Parallel construction of binary trees with almost optimal weighted path length

D.G.Kirkpatrick and T. Przytycka

Technical Report 89-25

Parallel construction of binary trees with almost optimal weighted path length

D.G.Kirkpatrick and T.Przytycka

The University of British Columbia, Computer Science, Vancouver, B.C., Canada, V6T 1W5

Abstract

*We present sequential and parallel algorithms to construct binary trees with almost optimal weighted path length. Specifically, assuming that weights are normalized (to sum up to one) and error refers to the (absolute) difference between the weighted path length of a given tree and the optimal tree with the same weights, we present: an O(logn) time and $n\frac{loglogn}{logn}$ EREW processor algorithm which constructs a tree with error less than 0.172; an O(klogn log*n) time and n CREW processor algorithm which produces a tree with error at most $\frac{1}{n^k}$, and an $O(k^2 logn)$ time and n^2 CREW processor algorithm which produces a tree with error at most $\frac{1}{n^k}$. As well, we present two sequential algorithms: an O(kn) time algorithm which produces a tree with error at most $\frac{1}{n^{2k}}$ and O(kn) time algorithm which produces a tree with error at most $\frac{1}{2^{n2^k}}$ .The last two algorithms use different computation models.*

1. Introduction

One of the classical problems in communication is the construction of an optimal code. Let $V=\{v_1,...,v_n\}$ be a set of letters and $w(v_i)$ be the frequency of the occurrences of letter v_i in a word. The goal is to find a binary code for every letter of V, such that no code is a prefix of any other code and the average word length, defined as $\sum_{v\in V} l(v)w(v)$, where $l(v)$ denotes the length of code of letter v, is minimized. Equivalently, one can search for a binary tree T whose set of leaves is equal to V and which minimizes the following cost function $c(T)=\sum_{v\in V} l_T(v)w(v)$ where $l_T(v)$ denotes the length of the path from the root to the leaf v in the tree T. We call such a tree an *optimal tree*. The notion of an optimal tree

extends in an obvious way to the case when w is a *weight function* such that w: V->R^+. However in this case the problem can be reduced to our initial problem by dividing the weight of each element v, w(v), by $W=\sum_{v\in V} w(v)$. So hereafter we assume W=1. Note that this does not imply that the cost of an optimal tree is bounded by a constant. In fact it may achieve logn (for lower bounds on a weighted tree path see for example [M85]).

An optimal tree can be constructed in O(nlogn) sequential time by an algorithm due to Huffman ([Huff52]). It can be shown (see [H73]) that an optimal search tree (i.e. a tree which minimizes the cost function over all binary trees whose leaves occurs in a fixed order) whose leaves occurs in the order of increasing weights is also an optimal tree. This observation together with the parallel dynamic programming algorithm of Miller, Ramachandran and Kaltofen [MRK85] leads to an $O(\log^2 n)$ time n^6 processor parallel algorithm for construction of an optimal binary tree (see [T87] for a detailed description). An improved algorithm has been proposed by [AKLMT89] where the special structure of the dynamic programming problem has been used to produce a polylog algorithm using n^2 processors. This algorithm still does not achieve an optimal speedup, that is, the processor-time product does not match time complexity of the best known sequential algorithm. The question (cf. [AKLMT89]) of whether there exists a parallel algorithm which constructs an optimal tree in polylogarithmic time using $n^{2-\varepsilon}$ processors remains open. In the same paper, an approximate solution to the problem is proposed. Let T^*_V be an optimal tree for set V and T be an arbitrary tree with leaves equal to V. The *error of tree T*, ΔT, is defined as $c(T)-c(T^*_V)$. A tree is called *almost optimal* if its error is small. In [AKLMT89] an algorithm is presented which produces a tree T with $\Delta T\leq 1$ in O(logn) time using n/logn processors if the input sequence is sorted according to the weights. Also the parallel algorithm to produce an almost optimal binary search tree presented in the same paper can be used to construct a tree T with $\Delta T\leq 1/n^k$ in $O(k\log^2 n)$ time and with $n^2/\log^2 n$ processors.

In this paper, we present a family of parallel and sequential algorithms to construct an almost optimal binary tree. Each of our algorithms is an interpretation of the General Construction Scheme (GCS) defined section 4. One can think of GCS as an abstract algorithm whose meaning depends on an interpretation of basic functions used in its formulation. We present a general theorem which allows to estimate the error obtained when constructing a tree using an interpretation of GCS.

In the second section, we present the Basic Construction Scheme (BCS) which defines a family of bottom-up tree constructions. This scheme leads to efficient algorithms provided that there do not exist elements of very small weight. We prove that an algorithm which is an interpretation of BCS cannot produce a tree with error greater than 1. We also present a modification of BCS which reduces the maximal error to 0.172. In the third section, we define approximate sorting, merging of approximately sorted sequences, and approximation of a sequence by a sequence. Those notions are very helpful in estimating the error produced by some of our algorithms. In the fourth section, we define GCS which is a modification of BCS. This scheme leads to efficient algorithms for sets including elements of arbitrarily small weight. In the sixth section, we present a number of parallel interpretations of GCS. In particular,we give an O(logn) time and $n\frac{\log\log n}{\log n}$ EREW processor algorithm which constructs a tree with error at most 0.172 an O(klogn log*n) time and n CREW processor algorithm which produces a tree with error at most $\frac{1}{n^k}$, and an $O(k^2\log n)$ time and n^2CREW processor algorithm which produces a tree with error at most $\frac{1}{n^k}$. The algorithm obtained as the result of the second parallel interpretation of GCS achieves almost optimal speedup over the Huffman algorithm and produces a tree with a very small error. This result has been achieved by applying a *cascading sampling technique* - a new technique related to that used in Cole's merging sort algorithm [C86]. In the seventh section, we present two sequential algorithms which are also interpretations of GCS. The first of them produces a tree with error at most $\frac{1}{n^{2k}}$ and runs in O(kn) time assuming a RAM model with bounded register capacity. The second algorithm

produces a tree with error at most $\frac{1}{2^{n2^k}}$ and also runs in O(kn) time but it uses an integer sorting algorithm which assumes a RAM model of computation with unbounded register capacity.

