Huffman tree construction problem to the *concave least weight subsequence* problem runs in $O(\sqrt{n}\log n)$ time. 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 $wpl(T') \leq wpl(T) + n^{-k}$ for a fixed error parameter k (assuming $\sum w_i = 1$). They presented efficient parallel algorithms for the construction of almost optimal codes, that work in $O(k\log n\log^* n)$ time and 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 recently improved in [BKN02].

A problem related to the problem discussed in this paper is the construction of optimal alphabetic codes. In case of the alphabetic codes we have an additional limitation that the codeword for e_i precedes the codeword for e_j in lexicographic order for all $i < j$. The best known NC algorithm constructs an optimal alphabetic code in time $O(log^3 n)$ with $n^2\log n$ processors (see [LPW93]).

In this paper we consider a parallel algorithm for the construction of minimum-redundancy length-limited codes that is based on the **Package-Merge** algorithm of Larmore and Hirschberg [LH90]. Our algorithm constructs a length-limited code in $O(L)$ time with n processors on a CREW PRAM. We also describe an algorithm for the construction of length-limited codes that works with an error $1/n^k$ in $O(k\log n)$ time with n processors on a CREW PRAM. The last algorithm gives us an optimal speed-up compared to the best known sequential algorithm.

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 trees with the copy of the list S^1. Here t_i^j denotes the i-th item in the list S^j. An operation $meld(t_1, t_2)$ creates a new tree t with two sons t_1 and t_2 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

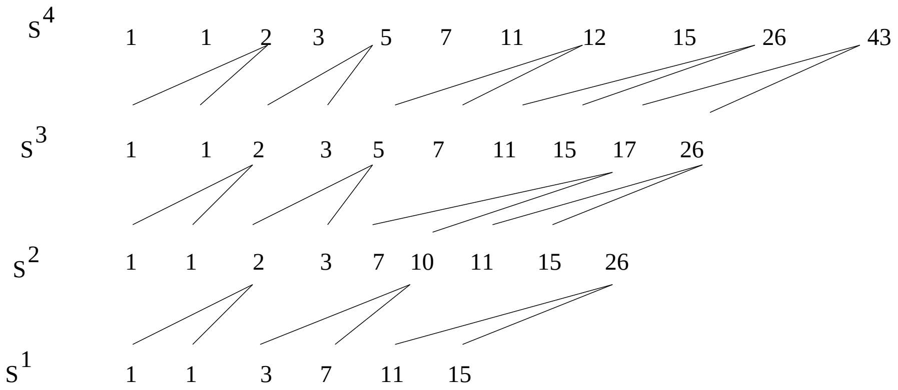

Figure 1: Example of **Package-Merge** for $L = 4$

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.

When the list S^L is constructed we can easily compute depths of all elements in an optimal code. Indeed S^L consists of $2n-1$ trees and these trees have a total number of 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 the rest of this paper we describe parallel algorithms for the construction of S^L. We will see in the next section that the most time-consuming operation is merging of two lists. We show how after a certain pre-processing stage a logarithmic number of merge operations can be performed in a logarithmic time. During this pre-processing stage we compute the *predecessor values* $pred(e, i)$ for every element e and every list S^i. These values can be efficiently re-computed after a meld operation and they will also allow us to merge arrays in a constant time.

2 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.

1: for $l := 1$ to m pardo
2: $exc[l] := NULL$
3: if $(sibling(first(W_l^j)) < 2^{l-1})$ AND $(sibling(first(W_l^j)) + first(W_l^j) < 2^l)$
4: $exc[l] := meld(first(W_l^j), sibling(first(W_l^j)))$
5: $first(W_l^j) := next(first(W_l^j))$
6: if $(exc[l] \neq NULL)$
7: $length(W_l^j) := length(W_l^j) - 1$
8: $length(W_{l+1}^j) := length(W_{l+1}^j) - 1$
9: for $i := 1$ to $length(W_l^j)/2$ pardo
10: $t_i^j := meld(t_{2i}^j, t_{2i+1}^j)$
11: $W_{l+1}^{j+1} := merge(W_l^j[1, .., length(W_l^j)/2], W_{l+1}^1)$
12: $W_l^{j+1} := merge(W_l^{j+1}, exc[l])$

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

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 belongs to W_{l-1}^j. However there is at most one such exception in every class W_l^j and this exception can be inserted into a class W_l^j in a constant time with $|W_l^j|$ processors.

