Approximating Huffman Codes in Parallel

Piotr Berman[1] *, Marek Karpinski[2] **, and Yakov Nekrich[2] ***

[1] Dept.of Computer Science and Engineering,
The Pennsylvania State University
[2] Dept. of Computer Science, University of Bonn.

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 $O(H)$ and with n processors. This represents a useful improvement since most practical situations satisfy $H = O(\log n)$.

Keywords: Parallel Algorithms, Approximation Algorithms, Huffman Codes

1 Introduction

A Huffman code for an alphabet $a_1, a_2, \ldots, a_n$ with weights $w_1, w_2, \ldots, w_n$ is a prefix code that minimizes the average codeword length, defined as $\sum_{i=1}^{n} w_i l_i$ so that $\sum 2^{-l_i} \leq 1$. The problem of construction of Huffman codes is closely related to the construction of Huffman trees (cf., e.g., [H51], [vL76]).

A problem of constructing a binary Huffman tree for a sequence $\bar{w} = w_1, w_2, \ldots, w_n$ consists in constructing a binary tree T with leaves,

* E-mail berman@cse.psu.edu. Research done in part while visiting Dept. of Computer Science , University of Bonn. Work partially supported by NSF grant CCR-9700053 and DFG grant Bo 56/157-1.

** E-mail marek@cs.uni-bonn.de. Work partially supported by DFG grants , DIMACS, and IST grant 14036 (RAND-APX).

*** E-mail yasha@cs.uni-bonn.de. Work partially supported by IST grant 14036 (RAND-APX).

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:

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

where l_i is the depth of the leaf corresponding to the element w_i. A problem related to the problem discussed in this paper is the construction of optimal alphabetic codes. In case of the alpahbetic codes we have an additional limitation that $l_i \leq l_j$ for all $i < j$.

The classical sequential algorithm, described by Huffman ([H51]) 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 [vL76]). However, no optimal parallel algorithm is known. Teng [T87] has shown that construction of a Huffman code is in a class NC. His algorithm, uses the parallel dynamic programming method of Miller et al. [MR85] 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 [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] 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 $wpl(T') \leq wpl(T) + n^{-k}$ for a fixed error parameter k (assuming $\sum w_i = 1$). We call n^{-k} an error factor. In practical situations the nearly optimal codes, corresponding to nearly optimal trees, are as useful as the Huffman codes, because compressing a file of polynomial size with an approximate Huffman code leads to the compression losses limited only by a constant. Kirkpatrick and Przytycka [KP96] propose several algorithms for that problem. In particular, they present an algorithm that works 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. In case of the optimal alphbetic codes the best known NC algorithm solves this problem in time $O(log^3 n)$ with $n^2 \log n$ processors (see [LPW93].

The problems considered in this paper were also partially motivated by a work of one of the authors on decoding the Huffman codes [N00b], [N00a].

In this paper we improve the before mentioned results by presenting an algorithm that works in $O(k \log n)$ time and 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 and this operation is repeated $O(\log n^k)$ times. We have developed a method for performing such a merging in a constant time.

We also further improve this result and design an algorithm that constructs almost-optimal codes in time $O(\log n)$ and with $n/\log n$ processors, provided that elements are sorted. This results in an optimal speed-up of the algorithm of van Leeuwen [vL76]. 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 radix sort algorithms we construct an optimal-work probabilistic algorithm that works in expected logarithmic time. We construct also a deterministic algorithm that works on a CRCW PRAM in $O(k \log n)$ 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 and with n processors, for H the height of Huffman tree. This is also an improvement of the algorithm of Larmore and Przytycka for the case when $H = o(\sqrt{n}\log n)$. We observe that in the most practical applications height of the Huffman tree is $O(\log n)$.

2 A Basic Construction Scheme

Our algorithm uses the following *tree* data structure. A single element is a tree, and if t_1 and t_2 are two trees, then $t = meld(t_1, t_2)$ is also a tree, so that $weight(t) = weight(t_1) + weight(t_2)$. 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. This tree can be proven to be optimal.