2. Basic Construction Scheme (BCS)

An optimal tree can be constructed by the following simple algorithm due to Huffman:

While $|V| > 1$ do
Let v_1, v_2 be the pair of elements from V of smallest weight. Construct internal node u with v_1 and v_2 as children and define $w(u)=w(v_1)+w(v_2)$.
$V:=V-\{v_1,v_2\}\cup\{u\}$.

The Huffman algorithm, as described above, is highly sequential. We start by presenting an alternative way of constructing a tree isomorphic to the Huffman tree. Our algorithm produces the tree in a bottom-up fashion. At each stage of the algorithm we are dealing with sequences of roots of disjoint subtrees of the constructed tree. Each subtree has associated weight equal to the sum of weights of the elements in its leaves. If an element v belongs to a sequence X then *pred*(X,v) (resp., *succ*(X,v)) denotes the element which precedes v (resp., which occurs after v) in the sequence t and *dist*(X,v) denotes the number of elements (including v) which precede v in the sequence t. If u is an internal node of the constructed tree then *left*(u) (resp., *right*(u)) denotes left (resp., right) child of u and *parent*(u) denotes the parent of u.

For every $v \in V$ define *rank(v)* to be the integer such that $\frac{1}{2^{rank(v)-1}} > w(v) \geq \frac{1}{2^{rank(v)}}$. If rank(v) = i then we also say that *element v belongs to level i.* Let $I=\max\{i \mid |V_i|>0\}$. By convention, we think of a tree as having its root at the top and leaves at the bottom so a higher level is a level of elements of smaller rank. Let **sort** be an increasing sorting procedure, and **merge** be a procedure which given two sorted sequences produces a sorted sequence containing all elements from both sequences. Let **pair_elements** be a procedure which given a sequence U of even length produces a sequence C defined as

follows: Create a common father u for every pair of elements u_1, u_2 such that u_2=*succ*(U,u_1) and such that the value *dist*(U,u_1) is odd (define $w(u)=w(u_1)+w(u_2)$ and *left*(u)=u_1, *right*(u)=u_2). The order of elements in sequence C corresponds to the order of their children in sequence U. We use # to denote the concatenation operation on sequences. As we prove later, the following algorithm constructs a tree isomorphic to the Huffman tree:

1. Divide elements of V into sets $V_1,V_2,$....such that. v∈ V_i iff rank(v)=i;
2. For every i do **sort**(V_i);
3. Let $V_{i_1},...,V_{i_L}$ ($i_r<i_{r+1}$, i_L=I)be the list of nonempty sets; i:=i_L ; j=2i; U_j:=V_I; k:=L-1;
   - - - i is the index of currently processed level, i_k is the index of
   - - - the closest nonempty level to be processed
4. while k>0 or $|U_j|$ >1 do
5.1. if $|U_j|$ = 1 then U_{2i_k}:=U_j#V_{i_k}; i:=i_k; j:=2i; k:=k-1;
   - - - put the only element of U_j to the closest nonempty level
5.2. else c:=**pair_elements**(*first*(U_j),*succ*(*first*(U_j));
5.3. U_{j-1}:=**merge**(c,U_j-{*first*(U_j),*succ*(*first*(U_j)}); j:=j-1;
   - - - the parent ,c, of two first elements may have rank equal to i or i-1
   - - - so it is initially inserted into the sequence of elements of rank i;
5.4. if $|U_j|$ is even then c:=*pred*(*last*(U_j))#*last*(U_j) else c:=*last*(U_j);
5.5. C_{i-1}:=**merge**(**pair_elements**(U_j-c),V_{i-1});
   - - - pair elements of U_j except for the last element (or last two
   - - - elements); the last element may have rank equal i-1 (compare
   - - - step 5.3) so should not be paired at this point;
   - - - merge the resulting sequence with the elements of V_{i-1}
5.6 U_{j-1}:=**merge**(C_{i-1},c); j:=j-1;
   - - - merge the remaining (one or two) elements
5.7. i:=i-1; if $|V_i|$ > 0 then k:=k-1.

Theorem 1. The above algorithm produces a tree isomorphic to the Huffman tree.

Proof: Assume that in the above algorithm we replace the procedure **pair_elements** with a procedure which pairs elements of a sequence pair after pair from left to right. It suffices to prove that during such a pairing step we always pair two smallest elements (i.e. roots of two subtrees of smallest weight obtain a common parent). Note that all sequences occurring

in the algorithm are sorted and that for any i<k elements in V_i are greater than elements in V_k. Note also that the following statement is an invariant of "while" loop: For any $i<\lceil \frac{j}{2} \rceil$ elements in V_i are greater than elements in U_j. This is certainly true before the first iteration. So in step 5.2 we pair two smallest elements. After step 5.3, for any $i<\lceil \frac{j}{2} \rceil$ all elements of U_j except, possibly, the last one are smaller than any element of V_i. Also for any $i<\lceil \frac{j}{2} \rceil$ the last element of U_j is smaller than any element of V_{i-1}. Since the last element does not take part in the pairing step in line 5.5 so also in this step the smallest possible pair of elements is paired. After step 5.5 elements of C_{i-1} are smaller than elements of V_{i-2}. So elements of the sequence U_{j-1} created in step 5.6 are also smaller than elements of V_{i-2}. From this follows the invariant of "while" loop and consequently the fact that we always pair the smallest possible pair of elements. ■

