View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

ELSEVIER

Available online at www.sciencedirect.com

Journal of Discrete Algorithms 5 (2007) 479–490

JOURNAL OF DISCRETE ALGORITHMS

www.elsevier.com/locate/jda

Approximating Huffman codes in parallel ☆

P. Berman [a,1], M. Karpinski [b,2], Y. Nekrich [b,*,3]

[a] *Department of Computer Science and Engineering, The Pennsylvania State University, USA*
[b] *Department of Computer Science, University of Bonn, Germany*

Received 31 October 2005; received in revised form 18 May 2006; accepted 19 October 2006

Available online 18 January 2007

Abstract

In this paper we present new results on the approximate parallel construction of Huffman codes. Our algorithm achieves linear work and logarithmic time, provided that the initial set of elements is sorted. This is the first parallel algorithm for that problem with the optimal time and work. Combining our approach with the best known parallel sorting algorithms we can construct an almost optimal Huffman tree with optimal time and work. This also leads to the first parallel algorithm that constructs exact Huffman codes with maximum codeword length H in time $O(H)$ with $n/\log n$ processors, if the elements are sorted.
© 2006 Published by Elsevier B.V.

Keywords: Approximation algorithms; Huffman codes; Parallel algorithms

1. Introduction

A Huffman code for an alphabet $a_1, a_2, \dots, a_n$ with weights $p_1, p_2, \dots, p_n$ is a prefix code that minimizes the average codeword length, defined as $\sum_{i=1}^{n} p_i l_i$. The problem of construction of Huffman codes is equivalent to the construction of Huffman trees (cf., e.g., [6,9]).

A problem of constructing a binary Huffman tree for a sequence $\bar{p} = p_1, \dots, p_n$ consists in constructing a binary tree T with leaves, corresponding to the elements of the sequence, so that the *weighted path length* of T is *minimal*. The weighted path length of T, $wpl(T)$ is defined as follows:

$$wpl(T, \bar{p}) = \sum_{i=1}^{n} p_i l_i$$

where l_i is the depth of the leaf corresponding to the element with weight p_i.

☆ A preliminary version of this paper appeared in the Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP 2002), LNCS, vol. 2380, pp. 845–855.

* Corresponding author.
E-mail addresses: berman@cse.psu.edu (P. Berman), marek@cs.uni-bonn.de (M. Karpinski), yasha@cs.uni-bonn.de (Y. Nekrich).

[1] Research done in part while visiting Department of Computer Science, University of Bonn. Work partially supported by NSF grant CCR-9700053 and DFG grant Bo 56/157-1.

[2] Supported in part by DFG grants, DIMACS, PROCOPE Project, and IST grant 14036 (RAND-APX).

[3] Work partially supported by IST grant 14036 (RAND-APX).

1570-8667/$ – see front matter © 2006 Published by Elsevier B.V.
doi:10.1016/j.jda.2006.10.007

The classical sequential algorithm, described by Huffman [6] can be implemented in $O(n \log n)$ time. Van Leeuwen has shown that if elements are sorted according to their weight, a Huffman code can be constructed in $O(n)$ time (see [9]). However, no optimal parallel algorithm for the problem of the construction of a Huffman code is known. Teng [14] has shown that the construction of a Huffman code is in class *NC*. His algorithm uses the parallel dynamic programming method of Miller et al. [11] and works in $O(\log^2 n)$ time on n^6 processors. Attalah et al. have proposed an n^2-processor algorithm, working in $O(\log^2 n)$ time. This algorithm is based on the multiplication of concave matrices. The fastest n-processor algorithm is due to Larmore and Przytycka [8]. Their algorithm based on a reduction of Huffman tree construction problem to the *concave least weight subsequence* problem runs in $O(\sqrt{n} \log n)$ time. Milidiu, Laber, and Pessoa [10] describe an algorithm that works in $O(H \log\log(n/H))$ time with n processors, where H is the height of a Huffman tree.

Kirkpatrick and Przytycka [7] introduce an approximate 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 $wpl(T') \leqslant wpl(T) + n^{-\alpha}$ for a fixed error parameter $\alpha \in \mathbb{N}$ (assuming $\sum p_i = 1$). We call $n^{-\alpha}$ an error factor. If the file size is polynomial in the size of the alphabet, compression with an almost optimal code instead of the optimal code leads to a compression loss limited by a constant. Kirkpatrick and Przytycka [7] propose several algorithms for that problem. In particular, they present an algorithm that works in $O(\alpha \log n \log^* n)$ time with n processors on a CREW PRAM, and a $O(\alpha^2 \log n)$ time algorithm that works with n^2 processors on a CREW PRAM.

The problems considered in this paper were also partially motivated by a work of one of the authors on decoding the Huffman codes [12,13].

In this paper we improve the before mentioned results by presenting an algorithm that works in $O(\alpha \log n)$ time with n processors. As we will see in the next section, the crucial step in computing a nearly optimal tree is merging two sorted arrays; this step is repeated $O(\log n^\alpha)$ times. We have developed a method for performing such a merging in average constant time.

We also further improve this result and design an algorithm that constructs almost-optimal codes in time $O(\log n)$ with $n/\log n$ processors, provided that elements are sorted. This results in an optimal speed-up of the algorithm of van Leeuwen [9]. Our algorithm works deterministically on a CREW PRAM and is the first parallel algorithm for that problem with the optimal time and work. Combining that algorithm with parallel integer sorting algorithms, we obtain several further results for the case of unsorted weights in Section 5. For instance, there is a deterministic algorithm that works on a CRCW PRAM in $O(\alpha \log n)$ time and with $n \log\log n/\log n$ processors.

The above described approach also leads to an algorithm for constructing exact Huffman trees that works in $O(H)$ time with n processors, where H is the height of a Huffman tree. This is an improvement of the result of Milidiu, Laber, and Pessoa [10]. This is also an improvement of the algorithm of Larmore and Przytycka [8] for the case when $H = o(\sqrt{n} \log n)$. We observe that in many situations the height of the Huffman tree is $O(\log n)$.

2. The basic construction scheme

Our algorithm uses the following *tree* data structure. A single element is a tree; if t_1 and t_2 are two trees, then $t = meld(t_1, t_2)$ is also a tree with weight $weight(t) = weight(t_1) + weight(t_2)$ so that t_1 and t_2 are children of t. Initial elements will be called leaves.

In a classical Huffman algorithm the set of trees is initialized with the set of weights. Then one melds consecutively two smallest elements in the set of trees until only one tree is left. The tree constructed by the Huffman algorithm can be proven to be optimal.

Kirkpatrick and Przytycka [7] presented a scheme for the parallelization of the Huffman algorithm called **Oblivious-Huffman**. In [7] the set of element weights $p_1, p_2, \ldots, p_n$ is partitioned into classes $W_1, \ldots, W_m$, such that elements of class W_i satisfy the condition $1/2^i \leqslant p < 1/2^{i-1}$. Here and further we assume that element weights are normalized. In the following sections elements of W_i are always sorted according to their weight; $W_i[c]$ refers to the cth smallest element in W_i. For any tree $t \in W_i$, $pos[t]$ denotes its position in W_i, so that $W_i[pos[t]] = t$. Length of a class A is denoted by $l(A)$; $last(A)$ denotes the last element in A. For ease of description, we sometimes do not distinguish between an element (tree) and its weight.