Kirkpatrick and Przytycka [KP96] presented a scheme for parallelization of a Huffman algorithm. The set of element weights $p_1, p_2, \ldots, p_n$ is partitioned into sorted arrays $W_1, \ldots, W_m$, such that elements of array W_i satisfy the condition $1/2^i \leq p < 1/2^{i-1}$. In this paper we view (sorted) arrays as an abstract data type with the following operations: extracting of subarray $A[a, b]$, measuring the array length, $l(A)$, and merging two sorted arrays, $merge(A, B)$. The result of operation $merge(A, B)$ is a

Algorithm Oblivious-Huffman

```
for i := m downto 1 do
    if l(W_i) = 1)
        W_{i-1} := merge(W_i, W_{i-1})
    else
        t := meld(W_i[1], W_i[2])
        W_i := merge(t, W_i(3, l(W_i)))
        a := l(W_i)
        b := ⌊(a − 1)/2⌋
        for i := 1 to b pardo
            W_i[i] := meld(W_i[2i − 1], W_i[2i])
        W_i := merge(W_i(1, b), W_i(2b + 1, a))
        W_{i-1} := merge(W_{i-1}, W_i)
```

Fig. 1. Huffman tree construction scheme

sorted array C which consists of elements of A and B. If we use n processors, then each entry of our sorted array has an associated processor.

Obviously elements of a class W_i can be melded in parallel before the elements of classes W_j , $j < i$ are processed. The scheme for the parallelization is shown on Figure 1, $W(a, b)$ denotes an array $W[a], W[a+1], \ldots, W[b]$. We refer the reader to [KP96] for a more detailed description of this algorithm.

Because the total number of iterations of algorithm **Oblivious-Huffman** equals to the number of classes W_i and the number of classes 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 [KP96] also describe an approximation algorithm, based on **Oblivious-Huffman**. In this paper we convert **Oblivious-Huffman** into an approximation algorithm in a different way. We replace each weight w_i with $w_i^{new} = \lceil w_i n^k \rceil n^{-k}$. Let T^* denote an optimal tree for weights $w_1, \ldots, w_i$. Since $w_i^{new} < w_i + n^{-k}$,

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

because all l_i are smaller than n. Hence $wpl(T^*, \bar{p}_{new}) < wpl(T, \bar{w}) + n^{-k+2}$. Let T_A denote the (optimal) Huffman tree for weights w_i^{new}. Then

$$wpl(T_A, \bar{w}) \leq wpl(T_A, \bar{w}^{new}) \leq wpl(T^*, \bar{w}^{new}) < wpl(T^*, \bar{w}) + n^{-k+2}$$

Therefore we can construct an optimal tree for weights w^{new}, then replace w_i^{new} with w_i and the resulting tree will have an error of at most n^{-k+2}.

If we apply algorithm **Oblivious-Huffman** to the new set of weights, then the number of iterations of this algorithm will be $\lceil k \log_2 n \rceil$, since new elements will be divided into at most $\lceil k \log_2 n \rceil$ arrays. An additional benefit is that we will use registers with polynomially bounded values. Note that in [KP96] PRAM with an unbounded register capacity was used. That advantage of our algorithm will be further exploited in section 4.

3 An $O(k \log n)$ Time Algorithm

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

Algorithm **Oblivious-Huffman** performs $k \log n$ iterations and in each iteration only the merge operations are difficult to implement in a constant time. All other operations can be performed in a constant time. We will use the following simple fact, described in [V75]:

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 $B = 1, \ldots, n$. 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 to implement operation $merge(W_i(1, b), W_i[2b+1, a])$ (line 11 of Figure 1) in a constant time.

Operation $merge(W_{i-1}, W_i)$ is the slowest one, because array W_i can have linear size and merging two arrays of size n requires $\log \log n$ operations in general case (see [V75]). In this paper we propose a method, that allows us to perform every merge of **Oblivious-Huffman** in a constant time. The key to our method is that at the time of merging, all elements in both arrays know their predecessors in other array, and can thus compute their positions in a resulting array in a 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 operation.

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 $r(p) = p \cdot 2^k$. We will denote by $r(i, c)$

a relative weight of the c-th element in array W_i, $w[e]$ will denote the weight of element e, and $pos[e]$ will denote the position of an element e in its array W_i, so that $W_i[pos[e]] = e$. To make description more convenient we say that in every array W_k $W_k[0] = 0$ and $W_k[l(W_k) + 1] = +\infty$ At the beginning we construct a list R of all elements, sorted according to their relative weight. We observe that elements of the same class W_k will appear in R in a non-decreasing order of their weight. We assume that whenever $e \neq e'$, $r(e) \neq r(e')$.Besides that, if leaf e and tree t are of a 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 a weight of t is bigger (smaller) than a weight of e.