Consider the above algorithm from a more general point of view. Replace the sorting procedure by a procedure ORDER which defines some (not necessarily sorted) order and the procedure **merge** by a procedure MERGE which given two sequences V, C produces a sequence of elements in V∪C with the property that it is sorted according to ranks (but not necessarily within the ranks). This produces a general scheme called the Basic Construction Scheme (BCS). One can think of BCS as an abstract algorithm (i.e. an algorithm whose meaning (interpretation) depends on the definitions of procedures ORDER and MERGE. We fix an interpretation by defining a representation of sequences and giving an interpretation for procedures ORDER, MERGE. Different interpretations define different tree construction algorithms. (We identify the algorithm defined by an interpretation of BCS with the interpretation itself.) In particular our initial algorithm is an interpretation of BCS. We call this interpretation *the Huffman tree algorithm* and denote by H. Note that although **pair_elements** has a fixed meaning, its implementation depends on the representation of sequences so, for consistency, we replace **pair_elements** by a procedure PAIR_ELEMENTS depending on the interpretation. When we refer to a

sequence (V_i, C_i or U_i) obtained by preforming BCS in interpretation A we use A as a superscript in the name of the sequence (V_i^A, C_i^A or U_i^A respectively). Note that in fact we can use different interpretations of procedure MERGE and PAIR_ELEMENTS in different steps of an interpretation of BCS. If that is the case, then we indicate this by adding a subscript equal to the number of the substep of step 5 to the name of the corresponding interpretation. For example, $MERGE_3$ denotes an interpretation of the MERGE procedure used in step 5.3.

We define the level of a sequence U_j to be equal to $\lceil \frac{j}{2} \rceil$. Elements of U_j whose rank is equal to the level of U_j form the *main subsequence* of the sequence U_j. Elements of U_j whose rank is less than the level of U_j form the *head* of the sequence U_j. Elements of U_j whose rank is greater than the level of U_j form the *tail* of the sequence U_j.

Lemma 2: Sequences U_j satisfy the following properties:

(i) if j is even then U_j has empty tail,

(ii) if j is odd then U_j has empty head,

(iii) there is at most one element in a tail,

(iv) there are at most two elements in a head,

(v) the element in a tail of a sequence has rank smaller by one than the level of the sequence,

(vi) if j is even and $|U_j| > 1$ then the element whose weight is equal to the sum of the weights of two first elements in the sequence has rank equal to or smaller by one than the level of the sequence.

Proof: The proof follows by induction on the level of a sequence. Consider first a sequence U_j of level I such that $j = 2I$. The sequence U_j has empty tail and head and all elements in U_j have rank equal to the level of U_j so (i) - (vi) are obviously true. Consider now sequence U_{2I-1}. If such a sequence is constructed then it is obtained from the sequence

U_{2I} by pairing two first elements and merging the resulting element with the rest of the elements in the sequence. It is obvious that U_{2I-1} has empty head and has a one-element tail. The rank of the element in the tail is equal to I-1. So (i) - (vi) hold also for j=2I-1.

Assume that (i) - (vi) hold for all sequences U_j of level less than i. Let U_j be a sequence of level i such that j=2i. We have to show (i), (iv), and (vi) in this case. To show point i) note that if sequence U_{j+1} has not been constructed then sequence U_j contains elements from V_i and the only element of the last nonempty sequence U_{2k} (k>i). If U_{j+1} has been constructed then U_j contains elements from V_i, C_i and at most two elements form U_{j+1}. Since elements in V_i and C_i have rank equal to i and by inductive hypothesis point v) elements in U_j have rank at most i so point i) follows. To show points (iv) and (vi) note that the elements in head may came either from the sequence U_{j+1} or, if U_{j+1} has not been constructed, from the last nonempty (one element) sequence U_{2k} (k>i). In the first case iii) follows from the fact that U_{j+1} has no head and in the second case it follows form the fact that in this case there is only one element in the head.

Assume now that j=2i-1. We have to show points (ii), (iii), and (v). Point ii) follows form the facts (iv) and (vi) which have been proven above. Point (iii) follows from point (i) by construction of BCS. Point v) follows from point (vi) by construction of BCS.■

Lemma 3: The cardinalities of all sequences occurring in the description of BCS and the number of iterations performed by an interpretation of BCS does not depend on the interpretation.

Proof: Note that in any interpretation of BCS we start with the sequences $V_1,...,V_I$ such that for each i=1,...,I the cardinality of V_i does not depend on the interpretation. The cardinality of any sequence constructed by an interpretation of BCS depends only on cardinalities of the sequences used for the construction and therefore, by induction, is independent of the interpretation. Similarly the number of iterations depends only on the cardinalities of the sequences and therefore is independent of the interpretation. ■

An important consequence of the above lemmas is that if T is a tree constructed by an interpretation of BCS then ΔT can be expressed in the following way:

Lemma 4: Let T be a tree obtained by some interpretation, A, of BCS and let

$$\Delta_j = \begin{cases} 0 & \text{-if } U_j \text{ has not been constructed or j is even and } |U_j|=1 \text{ or j is odd and } |U_j|\leq 2 \\ w(first(U_j^A))+w(succ(first(U_j^A)))-w(first(U_j^H))-w(succ(first(U_j^H)) & \text{-if } |U_j|>1 \text{ and j is even} \\ w(last(U_j^H))-w(last(U_j^A)) & \text{-if } |U_j| \text{ is odd and j is odd and greater than 1} \\ w(last(U_j^H))+w(pred(last(U_j^H)))-w(last(U_j^A))-w(pred(last(U_j^A))) & \text{-otherwise} \end{cases}$$

then $\Delta T = \sum_{j=1}^{2I} \Delta_j$.

Proof: At any stage of the algorithm we are dealing with the forest of the subtrees constructed so far. The roots of the subtrees belong either to recently constructed sequence U_j or to sequences V_i ($i<\lceil \frac{j}{2} \rceil$). Let $c(T_j^H)$ (resp., $c(T_j^A)$) be the cost of the forest such that the roots of the trees in this forest belong to sequence U_r^H (resp., U_r^A) where U_r^H (resp., U_r^A) is the last constructed sequence such that r>j. By definition of Δ_j we have for j<2I: $c(T_j^A)-c(T_j^H) = c(T_{j+1}^A)-c(T_{j+1}^H)+\Delta_{j+1}$. So $\Delta T = \sum_{j=1}^{2I} \Delta_j$. ■

We use the above lemmas to prove:

Theorem 5: If T is a tree obtained by an interpretation of BCS then $\Delta T < 1$.