The pseudocode description of the parallel algorithm is shown on Figure 2. For simplicity we say $e < a$ for an element e and a number a whenever $weight(e) < a$. An array $exc[l]$ contains pointers to "exceptions" i.e. to elements $e \in W_l^j$, such that $sibling(e) \in W_{l-1}^j$ and $meld(e, sibling(e)) < 2^l$. We denote by $length(W_l^j)$ the number of elements in W_l^j, m is the maximum number of classes W_i.

The bottleneck of this algorithm is the merge operation shown on the line 12 of the Figure 2. This operation merges 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 a constant time. We will show below how arrays can be merged efficiently in an 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. Then 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 $weight(t) \cdot 2^{-l}$. We observe that 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 also compute for every item $e \in W_l^j$ and every i, $l \leq i \leq l + \log n$ the value of $pred(e, i) = W_i^1[k]$, s.t. $r(W_i^1[k]) < r(e) < r(W_i^1[k+1])$. In other words, $pred(e, i)$ is the biggest element in a class W_i^1, whose relative weight is smaller than or equal to $r(e)$.

Obviously, if $pred(t_1, i) = pred(t_2, i) = W_i^1[j]$ and $t_1, t_2 \in W_{i-1}^l$, then $t = meld(t_1, t_2)$ must be placed between $W_i^1[j]$ and $W_i^1[j+1]$ in W_i^{l+1}. Also if $t_1 \in W_i^l$ and $pred(t_1, i) = t_2$ then t_1 must be placed on the next position after t_2 in W_i^l.

Now we will show how $pred(e, i)$ can be computed and updated after each iteration.

Statement 1 *The values of $pred(e, i)$ for $e \in S^j$ 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 $P_{l,k}$ for all arrays $C_{l,k}[i]$. One such prefix sum can be computed in $O(\log n)$ time with $|R_l|/\log n$ processors. Therefore we can allocate processors in appropriate way in a logarithmic time and then compute all prefix sums also in a logarithmic time.

The values of $pred(e, i)$ can be computed from $C_{l,k}$ as follows. Suppose $e \in W_l^j$. Let $l' = l/\log n$ and $k' = i - l' \log n$ Let s be the index of e in $R_{l'}$ and let v be $P_{l',k'}[s]$. Then $pred(e, i)$ equals to $W_i^1[v]$.
□

We will 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 biggest element in W_l^j whose relative weight is smaller than that of e. These values can also be computed in $O(\log n)$ time with n processors.

Next we show how the values of $pred(e, i)$ can be updated after the operation *meld*. We will denote by $pos(t)$ position of an element t in its class W_l^j.

```
1:   for a < m, b ≤ |W_a^1| pardo
2:       s := W_a^1[b]
3:       for a < l ≤ a + log n pardo
4:           temp^l[s] := ⌈pos(pred'(s, l))/2⌉

5:   for i < m, c ≤ |W_i^j|/2 pardo
6:       s := meld(W_i^j[2c − 1], W_i^j[2c])
7:       pos(s) := c
8:       W_i[c] := s

9:   for a < m, b ≤ |W_a^1| pardo
10:      s := W_a^1[b]
11:      for a < l ≤ a + log n pardo
12:          c := temp^l[s]
12:          if r(W_l^j[c]) > r(W_a^1[b])
13:              c := c − 1
14:              if r(W_a^1[b + 1]) > r(W_l^j[c + 1])
15:                  pred(W_l^j[c + 1], a) := s
16:          pred'(s, l) := W_l^j[c]
```

Figure 3: Melding operation

For simplicity we will say that $t_1 > t_2$ when $weight(t_1) > weight(t_2)$.

First we store the tentative new value of $pred'(e, i)$ for all $e \in S^1$ in an array $temp$ (lines 1-4 of Figure 3). The values stored in $temp[]$ differ from the correct values by at most 1.

Next we meld the elements and change the values of $weight(s)$ and $pos(s)$ for all $s \in W_i$ (lines 5-8 of Figure 3).

Then we check whether the values of $pred'(s, i)$ for $s \in S^1$ are the correct ones. In order to achieve this we compare the relative weight of the tentative predecessor with the relative weight of s. If the relative weight of s is smaller, $pred(s, i)$ is assigned to the previous element of W_i. (lines 9-14 of Figure 3). In lines 15 and 16 we check whether the predecessors of elements in W_i^j have changed.