Since in the Huffman algorithm lightest elements are processed first and sum of any two elements in a class W_i is less than sum of any two elements in a class W_j, $j < i$, elements of the same class can be melded in parallel before

```
1   for i := m downto 1 do
2     if (light[i] ≠ NULL)
3       t := meld(light[i], W_i[1])
4       remove W_i[1] from W_i
5       if (weight(t) > 1/2^{i-1})
6         W_{i-1} := merge(W_{i-1}, {t})
7       else
8         W_i := merge(W_i, {t})
9     forall j ⩽ ⌊l(W_i)/2⌋ pardo
10      W̃_i[j] := meld(W_i[2j-1], W_i[2j])
11    W_{i-1} := merge(W_{i-1}, W̃_i)
12    if (l(W_i) is odd)
13      light[i-1] := last(W_i)
```

Fig. 1. Huffman tree construction scheme.

the elements of classes with smaller indices are processed. We refer the reader to [7] for a more detailed description of their algorithm.

In this paper a different schema for the parallelization of the greedy Huffman algorithm, further called **Parallel-Huffman** is used. The pseudocode description of this schema is shown on Fig. 1. The result of procedure $merge(A, B)$ applied to sorted arrays A and B is a sorted array C which consists of all elements of A and B. The algorithm consists of m iterations, where m is the index of the class to which the smallest element belongs. During the ith iteration all trees in class W_i, except of may be the last one, are processed. If W_i contains an odd number of elements, the last element is saved in variable $light[i-1]$ and will be processed during the next iteration. At the beginning of each iteration we check whether an element from class W_{i+1} that was not processed during the previous iteration is stored in $light[i]$. If this is the case, we meld $light[i]$ with the first element of W_i to obtain a new tree t. Then tree t is inserted into either W_i or W_{i-1} according to its weight. After this, the consecutive pairs of elements in W_i are melded and the resulting new trees are stored in array $\tilde{W}_i$ (lines 9–10 of Fig. 1). If W_i contains an odd number of elements, the last element has no sibling; this last element is saved in $light[i-1]$ (lines 12–13). Our pseudocode description is a simplified one; we assume that for all classes W_i, $i = m, \ldots, 1$, $W_i \neq \emptyset$. For completeness a more detailed pseudocode description is provided in Section 6.

During each iteration elements of some class W_i and, may be, one element $light[i]$ are processed. All elements e in W_i satisfy the inequality $1/2^i \leqslant weight(e) < 1/2^{i-1}$. We deal with tree $light[i]$, such that $weight(light[i]) < 1/2^{i-1}$, in lines 2–8. When tree t is reinstalled properly into W_i or W_{i-1}, all remaining elements that should be processed during the ith iteration satisfy the condition $1/2^i \leqslant weight(e) < 1/2^{i-1}$. Hence, all elements of W_i must be melded before any elements of classes W_a, $a < i$, are processed. After the elements of W_{i-1} are melded, the weights of the new trees are between $1/2^i$ and $1/2^{i-1}$; hence, the new trees must be installed into W_{i-1}.

Because the total number of iterations of algorithm **Parallel-Huffman** equals to the maximal class index m and is linear in the worst case, this approach does not lead to any improvements, if we want to construct an exact Huffman tree.

Kirkpatrick and Przytycka [7] also describe an approximation algorithm based on **Oblivious-Huffman**. In this paper we convert **Parallel-Huffman** into an approximation algorithm in a different way. We replace each weight p_i with $p_i^{new} = \lceil p_i n^{\alpha} \rceil n^{-\alpha}$. That is, p_i^{new} is the smallest number that is bigger than p_i and that is a multiple of $n^{-\alpha}$. Since $p_i^{new} < p_i + n^{-\alpha}$,

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

because all l_i are smaller than n. Hence $wpl(T, \bar{p}) \leqslant wpl(T, \bar{p}^{new}) < wpl(T, \bar{p}) + n^{-\alpha+2}$ for any tree T. Let T_A denote the (optimal) Huffman tree for weights $p_1^{new}, p_2^{new}, \ldots, p_n^{new}$. Let T^* denote an optimal tree for weights $p_1, \ldots, p_n$. Then

$$wpl(T_A, \bar{p}) \leqslant wpl\big(T_A, \bar{p}^{new}\big) \leqslant wpl\big(T^*, \bar{p}^{new}\big) < wpl(T^*, \bar{p}) + n^{-\alpha+2}$$

Therefore we can construct an optimal tree for weights p^{new}, then replace p_i^{new} with p_i and the resulting tree will have an error of at most $n^{-\alpha+2}$.

If we apply algorithm **Parallel-Huffman** to the new set of weights, then the number of iterations of this algorithm will be $\lceil \alpha \log_2 n \rceil$, since new elements will be divided into at most $\lceil \alpha \log_2 n \rceil$ arrays. The use of approximate weights has further advantages that will be discussed in Section 5.

3. An O($\alpha \log n$) time algorithm

In this section we describe an $O(\alpha \log n)$ time n-processor algorithm that works on CREW PRAM.

Algorithm **Parallel-Huffman** performs $\alpha \log n$ iterations and in each iteration only the merge operations are difficult to implement in constant time. All other operations can be performed in constant time. We will use the following simple fact described in e.g., [15]:

Proposition 1. *If array A has a constant number of elements and array B has at most n elements, then arrays A and B can be merged in a constant time and with n processors.*

Proof. Let $C = merge(A, B)$. We assign a processor to every possible pair $A[i]$, $B[j]$, $i = 1, \ldots, c$, and $j = 1, \ldots, n$. If $A[i] < B[j] < A[i+1]$, then $B[j]$ will be the $(i+j)$th element in array C. Also if $B[j] < A[i] < B[j+1]$, then $A[i]$ will be the $(i+j)$th element in array C. □

Proposition 1 allows us to implement procedures $W_{i-1} := merge(W_{i-1}, \{t\})$ and $W_i := merge(W_i, \{t\})$ (lines 5–8 of Fig. 1) in constant time with $|W_i|$ and $|W_{i-1}|$ processors respectively.

Operation $merge(W_{i-1}, W_i)$ is the slowest one, because both W_i and W_{i-1} can have linear size and merging two arrays of size n requires $\log \log n$ operations in the general case (see [15]). In this paper we propose a method that allows us to perform every merge of **Parallel-Huffman** in constant time (on average). The key to our method is that at the time of merging all elements in both arrays know their predecessors in the other array and can thus compute their positions in a resulting array in constant time. A merging operation itself is performed without comparisons. Comparisons will be used for the initial computation of predecessors and to update predecessors after each merge and meld procedure.