Proof: Let $d_i=\Delta_{2i}+\Delta_{2i-1}$. Let $U^A_{2i} = u^A_1, u^A_2, \ldots u^A_k$ be the sequences constructed by an interpretation, say A, and $U^H_{2i} = u^H_1, u^H_2, \ldots u^H_k$ be the sequence produced by the interpretation H (the Huffman tree algorithm). Note that if U_{2i} is not constructed then U_{2i-1} is also not constructed and then $d_i=0$. Alternatively consider the following four cases:

1) Neither of the interpretations creates a tail. Then by Lemma 2 point (vi) we have $\frac{1}{2^i} \le w(u^A_1)+w(u^A_2) \le \frac{1}{2^{i-1}}$ and $\frac{1}{2^i} \le w(u^H_1)+w(u^H_2) \le \frac{1}{2^{i-1}}$. Then $\Delta_{2i} \le \frac{1}{2^i}$ and $\Delta_{2i-1} \le \frac{2}{2^i}$. So $d_i \le \frac{1}{2^{i-1}} + \frac{1}{2^i}$.

2) Both interpretations create a tail. We can interpret this case as a case when in both interpretations all but at most one (last) elements are paired. Since unpaired elements have rank equal to i it follows that $d_i \le \frac{1}{2^i}$.

3) Interpretation H creates a tail and interpretation A does not create a tail. Then $\frac{1}{2^i} \le w(u^A_1)+w(u^A_2) \le \frac{1}{2^{i-1}}$ and $\frac{1}{2^{i-1}} \le w(u^H_1)+w(u^H_2) \le \frac{1}{2^{i-2}}$. Assume that $w(u^H_1)+w(u^H_2) = \frac{1}{2^{i-1}} + z$. Then $\Delta_{2i} \le -z$ and $\Delta_{2i-1} \le \frac{1}{2^i} + \frac{1}{2^i} + z$. So $d_i \le \frac{1}{2^{i-1}}$.

4) Interpretation A creates a tail and interpretation H does not create a tail. Then $\frac{1}{2^i} \le w(u^H_1)+w(u^H_2) \le \frac{1}{2^{i-1}}$ and $\frac{1}{2^{i-1}} \le w(u^A_1)+w(u^A_2) \le \frac{1}{2^{i-2}}$. Assume that $w(u^A_1)+w(u^A_2) = \frac{1}{2^{i-1}} + z$. Then $\Delta_{2i} \le \frac{1}{2^i} + z$ and $\Delta_{2i-1} \le \frac{1}{2^i} - z$. So $d_i \le \frac{1}{2^{i-1}}$.

Note that the most expensive case is case 1. However if in this case $|U^A_{2i}|$ is even then $\Delta_{2i-1} \le \frac{1}{2^i}$. If $|U^A_{2i}|$ is odd (and greater than 2) then this step is followed by step of type 1 such that $\Delta_{2i-1} = -\Delta_{2i-2}$. So generally we can assume that $d_i \le \frac{1}{2^{i-1}} + z_{i+1} - z_i$ where $z_i = \Delta_{2i-1}$ if U_{2i-1} has even number of elements and 1) holds and $z_i = 0$ otherwise. So we have $\Delta T = \sum_{i=1}^{I} d_i = \sum_{i=1}^{I} \frac{1}{2^{i-1}}$. It remains to show that $d_1 = -z_2$. It is obvious that $\Delta_1 = 0$. If $n=2$ then obviously $\Delta T = 0$. So assume that $n>2$. Then V_1 has at most one element. If V_1 has one element then U_3 has at most 2 elements neither of them being in a tail (otherwise the sum of the weights would be greater than one). So in this case $d_1 = -z_2$. If V_1 has zero elements then U_3 either has four equal elements (then the result is obvious) or at most 3 elements. If it has 3 or fewer elements then U_2 has two elements and therefore $\Delta_2 = 0$. If

U_3 has 2 elements then one of them must be a tail so z_2=0. If it has 0,1 or 3 elements then 1) does not hold and z_2=0 as well. ■

From the proof of the above theorem it follows that vertices of higher level may potentially contribute more to the total error. So it is natural to ask, how far we can reduce the error if we run an approximate algorithm until we reach some level, say t, and then use the Huffman tree algorithm (i.e. when we reach level t we sort all sequences on levels t or higher and for the remaining iterations interpret MERGE as an exact merging procedure).

To see how much we can reduce the error using this approach consider first the following problem:

Let $W=w_1,w_2,....,w_k$ where $k<n$ and $w_1+w_2+....+w_k<1$. Let $1-(w_1+w_2+....+w_k) = L$. Consider a sequence $w_1,w_2,....,w_k,w_{k+1},....,w_n$ such that $w_{k+1},....,w_n=L$. Elements $w_{k+1},....,w_n$ are called *flexible elements*. Let u be the heaviest flexible element. The sequence $w_1,w_2,....,w_k,w_{k+1},....,w_n$ is called a *feasible extension of the sequence W* if, for every i,j such that $w_i,w_j \le u$, $w_i \le 2w_j$. Denote by $\mathfrak{F}(W)$ the set of all feasible extensions of W. Assume that $\mathfrak{F}(W)$ is nonempty. Two feasible extensions of W with the same set of flexible elements are considered to be equal. Let $w \in \mathfrak{F}(W)$. By C(w) we denote the cost of an optimal tree for the sequence w. Let $\Delta W = \max_{u\, w\in\mathfrak{F}(W)} C(u)-C(w)$. We are going to estimate the value ΔW.