We also compute for every leaf e and every class i the value of $pred(e, i) = W_i[j]$, s.t. $r(i, j) < r(e) < r(i, j + 1)$. In other words, $pred(e, i)$ is the biggest element in class i, whose relative weight is smaller than or equal to $r(e)$. To find values of $pred(e, j)$ for some j we compute an array C^j with elements corresponding to all leaves, such that $C^j[i] = 1$ if $R[i] \in W_j$ and $C^j[i] = 0$ otherwise and compute prefix sums for elements of C^j. Since a prefix sum can be computed in a logarithmic time with $n/\log n$ processors we can compute $O(\log n)$ necessary prefix sums in a logarithmic time with n processors (see [B97]).

We use an algorithm from Figure 2 to update values of $pred(e, i)$ for all $e \in W_{i-1}, \ldots, W_1$ and values of $pred(e, t)$ for all $e \in W_i$ and $t = i-1, \ldots, 1$ after melding of elements from W_i .

First we store the tentative new value of $pred(e, i)$ for all $e \in W_{i-1}, \ldots, W_1$ in array $temp$ (lines 1-3 of Figure 2). The values stored in $temp$ differ from the correct values by at most 1.

Next we meld the elements and change the values of $w[s]$ and $pos[s]$ for all $s \in W_i$ (lines 4-8 of Figure 2).

Finally we check whether the values of $pred(s, i)$ for $s \in W_1 \cup W_2 \cup \ldots \cup W_{i-1}$ 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 10-14 of Figure 2). In lines 15 and 16 we check whether the predecessors of elements in W_i have changed.

When the elements of W_i are melded and predecessor values $pred(e, i)$ are recomputed $pos[pred(W_i[j], i-1)]$ equals to the number of elements in W_{i-1} that are smaller than or equal to $W_i[j]$. Analogically $pos[pred(W_{i-1}[j], i)]$ equals to the number of elements in W_i that are smaller than or equal to $W_{i-1}[j]$. Therefore indices of all elements in the merged array can be computed in a constant time.

```
for a < i, b ≤ l(W_a) pardo
    s := W_a[b]
    temp[s] := ⌈pos[pred(s, i)]/2⌉

for c ≤ l(W_i)/2 pardo
    s := meld(W_i[2c − 1], W_i[2c])
    w[s] := w[W_i[2c − 1]] + w[W_i[2c]]
    pos[s] := c
    W_i[c] := s

for a < i, b ≤ l(W_a) pardo
    s := W_a[b]
    c := temp[s]
    if r(i, c) > r(a, b)
        c := c − 1
        if r(a, b + 1) > r(i, c + 1)
            pred(W_i[c + 1], a) := s
    pred(s, i) := W_i[c]
```

Fig. 2. operation meld

After melding of elements from W_i every element of $W_{i-1} \cup W_{i-2} \cup \ldots \cup W_1$ has two candidates for a predecessor of rank $i-1$. We can find the new predecessor of element e by comparing $pred(e, i)$ and $pred(e, i-1)$. The pseudocode description of an operation $merge(W_{i-1}, W_i)$ (line 12 of Figure 1) is shown on Figure 3.

Since all operations of the algorithm **Oblivious-Huffman** can be implemented to work in a constant time, each iteration takes only a constant time. Therefore we have

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

The algorithm described in the previous section can also be applied to the case of exact Huffman trees. The difference is that in case of exact Huffman trees weights of elements are unbounded and number of classes W_i is $O(n)$ in the worst case. However, it is easy to see that number of classes W_i does not exceed $H+2$ where H is the height of the resulting Huffman tree. We can sort elements and distribute them into classes in time $O(\log n)$ with n processors. We can then compute values of *pred* for classes $H, H-1, \ldots, H-\log n$ and perform first $\log n$ iterations of

```
do simultaneously:
for j ≤ l(W_{i-1}) pardo              for j ≤ l(W_i) pardo
    t := W_{i-1}[j]                       t := W_i[j]
    k := pos[pred(t, i)]                  k := pos[pred(t, i-1)]
    pos[t] := j + k                       pos[t] := j + k
    W_i[j + k] := t                       W_i[j + k] := t

for a < i, b ≤ l(W_a) pardo
    s := W_a[b]
    if (w[pred(s, i-1)] > w[pred(s, i)])
        pred(s, i) := pred(s, i-1)
```

Fig. 3. Operation $merge(W_i, W_{i-1})$

Oblivious-Huffman in time $O(\log n)$. Then, we compute values of $pred$ for the classes $H-\log n, H-\log n-1, \ldots, H-2\log n$ and perform the next $\log n$ iterations of the basic algorithm. Proceeding in the same manner we can perform H iterations in $O(H)$ time. Therefore we get

Theorem 2. *A Huffman tree can be constructed in time $O(H)$ with n processors, where H is the height of the Huffman tree.*

4 An $O(kn)$ Work Algorithm

In this section we describe a modification 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 $e \in W_i$ and values of $pred(e, i)$ for all $e \in W_{i-1}$ then merging can be performed in a 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 $O(n)$. We are also able to recompute values of $\overline{pred}$ in a constant time after each iteration. The technique that we use is remniscent of the parallel merge sort algorithm of Cole ([C88]).