We say that element e is of rank k, if $e \in W_k$. A relative weight $r(p)$ of an element p of rank k is $r(p) = p \cdot 2^k$. To make description more convenient we say that in every array W_k $W_k[0] = 0$ and $W_k[l(W_k) + 1] = +\infty$. We assume that whenever $e \neq e'$, $r(e) \neq r(e')$; this can be "enforced" by introducing a tie-breaking rule, which will be described later. Besides that, if leaf e and tree t are of rank k, and t is the result of melding two elements t_1 and t_2 of rank $k+1$, such that $r(t_1) > r(e)$ and $r(t_2) > r(e)$ ($r(t_1) < r(e)$ and $r(t_2) < r(e)$), then the weight of t is bigger (smaller) than the weight of e. This fact can be generalized; in the following proposition, the *full tree* is a binary tree in which every internal node has exactly two children.

Proposition 2. *Let t be a full tree of rank k and e be an element of the same rank. If all leaves of t have relative weight smaller* (*bigger*) *than $r(e)$, then t has smaller* (*bigger*) *weight than e.*

This statement can be proven by induction on the depth of tree t. Obviously, if $r(t_1) < r(t_2)$, and t_1 and t_2 are of the same rank, then t_1 has smaller weight then t_2.

We compute for every leaf e and every class i the value of $pred(e, i) = W_i[j]$, s.t. $r(W_i[j]) < r(e) < r(W_i[j+1])$. In other words, $pred(e, i)$ is the biggest element in class i, whose relative weight is smaller than or equal to $r(e)$. At the beginning of our algorithm the values of $pred(e, j)$ for all leaves e and all classes j are found. To find those values, we copy all elements into an array R and sort R according to the relative weight of its elements. This can be done in $O(\log n)$ time with n processors. Then for every class j an array $C^j[i]$ is constructed, such that $C^j[i] = 1$ if $R[i] \in W_j$ and $C^j[i] = 0$ otherwise. We compute prefix sums for all arrays C^j and store them in arrays P^j, so that $P^j[i] = \sum_{m=1}^{i} C^j[m]$. Arrays C^j can be constructed from R in $O(\log n)$ time with n processors. A prefix sum for an array with n elements can be constructed in $O(\log n)$ time with n processors. Since the total number of arrays C^j is $O(\log n)$, all arrays P^j can be computed in $O(\log n)$ time with n processors. Let t be the position of an arbitrary element e in R. Then for all classes j, $pred(e, j) = P^j[t]$. Hence, $pred(e, j)$ can be found in $O(\log n)$ time with n processors.

If we are able to compute $pred(e, j)$ for all $e \in \tilde{W}_i$ and $j = i-1, i-2, \ldots, 1$ after melding the elements of W_i, we will be able to merge arrays $\tilde{W}_i$ and W_{i-1} (line 11 of Fig. 1) in constant time.

Our implementation of one iteration of the algorithm **Parallel-Huffman** consists of the following steps:

A.1. If there is an element $light[i]$, we meld it with the first element of W_i and insert the new element t at the appropriate position of W_i or W_{i-1}, as described in lines 2–8 of the pseudocode on Fig. 1. We compute the values of $pred(t, a)$ for $a = i-1, \ldots, 1$ and recompute the values of $pred(e, i)$ for all $e \in W_1 \cup \cdots \cup W_{i-1}$ if necessary.

A.2. We meld consecutive pairs of elements in W_i and store the resulting elements in $\tilde{W}_i$ (lines 9–10 of Fig. 1).

A.3. We compute the values of $pred(e, a)$ for all $e \in \tilde{W}_i$ and $a = i-1, \ldots, 1$, and $pred'(e, i)$ for all $e \in W_1 \cup W_2 \cup \cdots \cup W_{i-1}$, where $pred'(e, i) = s \in \tilde{W}_i$, such that $r(s) < r(e) < r(next(s))$. We denote by $next(e)$ for $e \in W_i$ the element that follows e in W_i ($next(e) = W_i[pos[e]+1]$). If e is the last element in W_i, $next(e) = +\infty$.

A.4. Using the values of $pred$ and $pred'$ computed during the previous step, we merge $\tilde{W}_i$ and W_{i-1}, store the result in W_{i-1}(line 11 on Fig. 1), and update the values of $pred(e, i-1)$ for $e \in W_{i-2} \cup \cdots \cup W_1$.

A.5. If the number of elements in W_i is odd, we save the last element of W_i in $light[i-1]$ (lines 12–13 of Fig. 1).

`Step A.1` We can insert $light[i]$ into W_i in O(1) time using Proposition 1. We can also compute values of $pred(t, a)$ for $a = i-1, \ldots, 1$, and correct values of $pred(e, i)$ for all $e \in W_1 \cup \cdots \cup W_{i-1}$ in O(1) time.

`Steps A.2 and A.3` We use the algorithm shown on Fig. 2 to compute values of $pred(e, t)$ for all $e \in \tilde{W}_i$ and $t = i-1, \ldots, 1$ after melding the elements from W_i. This algorithm also computes values $pred'(e, i)$ for all $e \in W_{i-1}, \ldots, W_1$. Here and further variables which are local for each processor are denoted by **local var** in a pseudocode description.

In lines 1–2 of Fig. 2, we meld the consecutive elements of W_i and store them in an array $\tilde{W}_i$. If left and right children of some element $\tilde{W}_i[c]$ i.e., $W_i[2c-1]$ and $W_i[2c]$ have equal $pred$ values for some class a, $pred(W_i[2c-1], a) = pred(W_i[2c], a)$, then $pred(\tilde{W}_i[c], a) = pred(W_i[2c-1], a) = pred(W_i[2c], a)$. In lines 4–5 of Fig. 2 we check for a more general condition. Namely, we check whether $pred(\tilde{W}_i[c], a) = pred(W_i[2c-1], a)$. The case when $pred(W_i[2c-1], a) \neq pred(W_i[2c], a)$ and $pred(W_i[2c-1], a) \neq pred(\tilde{W}_i[c], a)$ will be considered later.

In lines 6–11 values of $pred'$ for elements of all non-empty classes W_a, $a < i$, are computed. Suppose for some element $e \in W_a$ $pred(e, i) = s \in W_i$, and s has become the right child of some $s' \in \tilde{W}_i$. Then $s' = pred'(e, i)$ after melding, because the relative weights of both the left and the right children of s' are smaller than the relative weight of e. If s is the left child of s', then the relative weight of e may be smaller than the relative weight of s'. But the element s'' preceding s' in $\tilde{W}_i$ has smaller relative weight than e, since both of its children are of smaller relative weight. We can decide between s' and s'' and find the correct value of $pred'(e, i)$ in constant time.