Lemma 6. Let w: NxN–>R be a function defined as $w(r_1,r_2)=\frac{L}{2r_1+r_2}$. Then there exists such pair of integers r_1,r_2 such that $r_1+r_2=n-k$ and the sequence

$$w^* = w_1,w_2,....,w_k,\underbrace{2w(r_1,r_2),...,2w(r_1,r_2)}_{r_1},\underbrace{w(r_1,r_2),...,w(r_1,r_2)}_{r_2}$$

belongs to $\mathfrak{F}(W)$ and satisfies $C(w^*)=\min_{w\in\mathfrak{F}(W)} C(w)$. Furthermore there exists an optimal tree T* for the sequence w* in which all flexible leaves of weight $w(r_1,r_2)$ occur on one level, say h, and all flexible leaves of weight $2w(r_1,r_2)$ occur at level h-1.

Proof: Let $w = w_1,w_2,....,w_k,w_{k+1},....,w_n$ be a sequence which satisfy $C(w) = \min_{w\in\mathfrak{F}(W)} C(w)$. Let T by an optimal tree for w. Let u be the maximal flexible element of w.

First we show that there is an optimal tree in which all elements less than or equal to u occur on two consecutive levels. Assume that there are two elements $w_i,w_j \leq u$ such that w_i belongs to level h and w_j belongs to level h-s (s>1). Then the parent of w_i , say x, has weight at least twice as big as the weight of the smaller of its children. Since w is a feasible extension of $W=w_1,w_2,....,w_k$ it follows that $x \geq w_j$. Therefore we can switch w_j with x without increasing the cost of the tree. So there is an optimal tree for the sequence w, say T', in which all flexible elements occur at two consecutive levels.

Let the number of flexible elements on level h-1 of tree T' be equal to r_1 and the number of flexible elements on level h be equal to r_2. We do not increase the weight of the tree if we assign to flexible nodes on level h-1 weight $2w(r_1,r_2)$ and to flexible nodes on level h weight $w(r_1,r_2)$. In this way we obtain tree T* which satisfies the conditions stated in the lemma. ■

In Lemma 6 we have constructed a sequence w* such that $C(w^*) = \min_{w\in\mathfrak{F}(W)} C(w)$. Now we are going to construct a sequence w' (which is not necessarily a feasible extension of W) such that $C(w') \geq \max_{w\in\mathfrak{F}(W)} C(w)$. Let $u,w \in \mathfrak{F}(W)$ be such a pair of sequences such that u is obtained from w by reducing the value of a flexible element of w, say w_i, by some value x and increasing the value of another flexible element of w, say w_j, where $w_j \geq w_i$ by the same value x. Then we say that u is obtained from w by an *elementary shift of weight.*

Lemma 7: If u is obtained from w by an elementary shift of weight then $C(u) \leq C(w)$.

Proof: Assume that u is obtained from w by reducing the value of a flexible element of w, say w_i, by some value x and increasing the value of another flexible element of w, say w_j, where $w_j \geq w_i$, by the same value x. Let T be a tree which is optimal for the sequence w. Let T' be a tree obtained from T by changing weights of leaves corresponding to w_i and w_j by

subtracting and adding x respectively. Since $l_T(w_j) \le l_T(w_i)$ it follows that $c(T) \ge c(T')$. Furthermore the cost of T' is greater than or equal to the cost of an optimal tree for u. ■

Lemma 8. Let $w(r)=\frac{L}{n-k}$. Then the sequence $w'=w_1,w_2,....,w_k,w(r),w(r),....,w(r)$ satisfies $C(w') \ge \max_{w \in \mathfrak{F}(W)} C(w)$.

Proof: We show that for any feasible extension $w=w_1,w_2,....,w_k,w_{k+1},....,w_n$, there exists a sequence of elementary shifts of weight leading from w' to w. We define this sequence inductively. Let $W_i = w_1,w_2,....,w_k,w_{k+1}{}^i,....,w_n{}^i$ be the sequence obtained after i^{th} elementary shift of weight ($W_o = w' = w_1,w_2,....,w_k,w(r),w(r),....,w(r) = w_1,w_2,....,w_k,w_{k+1}{}^0,....,w_n{}^0$). If $W_i \ne w$ then preform the following shift of weight:

Let $w_j{}^i$ be the first flexible element such that $w_j{}^i < w_j$ and $w_t{}^i$ be the last flexible element such that $w_t{}^i > w_t$. Then define W_{i+1} as follows:

for $s \ne j,t$ $w_s{}^{i+1}=w_s{}^i$;

$w_j{}^{i+1}=w_j{}^i+\min(w_j-w_j{}^i,w_t{}^i-w_t)$;

$w_h{}^{i+1}=w_h{}^i-\min(w_j-w_j{}^i,w_t{}^i-w_t)$;

(shift $\min(w_{k+1}-w_t{}^i,w_t{}^i-w_n)$ weight from $w_h{}^i$ to $w_j{}^i$). It is obvious that a finite number of such steps convert w" into w. So by Lemma 7 $C(w') \le C(w)$. ■

Now we are ready to prove the following theorem:

Theorem 9: For any sequence $W=w_1,w_2,....,w_k$ where $k<n$ and $1-(w_1+w_2+....+w_k) = L>0$ holds $\Delta W \le L(3-2\sqrt{2})$.

Proof: Let T* be the optimal tree from Lemma 6 and let T" be a tree obtained from T* by replacing all flexible vertices with vertices of equal weight ($=\frac{L}{n-k}$). Of course $c(T") \ge c(T')$ where T' is an optimal tree for the sequence where all flexible vertices are equal. So

$$\Delta W \le c(T")-c(T^*) = r_2\left(\frac{L}{r} - \frac{L}{2r_1+r_2}\right) = \frac{r_2 L}{n-k} \cdot \frac{n-k-r_2}{2(n-k)-r_2}$$

Consider ΔW as a function on r_2. This function achieves its maximum, equal to $L(3-2\sqrt{2}) < 0.1716L$, for $r_2=r(2-\sqrt{2})$. So $\Delta W \leq L(3-2\sqrt{2})$.■

Corollary 10: If BCS is interpreted in such a way that an approximate construction is carried until some level t and the exact construction is carried for all levels greater than or equal to t, then $\Delta T \leq \frac{1}{2^{t-1}}+0.1716$.