For an array R we 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 $e \in W_l$, $l > i$ $(l < i)$ as the biggest element $\tilde{e}$ in $sample_{l-i-1}(W_i)$ $(sample_{i-l-1}(W_i))$, such that

$r(\tilde{e}) \leq r(e)$. Besides that we maintain the values of $\overline{pred}(e,i)$ only for $e \in sample_{l-i-1}(W_l)$. 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} elements. Obviously total number of values in $\overline{pred}$ is $O(n)$.

Now we will show how $\overline{pred}$ can be recomputed after elements in a class W_i are melded. Number of pairs (e,i) for which values $\overline{pred}(e,i)$ must be computed is $O(n)$, and we can assign one processor to every pair.

We denote by $sibling(e)$ an element with which e will be melded in **Oblivious-Huffman**. Consider an arbitrary pair (e,a), $e \in W_i$. First the value $\overline{pred}(e,a)$ is known, but the value of $\overline{pred}(s,i)$, where $s = sibling(e)$ may be unknown. We can set a tentative new value of $pred(e_m, a)$ where $e_m = meld(e,s)$ to $\overline{pred}(e,a)$.

Next we recompute the values of $\overline{pred}(s,i)$ for $s \in sample_{i-1}W_1 \cup sample_{i-2}W_2 \cup \ldots \cup sample_1 W_{i-1}$. Let $e_1 = pred(s,i)$, $e_2 = sibling(e_1)$ and $e = meld(e_1, e_2)$. The correct new values of $\overline{pred}(s,i)$ can be computed in a similar way as in section 3. If the relative weight of s is smaller than that of e, $\overline{pred}(s,i)$ is assigned to the element preceding e. Otherwise, we also compare the relative weight of s with the relative weight of the element following e. If the first one is bigger we set $\overline{pred}(s,i)$ to the element following e. We also can check whether the predecessors of elements in W_i are the correct ones at the same time. A pseudocode description of the parallel meld operation is shown on Figure 4.

When elements from W_i are melded the new elements will belong to W_{i-1}. Now we have to compute $\overline{pred}(e,a)$ in $sample_{i-a-2}(W_a)$ for every 2^{i-a-2}-th element of W_i. Suppose $\overline{pred}(e,a) = W_a[p \cdot 2^{i-a-1}]$. We can find the new "refined"value of $\overline{pred}(e,a)$ by comparing $r(e)$ with $r(W_l[p \cdot 2^{i-l-1} + 2^{i-l-2}])$. When the correct values of $\overline{pred}(e,i)$ $e \in sample_{l-i-1}(W_l)$ are known we can compute $\overline{pred}(e,i)$ for all 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 previous elements in $sample_{i-a-2}(W_a)$. Obviously e_n and e_p are in $sample_{i-a-1}(W_a)$ and $\overline{pred}(e,i)$ is between $\overline{pred}(e_p,i)$ and $\overline{pred}(e_n,i)$. New correct values of $\overline{pred}(e,i)$ can be found in a constant time.

Using the values of $\overline{pred}$ we can merge W_{i-1} and the melded elements from W_i in a constant time in the same way as described in section 3. A detailed description of the meld and merge operations for the modified algorithm will be given in the full version of the paper.

Since all other operations can also be done in a constant time we can perform $\log n$ iterations of **Oblivious-Huffman** in a logarithmic time. Therefore we get

```
for a < i, b ≤ l(sample_{i-a-1} W_a) pardo
    s := W_a[b · 2^{i-a-1}]
   temp[s] := ⌈pos[pred(s, i)]/2⌉

for c ≤ l(W_i)/2 pardo
    s := meld(W_i[2c − 1], W_i[2c])
    w[s] := w[W_i[2c − 1]] + w[W_i[2c]]
    pos[s] := c
    W_i[c] := s

for a < i, b ≤ l(sample_{i-a-1} W_a) pardo
    d1 := 2^{i-a-1}
    d2 := 2^{i-a-2}
    s := W_a[b · d1]
    c := temp[s]
    if r(i, c · d2) > r(a, b · d1)
        c := c − 1
        if r(a, (b + 1) · d1) > r(i, (c + 1) · d2)
            pred(W_i[(c + 1) · d2], a) := s
        else
            if r(i, (c + 1) · d2) < r(a, b · d1)
            c := c + 1
            if r(a, (b − 1) · d1) < r(i, (c − 1) · d2)
                pred(W_i[(c − 1) · d1], a) := W_a[(b − 1) · d2]
    pred(s, i) := W_i[c · d2]
```

Fig. 4. A melding operation for the improved algorithm

Theorem 3. *An almost optimal tree with error factor $1/n^k$ can be constructed in time $O(k \log n)$ and with $n/\log n$ processors, if elements are sorted according to their weight.*

We can combine the algorithm described above with different algorithms for the parallel bucket sort. Optimal time-processor product can be achieved under reasonable conditions.