When the values of $pred'(e, i)$ for $e \in W_{i-1}, \ldots, W_1$ are known, the computation of $pred(e, a)$ for $e \in \tilde{W}_i$, $a = i-1, \ldots, 1$, can be completed. Consider an element $t \in \tilde{W}_i$, $t = meld(t_1, t_2)$, $t_1, t_2 \in W_i$. Let t' and t'' be the elements preceding and following t in $\tilde{W}_i$, and let t'_1, t'_2 and t''_1, t''_2 be the children of t' and t'' respectively. Suppose $pred(t_2, a) = p_2$ and $pred(t_1, a) = p_1$ for some $a < i$, and $p_2 \neq p_1$ (see Fig. 3). Obviously, $pred(t, a)$ is between p_1 and p_2 in W_a. The case $pred(t_1, a) = pred(t, a)$ was considered earlier. For all $e \in W_a$, s.t. $pos[p_1] < pos[e] \leqslant pos[p_2]$, $pred'(e, i)$ is either t or t'. This follows from the fact that both t'_1 and t'_2 have smaller relative weight than e, and both t''_1 and t''_2 have bigger relative weight than e. Furthermore, if $pred(t, a) = p$ and p' is the element that follows p in W_a, then $r(p') > r(t)$. Therefore, we must check for every element $p \in W_{i-1} \cup \cdots \cup W_1$, whether the element t following $t' = pred'(p, i)$ has smaller relative weight than the element p' following p. And if this is the case, we set $pred(t, a) = p$ (lines 13–16 of Fig. 2).

`Step A.4` When the elements of W_i are melded, predecessor values $pred(e, i)$ are recomputed, and $pred'(e, i)$ for $e \in W_{i-1} \cup \cdots \cup W_1$ are found, classes $\tilde{W}_i$ and W_{i-1} can be merged easily (see Fig. 4). Indeed, $pos[pred'(W_{i-1}[j], i)]$ equals to the number of elements in $\tilde{W}_i$ that are smaller than or equal to $W_{i-1}[j]$. Analogously, $pos[pred(\tilde{W}_i[j], i-1)]$ equals to the number of elements in W_{i-1} that are smaller than or equal to $\tilde{W}_i[j]$. Therefore indices of all elements in the merged array can be computed in constant time. When elements of $\tilde{W}_i$ and W_{i-1} are merged, $pred(e, i-1)$ for all e in $W_{i-2}, W_{i-3}, \ldots, W_1$ must be updated. The new value of $pred(e, i-1)$ is the maximum of $pred(e, i-1)$ and $pred'(e, i)$, for all e.

If the number of elements in W_i is odd then the last element of W_i must be saved in $light[i-1]$ to be inserted into W_i during the next iteration (step A.5). This can obviously be done in O(1) time with 1 processor.

```
1  forall c ⩽ l(W_i)/2 pardo
2    W̃_i[c] = meld(W_i[2c − 1], W_i[2c])
3    forall a < i pardo
4      if ((r(pred(W_i[2c − 1], a)) < r(W̃_i[c])) AND
            (r(next(pred(W_i[2c − 1], a))) > r(W̃_i[c])))
5        pred(W̃_i[c], a) = pred(W_i[2c − 1], a)

6  forall s ∈ W_a, s.t. a < i, W_a ≠ ∅
7    local var temp := ⌈pos[pred(s, i)]/2⌉
8    if r(W̃_i[temp]) > r(s)
9      pred′(s, i) := W̃_i[temp − 1]
10   else
11     pred′(s, i) := W̃_i[temp]

12 forall s ∈ W_a, s.t. a < i, W_a ≠ ∅
13   local var v
14   if r(next(s)) > r(next(pred′(s, i)))
15     v := next(pred′(s, i))
16     pred(v, a) := s
```

Fig. 2. Melding consecutive pairs of elements in W_i.

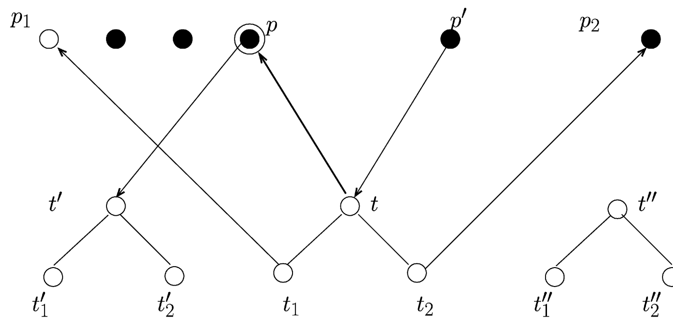

Fig. 3. Computing $pred(t, a)$, if $t_1 \neq t_2$.

```
1  forall j ⩽ l(W_{i−1}), k ⩽ l(W̃_i) pardo
2    local var k′, j′
3    k′ := pos[pred′(W_{i−1}[j], i)]
4    j′ := pos[pred(W̃_i[k], i − 1)]
5    pos[W_{i−1}[j]] := pos[W_{i−1}[j]] + k′
6    pos[W̃_i[k]] := pos[W̃_i[k]] + j′
```

Fig. 4. Merging W_{i-1} and $\tilde{W}_i$.

In the algorithm described in this section we assume that relative weights of all compared elements are different. We can guarantee that all elements have different weights by the following tie-breaking rule. We replace initial weights p_k by weights $p'_k = p_k \cdot n + k$. Furthermore, if $r(e_1) = r(e_2)$ we assume that $r(e_1) \succ r(e_2)$ if e_2 is a leaf and e_1 has children. If both e_1 and e_2 are leaves of the same relative weight, then e_1 and e_2 belong to different classes. In this case we assume that $r(e_1) \prec r(e_2)$ if $e_1 \in W_{i_1}$, $e_2 \in W_{i_2}$ and $i_1 > i_2$ (i.e., e_2 has bigger weight than e_1). For every non-leaf node e, *the dominating leaf* of e is the leaf descendant of e with the largest relative weight. When two elements are melded, the dominating leaf of the result can be found in constant time. If $r(e_1) = r(e_2)$ and both e_1 and e_2 are not leaves, then $r(e_1) \succ r(e_2)$, iff the dominating leaf of e_2 has smaller relative weight than the dominating leaf of e_1.

Steps A.1, A.4, and A.5 can be implemented to work in constant time with n processors. Computing $pred'(e, i)$ for all e in $W_{i-1}, W_{i-2}, \ldots, W_1$ takes O(1) time with n processors. The total number of operations that we must perform to compute $pred(e, i)$ for $e \in \tilde{W}_m \cup \cdots \cup \tilde{W}_1$ is $O(n \log n)$, because the total number of elements in all classes $\tilde{W}_i$ does not exceed the number of internal nodes in a Huffman tree. We can distribute the processors in such way that computing $pred(e, a)$, $a = i - 1, \ldots, 1$, for class $\tilde{W}_i$ takes $O(\lceil (|\tilde{W}_i|/n) \log n \rceil)$ time with n processors. Therefore

computing $pred(e,a)$ for all classes $\tilde{W}_i$ and all a takes $O(\log n)$ time with n processors. Thus steps A.2 and A.3 require $O(\log n)$ time for *all* iterations. We obtain the following

Theorem 1. *An almost optimal tree with error factor $1/n^\alpha$ can be constructed in $O(\alpha \log n)$ time with n processors on a CREW PRAM.*

4. An $O(\alpha n)$ work algorithm

In this section we describe a further improvement of the merging scheme presented in the previous section. The modified algorithm works on a CREW PRAM in $O(\log n)$ time and with $n/\log n$ processors, provided that initial elements are sorted.