Proof: Let $W=w_1,...,w_k$ be the sequence of weights of elements of rank smaller than t. Then for any interpretation of BCS sequence U_{2t-1} is a feasible extension of W. By the proof of Theorem 5 and Theorem 9 it follows immediately that $\Delta T \leq \frac{1}{2^{t-1}} + 0.1716$.■

Note that from the above corollary it follows that if we start the exact construction from level 10 then $\Delta T \leq 0.172$.

3. Approximate sorting and merging of approximately sorted sequences

A sequence $u_1,u_2,...u_k$ is *ε-sorted* if and only if $\max_i(\max_{j<i}(w(u_j)-w(u_i))\leq\varepsilon$.

A sequence $u_1,u_2,...u_k$ is an *ε-approximation* of the sequence $v_1,v_2,...v_k$ if for every i $w(u_i)+\varepsilon\geq w(v_i)\geq w(u_i)-\varepsilon$. Note that if $u_1,u_2,...u_k$ is a permutation of a sorted sequence $v_1,v_2,...v_k$ and is ε-sorted then it is also an ε-approximation of the sequence $v_1,v_2,...v_k$.

A procedure performs *ε-merging* if given two ε-sorted sequences produces an ε-sorted sequence.

Example 11. Consider the standard merging procedure applied to two ε-sorted sequences $C = c_1,c_2,...c_k$ and $V = v_1,v_2,...v_r$ (i.e. the procedure which inductively compares the first elements of the input sequences, removes the smaller of them, makes it the next element of the output sequence, and so on). Let $U=u_1,u_2,...u_{k+r}$ be the resulting sequence. To see that this procedure is ε-merging note that for any two elements $u_i \in V$,

$u_j \in C$ (resp., $u_i \in C$, $u_j \in V$) if $i<j$ then there exist an element $u_t \in V$ (resp., $u_t \in C$) such that $i \le t < j$ and $w(u_t) \ge w(u_i)$. Since V (resp., C) is ε-sorted it follows that $w(u_t) \le w(u_j) - \varepsilon$. So $w(u_i) \le w(u_j) - \varepsilon$.

Example 12: Consider a merging procedure which first divides input sequences V and C into m subsequences $V_1, V_2, \ldots V_m$ and $C_1, C_2, \ldots C_m$ respectively (some of them possibly empty) such that i^{th} subsequence contains elements whose weights are the interval $[\frac{i}{m}, \frac{i-1}{m})$ (recall that all elements are from the interval [0,1]). The merged sequence is equal to $V_1, C_1, V_2, C_2, \ldots, V_m C_m$. It is easy to see that this procedure first constructs $\frac{1}{m}$ - sorted sequences and then performs $\frac{1}{m}$-merging of those sequences.

An important property of an ε-merging procedure is given in the following lemma:

Lemma 13: Let $C^A = c^A{}_1, c^A{}_2, \ldots c^A{}_k$ and $V^A = v^A{}_1, v^A{}_2, \ldots v^A{}_r$ be two ε-sorted sequences. Assume also that V^A is an ε-approximation of a sorted sequence V^H and that C^A is an ε-approximation of a sorted sequence C^H. Let U^A be a sequence obtained by merging of V^A and C^A by an ε-merging procedure and let U^H be a sequence obtained by (exact) merging of V^H and C^H. Then U^A is a 2ε-approximation of U^H.

Proof: Let u^A and u^H be two elements with the same index in U^A and U^H respectively. Assume without loss of generality that the number of elements form C^A in the sequence U^A which precede element u^A is less than or equal to the number of elements from C^H which precede element u^H in U^H. Then there exists j such that the j^{th} element of list V^A (v_j^A) occurs in U^A before u^A or is equal to u^A and the j^{th} element of list V^H (v_j^H) occurs in U^H after u^H or is equal to u^H. But V^A is ε-approximation of V^H so $w(v_j^A) \ge w(v_j^H) - \varepsilon$. However $w(v_j^H) \ge w(u^H)$ and (since U^A is ε-sorted) $w(u^A) \ge w(v_j^A) - \varepsilon$. So $w(u^H) - w(u^A) \le 2\varepsilon$. ■

4. General Construction Scheme (GCS)

Note that if set V contains elements of very small weights then computing ranks may become a bottleneck for any interpretation of BCS. To obtain efficient implementations we divide all elements into two groups : heavy elements and light elements. More precisely let K(n) be an integer function of n. An element whose rank is greater than or equal to K(n) is called a *heavy element* and an element whose rank is less then K(n) is called *light element.* We assume that K(n) is chosen in such a way that it is easy to decide whether a given element is light or heavy and that it is easy to compute ranks of heavy elements. Denote the set of heavy (resp., light) elements of V by V_h (resp., V_l) and let $V' = V_h \cup \{u\}$ where u is an arbitrary element of V_l.

The General Construction Scheme (GCS) is a modification of the basic construction scheme. Roughly speaking, we use BCS for the set of heavy elements and then add to the resulting tree a subtree of light elements. (A similar approach was also taken in [AKLMT89] for the algorithm to construct an almost optimal binary search tree.)

GENERAL CONSTRUCTION SCHEME (GCS)

1. Divide set V into sets V_h and V_l.
2. Divide elements of V_h into sets $V_1, V_2, \ldots V_{K(n)}$ according to ranks
3. For every i compute ORDER(V_i).
4. Choose an arbitrary light element u and add it at the beginning of the sequence V_{K_n} (let $V'_{K(n)}$ be the resulting sequence)
5. Perform steps 3-5 of the BCS for set $V' = V_h \cup \{u\}$;
6. Replace u by an almost full binary tree[1] composed of all light elements.