If the number of elements in W_i^j is odd then the last element of W_i must be inserted into W_i^j. With $|W_i^j|$ processors we can perform this operation in a constant time. We can also correct values of $pred(e,i)$ in a constant time with a linear number of processors.

When the elements of W_i^j are melded and predecessor values $pred(e,i)$ are recomputed $pos(pred(W_i^j[t], i-1))$ equals to the number of elements in W_{i-1}^1 that are smaller than or equal to $W_i^j[t]$. Analogically $pos(pred'(W_{i-1}^1[t], i))$ 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 a constant time.

After melding of elements from S^j every element of W_l^1 has two predecessors in classes $i = l+1, \ldots, l+\log n$. We can find the new predecessors of an element e by comparing $pred'(e,i)$ and $pred'(e,i-1)$.

In this way we can perform $\log n$ iterations of **Package-Merge** in a 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

Theorem 1 *The algorithm* **Package-Merge** *can be implemented in $O(L)$ time with $n \log n$ processors.*

3 An 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 $pred'$ may be modified. In this section we show how the total work can by reduced by a logarithmic factor.

The main idea of our modified algorithm is that 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 a 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 a 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 the biggest element $\tilde{e}$ in $sample_{l-i-1}(W_i^1)$, such that $r(\tilde{e}) \leq r(e)$. Besides that we maintain the values of $\overline{pred}(e,i)$ only for $e \in sample_{l-i-1}(W_l^j)$. In other words for every 2^{l-i-1}-th element of W_l^j we know its predecessor up to 2^{l-i-1} elements. $\overline{pred'}(e,i)$ is defined in the same way. Obviously the total number of values in $\overline{pred}$ and $\overline{pred'}$ is $O(n)$.

Now we will show how $\overline{pred}$ and $\overline{pred'}$ can be recomputed after elements in W_l^j are melded. The number of pairs (e,i) for which values $\overline{pred}(e,i)$ and $\overline{pred'}(e,i)$ must be computed is $O(n)$ therefore we can presume that one processor is assigned to every such pair.

Consider an arbitrary pair (e,i), $e \in W_l^j$. First the value $\overline{pred}(e,i)$ is known, but the value of $\overline{pred}(s,i)$, where $s = sibling(e)$ may be unknown. Let e_p be the element preceding e and e_n be the element following e in $sample_{l-i}(W_l^j)$. If s follows e then correct value of $\overline{pred}(s,i)$ is between $\overline{pred}(e,i)$ and $\overline{pred}(e_n,i)$. If s precedes e in W_l^j then the correct value of s is between $\overline{pred}(e_p,i)$ and $\overline{pred}(e,i)$. Correct values of $\overline{pred}(s,i)$ can be computed in a constant time with $|sample_{l-i}(W_i^1)|$ processors.

When elements from W_l^j are melded, the new elements will belong to W_{l+1}^{j+1}. Now we have to compute $\overline{pred}(e,i)$ in $sample_{i-l-2}(W_i^1)$ for every 2^{i-l-2}-th element of W_{l-1}^{j+1}. Suppose $\overline{pred}(e,i) = W_l^1[k \cdot 2^{i-l-1}]$. We can find the new value of $\overline{pred}(e,i)$ by comparing $r(e)$ with $r(W_l^1[k \cdot 2^{i-l-1} + 2^{i-l-2}])$. When the correct values of $\overline{pred}(e,i)$ $e \in sample_{i-l-1}(W_l^j)$ are known we can compute $\overline{pred}(e,i)$ for all e from $sample_{i-l-2}(W_l^j)$. Let e be a new element in $sample_{i-l-2}(W_l^j)$ and let e_p and e_n be the next and previous elements in $sample_{i-l-2}(W_l^j)$. Obviously e_n and e_p are in $sample_{i-l-1}(W_l^j)$ and $\overline{pred}(e,i)$ is between $\overline{pred}(e_p,i)$ and $\overline{pred}(e_n,i)$. Therefore new correct values of $\overline{pred}(e,i)$ for all $e \in W_l^j$ can be found in a constant time with $|sample_{i-l-2}(W_l^j)|$ processors. Since $\sum_{l \leq m, j \leq i+\log n} |sample_{i-l-2}(W_l^j)| = O(n)$, $\overline{pred}(e,i)$ can be recomputed in time $O(1)$ with n processors. New values of $\overline{pred'}(e,i)$ can be computed in the same way.

