Approximating Optimal Binary Trees in Parallel

Piotr Berman* Marek Karpinski† Yakov Nekrich‡

July 4, 2001

Abstract

In this paper we present new results on an approximate construction of Huffman trees. Our algorithms match asymptotically the time and work needed by known sorting algorithms. For example, we show that an almost-optimal Huffman tree can be constructed in $O(\log n)$ time with n processors on a CREW PRAM improving the $O(\log n \log^* n)$ time and n processors result of Kirkpatrick and Przytycka. We also describe an $O(n \log\log n)$ work algorithm that works on a CRCW PRAM with $n/\log n$ processors. This is the first parallel algorithm for the problem with time and work being nearly optimal.

*Dept.of Computer Science and Engineering, The Pennsylvania State University. 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 berman@cs.uni-bonn.de

†Dept. of Computer Science, University of Bonn. Work partially supported by DFG grants , DIMACS and IST grant 14036 (RAND-APX). E-mail marek@cs.uni-bonn.de

‡Dept. of Computer Science, University of Bonn. Work partially supported by IST grant 14036 (RAND-APX). E-mail yasha@cs.uni-bonn.de

1 Introduction

A Huffman code for an alphabet $a_1, a_2, \ldots, a_n$ with weights $p_1, p_2, \ldots, p_n$ 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 closely related to the construction of Huffman trees.

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, 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 depth of the leave corresponding to the element w_i.

The classical sequential algorithm, described by Huffman ([H51]) can be implemented in $O(n \log n)$ time. If elements are sorted according to their weight, 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 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 multiplication of concave matrices. The best 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] proposed to investigate approximate, 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 $wpl(T') \leq wpl(T) + n^{-k}$. (assuming $\sum p_i = 1$). In practice nearly optimal codes are as useful as the Huffman codes. Kirkpatrick and Przytycka [KP96] propose several algorithms for this problem. In particular, they present an algorithm that works in $O(k \log n \log^* n)$ with n CREW processors and an $O(k^2 \log n)$ time algorithm that works with n^2 CREW processors.

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

In this paper we improve the previous results by presenting an algorithm that works in $O(k \log n)$ time with n processors. As we will see in the next section the crucial step in computing a nearly optimal tree is merging of two

sorted arrays and this operation must be repeated $O(\log n^k)$ times. We have developed a method for performing each such merging in constant time.

We will also describe an algorithm that constructs almost-optimal codes in time
$O(\log n \log\log n)$ with $n/\log n$ processors. The later algorithm works on a priority CRCW PRAM. This is the first parallel algorithm for the problem with time and work being nearly optimal.

2 Basic Construction Schema

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 the classical Huffman algorithm the set of trees is initialized with the set of weights. Then we consecutively meld 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 the 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 will 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 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.

Since in the Huffman algorithm lightest elements are processed first and sum of any two elements in class W_i is less than any element in class $W_j, j < i$, elements of the same class can be processed in parallel before the elements of classes with smaller indices are processed. The scheme for the parallelization is shown on Figure 1. 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 approximation algorithm in a different way. We replace each

Algorithm *Oblivious-Huffman*

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

Figure 1: Huffman tree construction scheme

weight p_i with $p_i^{new} = \lceil p_i n^k \rceil n^{-k}$. Let T^* denote an optimal tree for weights $p_1, \ldots, p_i$. 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 $wpl(T^*, \bar{p}_{new}) < wpl(T(\bar{p})) + n^{-k+2}$. Let T_A denote the (optimal) Huffman tree for weights p_i^{new}. Then

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

Therefore we can construct an optimal tree for weights p^{new}, than replace p_i^{new} with p_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 only with polynomially bounded values. Note that in [KP96] PRAM with unbounded register capacity was used. This advantage of our algorithm will be further exploited in section 4.

3 An $O(k \log n)$ algorithm

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

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

Statement 1 *If array A has a constant number of elements and array B has at most n elements, than arrays A and B can be merged in constant time 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. □

Statement 1 allows to implement operation $merge(W_i(1, b), W_i[2b+1, a])$ (line 11 of Figure 1) in constant time.

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

We will use the following notation. We will say that element e is of rank k, if $\lfloor \log w(e) \rfloor = k$, where $w(e)$ is the weight of e. Relative weight $r(p)$ of element p of rank k is $r(p) = p \cdot 2^k$. We will denote by $r(i, c)$ the 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 element e in its array W_i, so that $W_i[pos[e]] = e$. To make description more convinient we will 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 non-decreasing order of their weight. We will assume that whenever $e \neq e'$, $r(e) \neq r(e')$, this can be "enforced" by introducing a tie-breaking rule. 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, 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 weight of t is bigger (smaller) than 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 or equal than $r(e)$. To find values of $pred(e,j)$ for some j we compute 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. Prefix sum for any class k can be computed on an arithmetic circuit in linear depth and logarithmic time (see [B97]). In our case we have to solve $d = O(\log n)$ instances of prefix sum problems. Since total work for every single instance is linear we can pipeline all instances in such a way that all problems are solved in $O(d + \log n) = O(\log n)$ time with n processors. Thus we can iterate $j = 1, \ldots, k \log n$ and for each value of j we compute C^j and sent its content to the prefix sum circuit.