The main idea of our modified algorithm is that we do not use all values of $pred(e,i)$ at each iteration. In fact, if we know values of $pred(e,i-1)$ for all $e \in \tilde{W}_i$ and values of $pred'(e,i)$ for all $e \in W_{i-1}$, then merging can be performed in constant time. Therefore, we will use function $\overline{pred}$ instead of $pred$ such that the necessary information is available at each iteration, but the total number of values in $\overline{pred}$ is limited by $O(n)$. Again, we are able to recompute values of $\overline{pred}$ in constant average time (averaged by iteration).

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

Initial values of $\overline{pred}$ can be computed as follows. Consider an arbitrary pair of classes W_i and W_j. We store relative weights of $sample_{|i-j|-1}(W_j)$ and $sample_{|i-j|-1}(W_i)$ in arrays R_{ji} and R_{ij} respectively. We can merge two sorted arrays R_{ji} and R_{ij} in $O(\log n)$ time with $(|R_{ji}|+|R_{ij}|)/\log n$ processors. Then, we can find the values of $\overline{pred}(e,i)$ for $e \in sample_{|i-j|-1}(W_j)$ and $\overline{pred}(e,j)$ for $e \in sample_{|i-j|-1}(W_i)$ also in $O(\log n)$ time with $(|R_{ji}|+|R_{ij}|)/\log n$ processors. Since the total number of elements in all R_{ij} is $O(n)$ all initial values can be computed in $O(\log n)$ time with $n/\log n$ elements.

Each iteration of **Parallel-Huffman** can now be implemented with the following steps:

B.1. If there is an element $light[i]$, we meld it with the first element of W_i and insert the new element t at the appropriate position of W_i or W_{i-1}; this corresponds to lines 2–8 of the pseudocode on Fig. 1. We compute $\overline{pred}(t,a)$ for $a=i-1,\dots,1$ and recompute the values of $\overline{pred}(e,i)$ for all $e \in W_1 \cup \cdots \cup W_{i-1}$ if necessary.

B.2. We meld consecutive pairs of elements in W_i and store the resulting elements in $\tilde{W}_i$ (lines 9–10 of Fig. 1).

B.3. We compute the values of $\overline{pred}(e,a)$ for all $e \in \tilde{W}_i$ and $a=i-1,\dots,1$, where $\overline{pred}(e,a)$ for each $e \in \tilde{W}_i$ is the biggest element in $s \in sample_{|i-a|-1}(W_i)$, such that $r(s) \leqslant r(e)$. We also compute $\overline{pred}'(e,i)$ for all $e \in W_1 \cup W_2 \cup \cdots \cup W_{i-1}$, where $\overline{pred}'(e,i) = s \in sample_{|i-a|-1}(\tilde{W}_i)$, such that $r(s) < r(e) < r(next(s))$ for $e \in W_a$. If necessary, we "refine" the values of $\overline{pred}(e,a)$ and $\overline{pred}'(e,i)$ as described below.

B.4. Using the "refined" values of $\overline{pred}$ and $\overline{pred}'$ computed during the previous step, we merge $\tilde{W}_i$ and W_{i-1}, store the result in W_{i-1} (line 11 on Fig. 1), and update the values of $\overline{pred}(e,i-1)$ for $e \in W_{i-2} \cup \cdots \cup W_1$.

B.5. If the number of elements in W_i is odd, we save the last element of W_i in $light[i-1]$ (lines 12–13 of Fig. 1).

Steps B.1–B.5 are almost identical to steps A.1–A.5 from Section 3; the only difference is that the values of $\overline{pred}$ and $\overline{pred}'$ are computed and maintained. Now we turn to the description of steps 2 and 3, i.e. we show how $\overline{pred}$ can be recomputed after elements in a class W_i are melded. We assign one processor to every pair (e,i) for which $\overline{pred}(e,i)$ must be computed.

Observe that if $|i-a|=1$, $sample_{|i-a|-1}(W_i) = sample_0(W_i) = W_i$. Hence for $|i-a|=1$ and $e \in W_i$, $\overline{pred}(e,a) = pred(e,a)$, and the values of $pred(e,a)$ are known for all $e \in W_i$. Therefore, if $a=i-1$ $\overline{pred}(e,a)$ for $e \in \tilde{W}_i$ and $\overline{pred}'(e,i)$ for $e \in W_{i-1}$ can be recomputed after melding in the same way as in Section 3. Below we show how the values of $\overline{pred}(e,a)$ for $e \in W_i$ and $a=i-2,i-3,\dots,$ and $\overline{pred}'(e,i)$ for $e \in W_{i-2} \cup W_{i-3} \cup \cdots \cup W_1$ are maintained. In this case step B.3 consists of the following substeps:

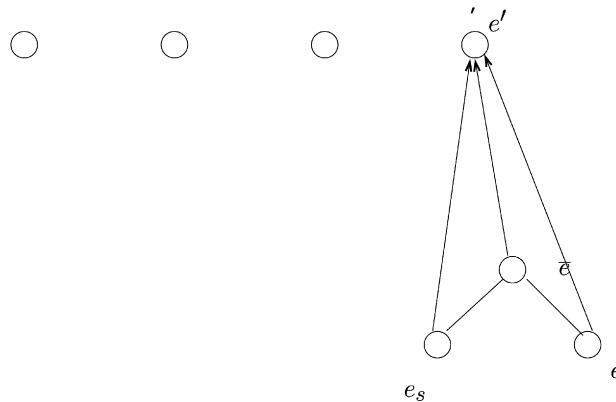

Fig. 5. Recomputing $pred(\bar{e}, a)$, if $pred(e_s, a) = pred(e, a)$.

B.3.1. We compute for each $e \in sample_{|a-i|-2}(\tilde{W}_i)$ and $a = i-2, \ldots, 1$ the biggest p in $sample_{|i-a|-1}(W_a)$, such that $r(p) < r(e)$, and for each $e' \in sample_{|i-a|-1}(W_a)$, $a = i-2, \ldots, 1$, the biggest $p' \in sample_{|i-a|-2}(\tilde{W}_i)$ such that $r(p') < r(e')$.

B.3.2. For each $e \in sample_{|a-i|-2}(\tilde{W}_i)$ and $a = i-2, \ldots, 1$, we compute the biggest p in $sample_{|i-a|-2}(W_a)$, such that $r(p) < r(e)$. (We refine the values of $\overline{pred}(e, a)$.)

B.3.3. For each e' such that $e' \in sample_{|i-a|-2}(W_a)$ and $e' \notin sample_{|i-a|-1}(W_a)$, $a = i-2, \ldots, 1$, we compute $\overline{pred}'(e', i)$. (We compute the missing values of $\overline{pred}'$).