Using a parallel bucket sort algorithm described in [H87] we can sort polynomially bounded integers in $O(\log n \log\log n)$ time and with $n/\log n$ processors on a priority CRCW PRAM. Using the algorithm described by Bhatt et al. [BDH$^+$91] we can also sort polynomially bounded integers in the same time and the processor bounds on arbitrary CRCW PRAM. Combining these results with our modified algorithm we get

Proposition 2 *An almost optimal tree with error $1/n^k$ can be constructed in $O(k \log n \log\log n)$ time and with $n/\log n$ processors on a priority CRCW PRAM or on an arbitrary CRCW PRAM.*

Applying an algorithm of Hagerup [H87] we get the following result

Proposition 3 *An almost optimal tree with error $1/n^k$ can be constructed for the set of n uniformly distributed random numbers with $n/\log n$ processors in time $O(k \log n)$ and with probability $1/C^{-\sqrt{n}}$ for any constant C .*

By using the results of Andersson, Hagerup, Nilsson and Raman [AHNR98], n integers in the range $0..n^k$ 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 $k \log n$. Finally [AHNR98] 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 $O(\log^{2+\varepsilon} n)$.

Proposition 4 *An almost optimal tree with error $1/n^k$ can be constructed with expected time $O(\log n)$ and expected work $O(n)$ on a CREW PRAM with word size $\log^{2+\varepsilon} n$.*

The last statement shows that a Huffman tree can be probabilistically constructed on a CREW PRAM with polylogarithmic word length.

5 Conclusion

This paper describes the first optimal work approximate algorithms for constructing 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 probabilistically construct an almost-optimal tree with linear expected work in logarithmic expected time provided that the machine word size is $\log^{2+\varepsilon} n$. This is the first optimal work and the logarithmic time algorithm for that problem.

Our approach also leads to 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. This gives the first parallel algorithm that works in $O(H)$ time and with n processors. In practical applications H is usually of order $O(\log n)$. 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.

Acknowledgments

We thank Larry Larmore for stimulating comments and discussions.

References

[AHNR98] Andersson, A., Hagerup, T., Nilsson, S., Raman, R., *Sorting in Linear Time?*, Journal of Computer and System Sciences **57** (1998), pp. 74–93.

[BDH+91] Bhatt, P., Diks, K., Hagerup, T., Prasad, V., T.Radzik, Saxena, S., *Improved deterministic parallel integer sorting*, Information and Computation **94** (1991), pp. 29–47.

[B97] Blelloch, G., *Prefix Sums and Their Applications*, Reif, J., ed, Synthesis of Parallel Algorithms, pp. 35–60, 1997.

[C88] Cole, R., *Parallel Merge Sort*, SIAM Journal on Computing **17** (1988), pp. 770–785.

[H87] Hagerup, T., *Toward optimal parallel bucket sorting*, Information and Computation **75** (1987), pp. 39–51.

[H51] Huffman, D. A., *A method for construction of minimum redundancy codes*, Proc. IRE,40 (1951), pp. 1098–1101.

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

[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.

[MR85] Miller, G., Reif, J., *Parallel tree contraction and its applications*, Proc. 26th Symposium on Foundations of Computer Science (1985), pp. 478–489.

[N00a] Nekrich, Y., *Byte-oriented Decoding of Canonical Huffman Codes*, Proc. Proceedings of the IEEE International Symposium on Information Theory 2000, (2000), p. 371.

[N00b] Nekrich, Y., *Decoding of Canonical Huffman Codes with Look-Up Tables*, Proc. Proceeding of the IEEE Data Compression Conference 2000 (2000), p. 342.

[T87] Teng, S., *The construction of Huffman equivalent prefix code in NC*, ACM SIGACT **18** (1987), pp. 54–61.

[V75] Valiant, L., *Parallelism in Comparison Problems*, SIAM Journal on Computing **4** (1975), pp. 348–355.

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