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 predecessor of elements in W_i have changed.

If number of elements in W_i is odd then last element of W_i must be inserted into W_i (line 11 of Figure 1). Using Statement 1 we can perform this operation in constant time. We can also correct values of $pred(e,i)$ in constant time with linear number of processors.

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 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 or equal to $W_{i-1}[j]$. Therefore indices of all elements in the merged array can be computed in constant time.

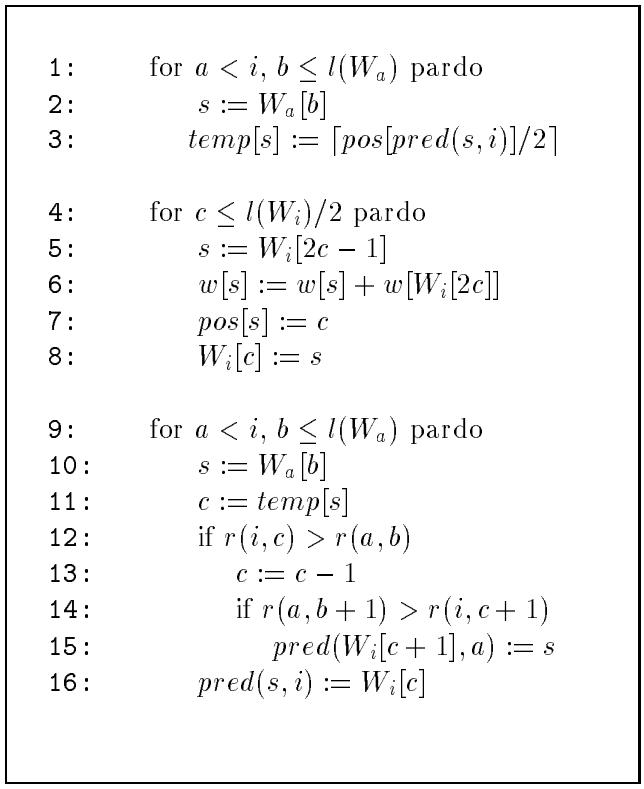

Figure 2: Melding operation

After melding of elements from W_i every element of $W_{i-1} \cup W_{i-2} \cup \ldots W_1$ has two predecessors 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 operation $merge(W_{i-1}, W_i)$ (line 12 of Figure 1) is shown on Figure 3.

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

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

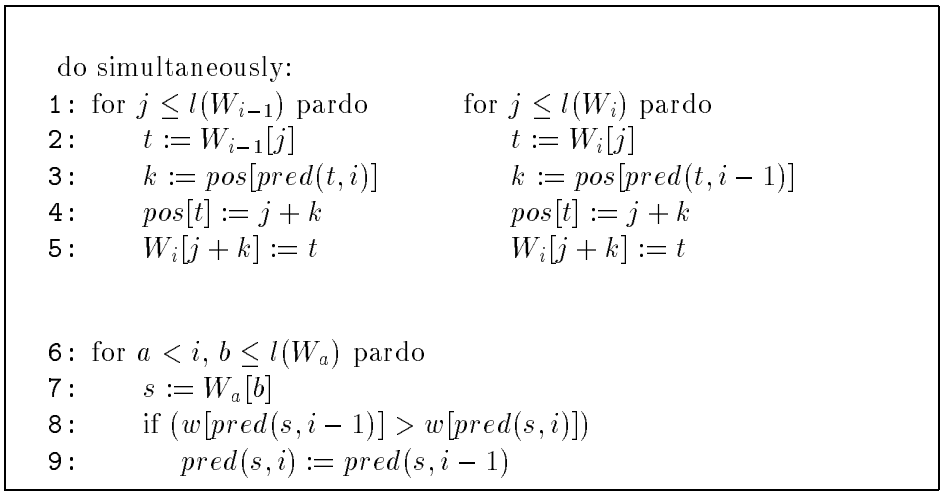

Figure 3: Operation $merge(W_i, W_{i-1})$

4 An $O(kn \log\log n)$-work algorithm

In this section we will describe a modification of the merging scheme, presented in the previous section. The modified algorithm works on a CRCW PRAM in $O(\log n \log\log n)$ time with $n/\log n$ processors.