`Step B.3.1` We denote by *sibling*(e) an element with which e will be melded in **Parallel-Huffman**. Consider an arbitrary pair (e, a), $e \in sample_{|i-a|-1}(W_i)$. First, the value $\overline{pred}(e, a)$ is known, but the value of $\overline{pred}(e_s, a)$, where $e_s = sibling(e)$ may be unknown. Let e_p and e_n be the previous and the next elements of e in $sample_{|i-a|-1}(W_i)$, let $\bar{e} = meld(e, e_s)$ (see Fig. 5). The set $sample_{|i-a|-1}(W_i)$, $a < i-1$, consists of elements e such that $pos[e] = 0 \pmod 2$, therefore when consecutive elements of W_i are melded all $e \in sample_{|i-a|-1}(W_i)$ will always be the right children of elements in $\tilde{W}_i$. Since e_s precedes e in W_i then the correct value of $\overline{pred}(e_s, a)$ is between $\overline{pred}(e_p, a)$ and $\overline{pred}(e, a)$. We check whether $\overline{pred}(e, a) = \overline{pred}(\bar{e}, a)$, and if this is the case set $\overline{pred}(\bar{e}, a)$ to $\overline{pred}(e, a)$. We will show below how the correct value of $\overline{pred}(\bar{e}, a)$ can be computed if $\overline{pred}(\bar{e}, a) \neq \overline{pred}(e, a)$.

Next, we compute the values of $\overline{pred}'(e', i)$ for $e' \in sample_{i-2}(W_1) \cup sample_{i-3}(W_2) \cup \cdots \cup sample_1(W_{i-2})$. Suppose that $e' \in sample_{|a-i|-1}(W_a)$. Let $e = \overline{pred}(e', i)$, $e_s = sibling(e)$ and $\bar{e} = meld(e, e_s)$. The computation of $\overline{pred}'(e', i)$ is similar to the computation of $pred'$ in Section 3. But now $pred(e', i)$ can only be the right child of $\bar{e}$; hence, the relative weight of e' is bigger than the relative weight of e, and $pred'(e, i) = \bar{e}$. When the values of $pred'(e, i)$ are known, the computation of $pred(e, i)$ can be completed as follows. Observe that $\overline{pred}(\bar{e}, a) \neq \overline{pred}(e, a) \Rightarrow \overline{pred}(e_s, a) \neq \overline{pred}(e, a)$. Since the case $\overline{pred}(\bar{e}, a) = \overline{pred}(e, a)$ was considered above, only the case $\overline{pred}(\bar{e}, a) \neq \overline{pred}(e, a)$ (and hence $\overline{pred}(e_s, a) \neq \overline{pred}(e, a)$) has to be considered (see Fig. 6). Let $e' = pred(\bar{e}, a)$ and let f' be the element following e' in $sample_{i-a-1}(W_a)$. Since $f' \leqslant pred(e, a)$, $r(f') < r(\bar{e}_n)$, where $\bar{e}_n$ is the element following $\bar{e}$ in $\tilde{W}_i$. Therefore, $pred'(f', i) = \bar{e}$. Hence, $pred(\bar{e}, a)$ is such element e' that $r(e') < r(\bar{e})$ and $pred'(f', i) = \bar{e}$, where f' is the element that follows e' in $sample_{i-a-1}(W_a)$. A pseudocode description of the parallel meld operation is shown on Fig. 8.

`Steps B.3.2 and B.3.3` When elements from W_i are melded the new elements will belong to $\tilde{W}_i$. Since we have melded consecutive pairs of elements from W_i, distances between consecutive elements from $sample_{|i-a|-1}(W_i)$ have decreased by factor 2 and we now know $\overline{pred}(e, a)$ in $sample_{|i-a|-1}(W_a)$ for every $2^{|i-a|-2}$th element from W_i. Now we have to compute $\overline{pred}(e, a)$ in $sample_{|i-a|-2}(W_a)$. Suppose $\overline{pred}(e, a) = W_a[p \cdot 2^{|i-a|-1}]$ for some p. We can find the new "refined" value of $\overline{pred}(e, a)$ by comparing $r(e)$ with $r(W_a[p \cdot 2^{i-a-1} + 2^{i-a-2}])$. When the new correct values of $\overline{pred}(e, i)$ for $e \in sample_{|a-i|-1}(W_i)$ are known, we can compute $\overline{pred}'(e, i)$ for new elements e from $sample_{|i-a|-2}(W_a)$. Let e' be a new element in $sample_{|i-a|-2}(W_a)$ and let e'_p and e'_n be the next and the pre-

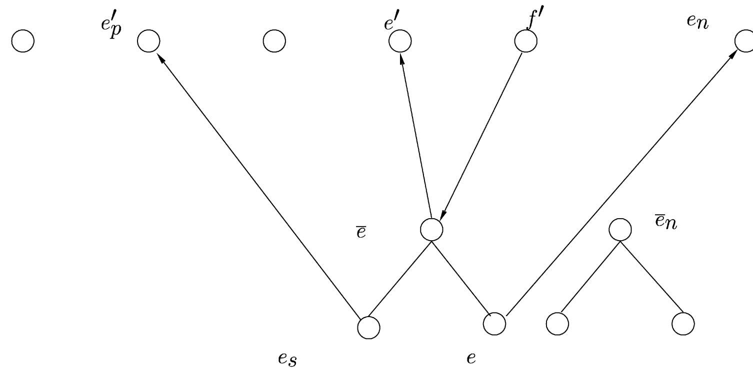

Fig. 6. Recomputing $pred(\bar{e}, a)$, if $pred(e_s, a) \neq pred(e, a)$.

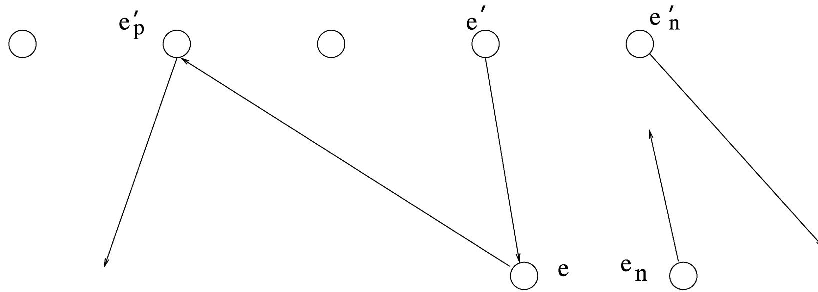

Fig. 7. Computing predecessors of W_a after elements of W_i are melded.