Similarly to BCS, GCS can be treated as an abstract algorithm. Furthermore, every interpretation of BCS leads to an interpretation of GCS. Note that the error of a tree constructed by an interpretation of GCS is composed of two factors: the error resulting from an interpretation of BCS and the error from the light elements. We call the first

[1] An almost full binary tree is a tree whose leaves occurs on at most two different levels

component *construction error* and the second component *truncation error*. The total error can be approximated with the help of the following lemma:

Lemma 14: Let T' be an approximation of an optimal tree for V' and let T be the tree obtained from T' by replacing u by any binary tree of all light elements. Then $\Delta T=\Delta T'+\frac{n^2}{2^{K(n)}}$.

Proof: Since every leaf has depth at most n in T we have:

$$c(T) \le c(T') + n \sum_{v \in V_l} w(v) \le c(T') + \frac{n^2}{2^{K(n)}} \le c(T^*_{V'}) + \Delta T' + \frac{n^2}{2^{K(n)}}$$

so $c(T) \le c(T^*_V) + \Delta T' + \frac{n^2}{2^{K(n)}}$. ■

5. Parallel interpretations of GCS

As we have mentioned before, GCS presented in the previous section can be divided into two parts: computing a tree for heavy elements and modification of the resulting tree with a subtree of light elements. The second part can be implemented in O(logn) time with n/logn EREW PRAM processors independently of the choice of K(n). The first part involves O(K(n)) iterations of step 5 of BCS so in order to obtain an efficient parallel algorithm it is natural to chose $K(n)=k\lceil \log n \rceil$ where k is some integer constant. With this definition of K(n), truncation error is bounded by $\frac{1}{n^{k-2}}$. So we concentrate on an interpretation of the first part (i.e. on an interpretation of the BCS for heavy elements).

5.1. O(logn) time $n\frac{\log\log n}{\log n}$ EREW processors parallel interpretation of GCS with construction error bounded by 0.172

By Theorem 5, any interpretation of BCS gives a construction error bounded by 1. By Corollary 9 any interpretation of BCS in which starting from level t we apply the Huffman tree algorithm leads to a construction error bounded by $\frac{1}{2^{t-1}} + 0.1716$. So, in

particular, we can start with arbitrarily ordered sequences and for sequences on levels smaller than t interpret MERGE as concatenation of two sequences. Then sort all sequences and finish the construction interpreting MERGE as an exact merging procedure. We can compute ranks for heavy elements in O(loglogn) time with O(n) EREW processors by a binary search or (simulating $\frac{\log n}{\log\log n}$ processors by one processor) in O(logn) time with $n\frac{\log\log n}{\log n}$ EREW processors. A concatenation step and a pairing step can be implemented in O(1) time with n processors or in O(logn) time with $\frac{n}{\log n}$ EREW processors (by applying Brent's scheduling principle; since we can mantain information about the position of every element within a list, the processors allocation is not a problem). Since there are at most 2^t elements on levels t or higher the "exact" part of the algorithm can be implemented in O(tlogt) time using Cole's parallel merge sort and Valiant's parallel merging procedure. Therefore the entire algorithm can be implemented in O(logn+tlogt) time with $n\frac{\log\log n}{\log n}$ EREW processors. In particular if we assume t=k=11 we obtain:

Corollary 15: A tree whose cost differs at most 0.172 by from the cost of an optimal tree can be constructed in O(logn) time with $n\frac{\log\log n}{\log n}$ EREW processors.

5.2 A O(klogn log*n) time n processor parallel interpretations of GCS with zero construction error

We represent sequences in the form of lists such that for every element its position within the list is known. To achieve zero construction error we can implement ORDER with Cole's parallel sorting algorithm [C86] which runs in O(logn) time using n CREW processors and MERGE with Valiant's merging algorithm ([V75],[BH86]) which merges two sorted arrays of sizes n_1, n_2 $(n_1 \le n_2)$ in $O(\log\log n_1)$ time with n_1+n_2 CREW processors. This permits a straightforward implementation of step 5 of BCS in O(loglogn) time with n processors. By the definition of heavy elements, we have O(klogn) iterations. So this implementation runs in O(klogn loglogn) time using n processors.

To reduce the running time to O(logn log*n) we introduce a *cascading sampling technique*. The main idea is to precede the sequence of mergings performed by the algorithm by a preprocessing step. The preprocessing step consists of log*n sampling steps. Informally, in the i^{th} sampling step every $(2^i)^{th}$ element from every level is divided by half and sent to the level below. Then, among the elements which arrived at given level every second element is divided by half and sent down another level, and so on until a level which is $\lceil \log^{(i)} n \rceil$ levels[1] below is reached or no elements are left. Formally, let V^k_i be the sorted sequence of elements on level i after the k^{th} sampling step. Initially $V^k_i = V_i^0 = V_i$. For every sequence V^k_i define a family of sorted sequences $S^k_{i,1}, S^k_{i,2}, \ldots$ in the following way:

$$S^k_{i,t} = \{x \mid \exists v \in V^{k-1}_i \text{ s.t. } w(x) = \frac{1}{2^i} w(v) \text{ and } dist(V^{k-1}_i, v) = 2^{i+t-1} m \text{ for some integer } m\};$$

Then V^k_i is defined as $V^{k-1}_i \cup \bigcup_{t \le \lceil \log^{(i)} n \rceil} S^{k}_{i,t}$ $\quad j+t=i$

Elements produced in the sampling process are called *sampling elements*. All other elements are called *real*. To simplify the description we assume that we add sampling elements at the beginning and at the end of every list. For each sampling element, x, there is a unique element in the next higher level whose weight has been divided by half to obtain x. This element is called the *source element* for element x. A source element may also be a sampling element. For any sampling element, x, define *source*(x) to be equal to the source element of x. Sampling elements which have been generated in the same sampling step form, using pointer *source*, sequences. The element generated in a different sampling step which is pointed by the pointer *source* of the last element of such a sequence is said to *originate the given sequence*. Let u_1, u_2, be a pair of sampling elements from the same level such that there are no other sampling elements between them. The subsequence of elements which lies between u_1 and u_2 is called *a basic sequence*. The sequence of elements from the higher level which lies between the source of u_1 and the source of u_2 is called *a gap*.