The main idea of this modification is to compute statistics $pred(e,i)$ for only $n/\log n$ elements. Further in this section $L = \lfloor \log n \rfloor$. We construct an array S that contains all elements sorted according to their weight. $\overline{S}$ will denote an array, consisting of every L-th element of S, such that $\overline{S}[i] = S[iL]$, and $\overline{R}$ contains all elements of $\overline{R}$ sorted accorted to relative weigh (see Figure 4).

Then we compute the values of $\overline{pred(e,i)}$ for every element e of $\overline{S}$. For this purpose we use an array $\overline{C}^i$, such that $\overline{C}^i[j]$ equals to 1if $\overline{R}[j] \in W_i$. Arrays $\overline{C}^i$ can be constructed in logarithmic time with $n/\log n$ processors. Using the same procedure as in the previous section we can compute $pred(e,i)$ in logarithmic time with $n/\log n$ processors.

Let $\overline{W}_i$ denote a (sorted) array, consisting of every L-th element of W_i, so that $\overline{W}_i[s] = W_i[sL]$. Using $pred(e,i)$ we can merge every $\lfloor \log n \rfloor$-th element of W'_i with W'_{i-1}. When elements of W_i are melded every element from W'_i is melded with an element from $W_i - W'_i$. Predecessors of the melded elements can be computed in constant time in the same way as in the previous section.

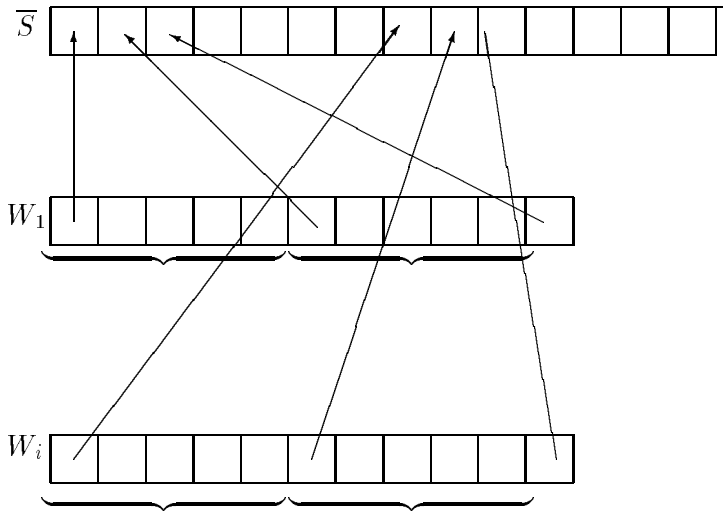

Figure 4: Structure of sample array $\overline{S}$

Now we can merge W_i and W_{i-1}, using $\overline{W}_i$ and $\overline{W}_{i-1}$.. Supposed $\overline{W}_i[l] < \overline{W}_{i-1}[s] < \overline{W}_i[l+1]$ and $\overline{W}_i[r] < \overline{W}_{i-1}[s+1] < \overline{W}_i[r+1]$. Then we have to merge $W_{i-1}[sL, (s+1)L]$ with $W_i[lL, (r+1)L]$. Using algorithm of Valiant (see [V75]) we can merge two arrays of size n_1 and n_2 $(n_1 < n_2)$ in $O(\log\log n_1)$ time with $n_1 + n_2$ processors. Therefore W_i and W_{i-1} can be merged in $O(\log\log\log n)$ time with $n/\log n$ processors. Hence, if elements in classes W_i are sorted, our algorithm can be implemented in $O(k \log n \log\log\log n)$ time with $n/\log n$ processors . Using a parallel bucket sort algorithm described in [H87] 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. [BDH$^+$91] we can also sort polynomially bounded integers with the same time and processor bounds on arbitrary CRCW PRAM. Combining these results with our modified algorithm we get

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

Hagerup [H87] describes an algorithm for sorting random uniformly distributed integers in $O(\log n)$ time with $n/\log n$ processors with probability

$1 - C^{-\sqrt{n}}$.

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

Theorem 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 $O(k \log n \log\log\log n)$ with probability $1/C^{-\sqrt{n}}$ for any constant C .*

5 Conclusion

In this paper we have described several algorithms for construction of almost optimal trees. These algorithms have a polynomially bounded error. The described algorithms are based on sorting initial set of elements. We show in this paper that construction of almost optimal tree for n elements is not slower than the best known deterministic algorithms for sorting n elements. In particular, we can construct an almost optimal tree in logarithmic time with linear number of processors in CREW PRAM model or in $O(\log n \log\log n)$ time with $n/\log n$ processors in CRCW PRAM model.

The question of existence of algorithms that can sort polynomially bounded integers with linear time-procesor product and achieve optimal speed-up remains open. It is also interesting, whether we can construct almost optimal trees without sorting of the initial set of elements.

Acknowledgments

We thank Larry Larmore for stimulating comments and discussions.

References

[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.

[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.

[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.