1 **forall** $c \leqslant l(W_i)/2$ **pardo**
2 $\quad \tilde{W}_i[c] := meld(W_i[2c-1], W_i[2c])$

3 **forall** $a < i-1, b \leqslant l(sample_{|i-a|-2}(W_i))$
4 $\quad$ **local var** c, s
5 $\quad c := b \cdot 2^{i-a-2}$
6 $\quad s := pred(W_i[2c], a)$
7 $\quad$ if $r(s) < r(\tilde{W}_i[c])$ AND
$\qquad\quad r(W_a[pos[s] + 2^{i-a-1}]) > r(\tilde{W}_i[c])$
8 $\qquad pred(\tilde{W}_i[c]) := s$
9

10 **forall** $a < i-1, b \leqslant l(sample_{|i-a|-1}(W_a))$ **pardo**
11 $\quad$ **local var** s
12 $\quad s := W_a[b \cdot 2^{|i-a|-1}]$
13 $\quad pred'(s, i) := \tilde{W}_i \lceil pos[pred(s, i)]/2 \rceil$

14 **forall** $a < i-1, b \leqslant l(sample_{|i-a|-1}(W_a))$ **pardo**
15 $\quad$ **local var** $d1, s, e$
16 $\quad d1 := 2^{|i-a|-1}$
17 $\quad s := W_a[b \cdot d1]$
18 $\quad e := pred'(s, i)$
19 $\quad$ **if** $r(e) > r(W_a[pos[s] - d1])$
20 $\qquad pred(e, a) := W_a[pos[s] - d1]$

Fig. 8. Melding elements of W_i for the improved algorithm for $a < i-1$.

vious elements in $sample_{|i-a|-2}(W_a)$ (Fig. 7). Obviously, e'_n and e'_p are also in $sample_{|i-a|-1}(W_a)$. If $\overline{pred'}(e'_p, i) = \overline{pred'}(e'_n, i)$ then $\overline{pred'}(e', i) = \overline{pred'}(e'_p, i) = \overline{pred'}(e'_n, i)$. Otherwise, let $e = \overline{pred'}(e', i)$ and let e_p and e_n be the elements preceding e and following e in $sample_{|i-a|-2}(\tilde{W}_i)$. If $e \neq \overline{pred'}(e'_p, i)$ (i.e., $\overline{pred'}(e', i) \neq \overline{pred'}(e'_p, i)$),

then $\overline{pred}(e,a) = e'_p$ and e_n has bigger relative weight than e'. Therefore the values of $\overline{pred'}(e',i)$ can be found as follows: First for all $a < i$ and all e', such that $e' \in sample_{|i-a|-2}(W_a)$ and $e' \notin sample_{|i-a|-1}(W_a)$ we check if $\overline{pred'}(e',i) = pred'(e'_p,i)$. Then, for all $e \in sample_{i-a-2}(W_i)$ and all $a < i$ we check if the element e' following $pred(e,a)$ in $sample_{i-a-2}(W_a)$ has bigger relative weight than e and smaller relative weight than the element following e in $sample_{i-a-2}(W_i)$. If this is the case, we set $pred'(e',i) = e$.

`Step B.4` Using the values of $\overline{pred}$ and $\overline{pred'}$, we can merge W_{i-1} and $\tilde{W}_i$ in constant time in the same way as described in Section 3. When elements of $\tilde{W}_i$ were added to W_{i-1}, distances between elements for which $pred(e,j)$, $j < i-1$, is known may exceed $2^{|j-i|-1}$. However, those distances never exceed $2^{|j-i|}$. We can find the new values using the refinement procedure similar to the procedure described above.

Since the total number of pairs (e,i) for which $\overline{pred}(e,i)$ or $\overline{pred'}(e,i)$ must be computed during all iterations of **Parallel-Huffman** is $O(n)$, we can perform $\alpha \log n$ iterations of **Parallel-Huffman** in $O(\alpha \log n)$ time. Therefore we obtain

Theorem 2. *An almost optimal tree with error factor $1/n^{\alpha}$ can be constructed in time $O(\alpha \log n)$ and with $n/\log n$ processors, if elements are sorted according to their weight.*

5. Almost-optimal codes for the unsorted set of weights

We can combine the algorithm that constructs a Huffman tree for a sorted set of weights with algorithms for the parallel bucket sort. Depending on the chosen computation model and assumptions about the size of the machine word we can get several slightly different results. We show that in this case optimal time-processor product can be achieved under reasonable conditions.

Recall that our algorithm works with weights p_i^{new}; p_i^{new} can be generated from p_i in $O(\log n)$ time with $n/\log n$ processors. Each weight p_i^{new} is of the form $m \cdot n^{-\alpha}$ for $m \in 0..n^{\alpha}$, i.e. p_i^{new} is a product of $n^{-\alpha}$ and an integer in the range $0..n^{\alpha}$. Hence, we can sort p_i^{new} by sorting polynomially bounded integers (i.e. integers whose values are bounded by a polynomial function of n). Observe that the algorithm of [7] works with weights that may be arbitrarily small.

Using a parallel bucket sort algorithm described in [4] we can sort polynomially bounded integers in $O(\log n \log\log n)$ time with $n/\log n$ processors on a priority CRCW PRAM. Using the algorithm described by Bhatt et al. [2] we can also sort polynomially bounded integers in the same time and processor bounds on arbitrary CRCW PRAM. Combining these results with our modified algorithm we get

Proposition 3. *An almost optimal tree with error $1/n^{\alpha}$ can be constructed in $O(\alpha \log n)$ time and with $n \log\log n/\log n$ processors on an arbitrary CRCW PRAM.*

Applying an algorithm of Hagerup [5] we get the following result:

Proposition 4. *An almost optimal tree with error $1/n^{\alpha}$ can be constructed for the set of n uniformly distributed random numbers with $n/\log n$ processors in time $O(\alpha \log n)$ and with probability $1/C^{-\sqrt{n}}$ for any constant C.*

Observe that in the case of Proposition 4 the weights of symbols are chosen uniformly at random; hence, the weights of different symbols are different with high probability.

By using the results of Andersson, Hagerup, Nilsson and Raman [1], n integers in the range $0..n^{\alpha}$ can be sorted in $O(\log n)$ time and with $n \log\log n/\log n$ processors on a unit-cost CRCW PRAM with machine word length $O(\log n)$. Finally, [1] shows that n integers can be probabilistically sorted in an expected time $O(\log n)$ and expected work $O(n)$ on a unit-cost EREW PRAM with word length $O(\log^{2+\varepsilon} n)$. In [3] it was shown that polynomially bounded integers can be deterministically sorted in $O(\log n)$ time with $n/\log n$ processors, if the word size is $O(\log^2 n)$.

Proposition 5. *An almost optimal tree with error $1/n^{\alpha}$ can be constructed in time $O(\alpha \log n)$ with $n/\log n$ processors on a EREW PRAM with word size $\log^2 n$.*

The last statement shows that an almost-optimal Huffman tree can be constructed in logarithmic time with $O(n)$ operations on a CREW PRAM with polylogarithmic word length.