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

Since all other operations can also be done in a constant time we can perform $\log n$ iterations of **Package-Merge** in a logarithmic time. Combining this fact with Statement 1 we get

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

4 Almost-optimal length-limited codes

In this section we assume that element weights p_i are normalized, i.e. $\sum p_i = 1$. We say that a length-limited code $\mathcal{L}$ is almost-optimal with an error ϵ if $Length(\mathcal{L}, \overline{p}) \leq Length(\mathcal{L}', \bar{p}) + \epsilon$ for all length-limited codes $\mathcal{L}'$. An almost-optimal length-limited code with an error $\frac{1}{n^k}$ can be sequentially constructed in time $O(n \log n)$.

To achieve this goal we construct an optimal code for the set of items $\overline{p^{new}} = p_1^{new}, p_2^{new}, \ldots, p_n^{new}$, where $p_i^{new} = \lceil p_i n^k \rceil n^{-k}$. Let $\mathcal{L}^*$ denote an optimal code for weights $p_1, \ldots, p_n$. Since $p_i^{new} < p_i + n^{-k}$,

$$\sum p_i^{new} l_i < \sum p_i l_i + \sum n^{-k} l_i < \sum p_i l_i + n^2 n^{-k}$$

because all l_i are smaller than n. Hence $Length(\mathcal{L}^*, \overline{p^{new}}) < Length(\mathcal{L}, \overline{p}) + n^{-k+2}$. Let $\mathcal{L}_A$ denote the (optimal) Huffman code for weights p_i^{new}. Then

$$\begin{aligned} &Length(\mathcal{L}_A, \overline{p}) < Length(\mathcal{L}_A, \overline{p^{new}}) \leq Length(\mathcal{L}^*, \overline{p^{new}}) \quad \text{and} \\ &Length(\mathcal{L}^*, \overline{p^{new}}) < Length(\mathcal{L}^*, \overline{p}) + n^{-k+2} \end{aligned}$$

Therefore we can construct an optimal code for weights p^{new}, than replace p_i^{new} with p_i and the resulting code will have an error of at most n^{-k+2}.

The construction of a length-limited code with maximum codeword length L can be reduced to 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]). If element weights in the code construction problem are polynomially limited then edge weights in the corresponding graph will be also polynomially limited. Hence we can construct an almost optimal code in $O(n \log n)$ time.

We can also construct an almost-optimal length-limited code in parallel in a logarithmic time with $n \log n$ operations. Supposed we want to construct a code with the maximum codeword length L and an error $1/n^k$. If $L < k \log n$ we can construct an almost-optimal code by applying **Package-Merge** to the set of weights $\overline{p^{new}}$ defined above. If $L > k \log n$, we can construct an optimal (not length-limited) code for weights $\overline{p^{new}}$. It was shown in [BKN02] that the maximal codeword length in such a code is at most $k \log n < L$, therefore this code is also an optimal length-limited code. Since an optimal code can be constructed in time $O(k \log n)$ with n processors (see [BKN02]), we can construct an almost-optimal length-limited code in $O(k \log n)$ time with n processors.

We can also conclude from the last two paragraphs that an almost-optimal length-limited code with error $1/n^k$, such that $k < L/\log n$, can be sequentially constructed in linear time or in parallel time $O(k \log n)$ with n processors.

5 Conclusion

We have described a parallel algorithm for the construction of length-limited Huffman codes. This algorithm yields an optimal parallelization of the Package-Merge algorithm of Larmore and Hirschberg[LH90] , the problem that was open for some time now.

We also describe an algorithm for the construction of almost-optimal length-limited codes that works in $O(n \log n)$ time. We show that this algorithm can be implemented in time $O(\log n)$ with $n \log n$ operations. This is an optimal speed-up of the best known algorithm for the construction of almost-optimal length-limited codes.

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 Disrete & Computational Geomertry **12** (1994), pp. 263–280.

[BKN02] Berman, P., Karpinski, M., Nekritch, Y., *Approximating Huffman Codes in Parallel*, to appear in Proceedings of the 29th ICALP (2002).

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

[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., W.Rytter, *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.

[S95] Schieber, B., *Computing a minimu-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.

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'