6. Application to the case of exact Huffman trees

The algorithm described in the previous section can also be applied to the construction of exact Huffman trees. The difference is that in the case of exact Huffman trees weights of elements are unbounded and the number of classes W_i is $O(n)$ in the worst case. It is easy to modify the algorithm **Parallel-Huffman**, so that the number of iterations is proportional to the number of non-empty classes. The pseudocode description of the modified variant called **Detailed-Parallel-Huffman** is provided on Fig. 9; *nextclass*$[i]$ is the next non-empty class after W_i. Let b be the number of non-empty classes processed by **Detailed-Parallel-Huffman**. Below we show that $b = O(H)$ where H is the height of a Huffman tree.

Proposition 6. *Suppose there is node e of depth d_i in some Huffman tree T, such that $e \in W_i$. Then there is at most one node e' of depth d_i in T, such that $e' \in W_{i+2} \cup W_{i+3} \cup \cdots \cup W_b$.*

Proof. Suppose that there are two nodes $e_1 \in W_{i_1}$ and $e_2 \in W_{i_2}$ so that both e_1 and e_2 are of depth d_i, and $i_1 > i + 1$, $i_2 > i + 1$. We can swap the sibling of e_1 and e_2; this does not change the weighted path length of T. Now the weight of the parent node of e_1 and e_2 is smaller than $1/2^i$, but the weight of e is at least $1/2^i$. Since e is of depth d_i and the parent of e_1, e_2 is of depth $d_i - 1$, we can swap e and the parent of e_1 and e_2, and obtain the tree T'. The weighted path length of T' is smaller than that of T; hence, T is not a Huffman tree.

It follows from Proposition 6 that $H \geqslant b/3$ and $b = O(H)$.

We can sort the initial set of elements in $O(\log n)$ time with n processors. Then, elements can be assigned to classes W_i in time $O(\log n)$ with $n/\log n$ processors. First, for every element e we compute a value diff(e); diff(e) equals to 1, if e and its predecessor belong to different classes, otherwise diff(e) is 0. Using prefix sums computation, we can identify elements that belong to the $\log n$ lowest not empty classes and distribute them into classes in time $O(\log n)$ with $n/\log n$ processors. After this, $\log n$ iterations of the algorithm **Detailed-Parallel-Huffman** can be performed in $O(\log n)$ time with $n/\log n$ processors as described in Section 4. Then the elements of the following $\log n$ not empty classes are distributed to classes and the next $\log n$ iterations of **Detailed-Parallel-Huffman** are performed. Proceeding in this way, we can perform $O(H)$ iterations in $O(H)$ time with $n/\log n$ processors. Thus we get

```
1   i := m
2   while i ⩾ 1 do
3     if (light[i] ≠ NULL)
4       t := meld(light[i], W_i[1])
5       remove W_i[1] from W_i
6       if (weight(t) > 1/2^(i−1)) OR (W_i ≠ ∅)
7         if (nextclass[i] ≠ i − 1)
8           nextclass[i − 1] := nextclass[i]
9           nextclass[i] := i − 1
10      if (weight(t) > 1/2^(i−1))
11        W_(i−1) := merge(W_(i−1), {t})
12      else
13        W_i := merge(W_i, {t})
14    forall j ⩽ ⌊l(W_i)/2⌋ pardo
15      W̃_i[j] := meld(W_i[2j − 1], W_i[2j])
16    W_(i−1) := merge(W_(i−1), W̃_i)
17    if (l(W_i) is odd)
18      light[nextclass[i]] := last(W_i)
19    i := nextclass[i]
```

Fig. 9. Huffman tree construction scheme (detailed description).

Theorem 3. *A Huffman tree of height H can be constructed in $O(H)$ time with n processors. A Huffman tree of height H can be constructed in $O(H)$ time with $n/\log n$ processors, if elements are sorted according to their weight.*

7. Conclusion

This paper describes the first optimal work approximate algorithms for the construction of Huffman codes. The algorithms have polynomially bounded errors. We also show that a parallel construction of an almost optimal code for n elements is as fast as the best known deterministic and probabilistic methods for sorting n elements. In particular, we can deterministically construct an almost optimal code in logarithmic time and with linear number of processors on CREW PRAM or in $O(\log n)$ time and with $n \log\log n/\log n$ processors on CRCW PRAM. We can also construct an almost-optimal tree with linear work in logarithmic time provided that the machine word size is $\log^2 n$. This is the first optimal work and the logarithmic time algorithm for that problem.

Our approach also leads to the first parallel algorithm that works in $O(H)$ time and with n processors. This gives the improvement of the construction of Huffman trees for the case when $H = o(\sqrt{n}\log n)$, where H is the maximum codeword length. The question of the existence of algorithms that deterministically sort polynomially bounded integers with linear time-processor product and achieve optimal speed-up remains widely open. It will be also interesting to know, whether efficient construction of almost optimal trees is possible without sorting initial elements.

Acknowledgements

We thank Larry Larmore for stimulating comments and discussions.

References

[1] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, Sorting in linear time? in: Proc. ACM Symposium on Theory of Computing, 1995, pp. 427–436.
[2] P. Bhatt, K. Diks, T. Hagerup, V. Prasad, T. Radzik, S. Saxena, Improved deterministic parallel integer sorting, Information and Computation 94 (1991) 29–47.
[3] Y. Han, X. Shen, Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs, SIAM J. Comput. 31 (2002) 1852–1878.
[4] T. Hagerup, Toward optimal parallel bucket sorting, Information and Computation 75 (1987) 39–51.
[5] T. Hagerup, Hybridsort revisited and parallelized, Information Processing Letters 32 (1989) 35–39.
[6] D.A. Huffman, A method for construction of minimum redundancy codes, Proc. IRE 40 (1951) 1098–1101.
[7] D. Kirkpatrick, T. Przytycka, Parallel construction of binary trees with near optimal weighted path length, Algorithmica (1996) 172–192.
[8] L. Larmore, T. Przytycka, Constructing Huffman trees in parallel, SIAM Journal on Computing 24 (6) (1995) 1163–1169.
[9] J. van Leeuwen, On the construction of Huffman trees, in: Proc. 3rd Int. Colloquium on Automata, Languages and Programming, 1976, pp. 382–410.
[10] R.L. Milidiu, E.S. Laber, A.A. Pessoa, A work efficient parallel algorithm for constructing Huffman codes, in: Proc. Data Compression Conference, 1999, pp. 277–286.
[11] G. Miller, J. Reif, Parallel tree contraction and its applications, in: Proc. 26th Symposium on Foundations of Computer Science, 1985, pp. 478–489.
[12] Y. Nekrich, Byte-oriented decoding of canonical Huffman codes, in: Proc. IEEE International Symposium on Information Theory, 2000, p. 371.
[13] Y. Nekrich, Decoding of canonical Huffman codes with look-up tables, in: Proc. IEEE Data Compression Conference, 2000, p. 342.
[14] S. Teng, The construction of Huffman equivalent prefix code in NC, ACM SIGACT 18 (1987) 54–61.
[15] L. Valiant, Parallelism in comparison problems, SIAM Journal on Computing 4 (1975) 348–355.