[1] $\log^{(i)} n$ denotes loglog....logn (taking i times log)

We say that the gap defined by u_1 and u_2 corresponds to the basic sequence defined by u_1 and u_2. Note that to merge two sequences on successive levels it suffices to merge each basic sequence with its corresponding gap. The number of elements in a gap is called the *size of the gap*. Two important properties of the sampling process are given by the following lemmas.

Lemma 16. After log*n sampling steps, each gap size is bounded by $2^{\log^* n+2}$.

Proof: In the proof of the lemma we use the following simple facts:

Fact 1: a) $\lfloor x \rfloor \le \lfloor \frac{x}{m} \rfloor m+m$; b) $\lceil x \rceil \le \lfloor \frac{x}{m} \rfloor m+m+1$.

Consider an interval [a,b] where $a,b \in (\frac{1}{2^{j+1}}, \frac{1}{2^j}]$. Let us restrict our attention only to those elements whose samples may belong to this interval. So we consider only elements v from levels i=1,2,...,j such that $v \in V_{j-t}$ and $w(v) \in [2^t a, 2^t b]$ We call those elements *interval elements*. Let n^k_i be the number of interval elements on level i after k sampling steps (for simplicity we assume that n^k_i is also defined for i less than 1 and then it is equal to 0). The proof of the lemma is based on the following fact:

Fact 2: $n^k_i \le 2^k n^k_{i+1} + 3\lceil \log^{(k)} n \rceil + 2^{k+1}$.

Proof of Fact 2: We prove Fact 2 by induction on k. For k=1 we have:

$$n^1_i \le n^0_i + \lceil \tfrac{1}{2} n^0_{i-1} \rceil + \ldots + \lceil \frac{1}{2^{\lceil \log n \rceil}} n^0_{i-\lceil \log n \rceil} \rceil \quad \text{and}$$

$$n^1_{i+1} \ge n^0_{i+1} + \lfloor \tfrac{1}{2} n^0_i \rfloor + \ldots + \lfloor \frac{1}{2^{\lceil \log n \rceil}} n^0_{i-\lceil \log n \rceil+1} \rfloor$$

$$\text{so } n^1_i - 2^1 n_{i+1} \le (n^0_i - 2\lceil \tfrac{1}{2} n^0_i \rceil) + (\lceil \tfrac{1}{2} n^0_{i-1} \rceil - 2\lfloor \tfrac{1}{4} n^0_{i-1} \rfloor) + \ldots + \lceil \frac{1}{2^{\lceil \log n \rceil}} n^0_{i-\lceil \log n \rceil} \rceil \le$$

$$\le (2+1)\lceil \log n \rceil + \lceil \frac{1}{2^{\lceil \log n \rceil}} n^0_{i-\lceil \log n \rceil} \rceil \quad \text{(by Fact 1)}$$

$$\le 3\lceil \log n \rceil + 1.$$

Assume now that $n^{k-1}_i \le 2^{k-1} n^{k-1}_{i+1} + 3\lceil \log^{(k-1)} n \rceil + 2^k$. But

$$n^k_i \le n^{k-1}_i + \lceil \tfrac{1}{2^k} n^{k-1}_{i-1} \rceil + \ldots + \lceil \frac{1}{2^{k+\lceil \log^{(k)} n \rceil - 1}} n^{k-1}_{i-\lceil \log^{(k)} n \rceil} \rceil \quad \text{and}$$

$$n^k_{i+1} \ge n^{k-1}_{i+1} + \lfloor \tfrac{1}{2^k} n^{k-1}_i \rfloor + \ldots + \lfloor \frac{1}{2^{k+\lceil \log^{(k)} n \rceil - 1}} n^{k-1}_{i-\lceil \log^{(k)} n \rceil+1} \rfloor.$$

Therefore

$$n^k{}_i - 2^k n^k{}_{i+1} \le (n^{k-1}{}_i - 2^k \lfloor \tfrac{1}{2^k} n^{k-1}{}_i \rfloor) + (\lceil \tfrac{1}{2^k} n^{k-1}{}_{i-1} \rceil - 2^k \lfloor \tfrac{1}{2^{k+1}} n^{k-1}{}_{i-1} \rfloor) + \ldots$$
$$+ \lceil \tfrac{1}{2^{k+\lceil \log^k n \rceil - 1}} n^{k-1}{}_{i-\lceil \log^k n \rceil} \rceil$$

$$\le 2^k + 1 + (\lceil \tfrac{1}{2^k} n^{k-1}{}_{i-1} \rceil - 2 \lfloor \tfrac{1}{2^{k+1}} n^{k-1}{}_{i-1} \rfloor + 2 \lfloor \tfrac{1}{2^{k+1}} n^{k-1}{}_{i-1} \rfloor - 2^k \lfloor \tfrac{1}{2^{k+1}} n^{k-1}{}_{i-1} \rfloor) +$$
$$\ldots + \lceil \tfrac{1}{2^{k+\lceil \log(k) n \rceil - 1}} n^{k-1}{}_{i-\lceil \log(k) n \rceil} \rceil$$

$$\le 2^k + 1 +$$
$$2 + 1 - (2^k - 2) \lfloor \tfrac{1}{2^{k+1}} n^{k-1}{}_{i-1} \rfloor + \ldots +$$
$$2 + 1 - (2^k - 2) \lfloor \tfrac{1}{2^{k+\lceil \log(k) n \rceil - 1}} n^{k-1}{}_{i-\lceil \log(k) n \rceil + 1} \rfloor +$$
$$\lceil \tfrac{1}{2^{k+\lceil \log(k) n \rceil - 1}} n^{k-1}{}_{i-\lceil \log(k) n \rceil} \rceil$$

$$\le 2^k + 3\lceil \log(k) n \rceil - 2 + \lceil \tfrac{1}{2^{k+\lceil \log(k) n \rceil - 1}} n^{k-1}{}_{i-\lceil \log(k) n \rceil} \rceil -$$
$$2^{k-1} \lfloor \tfrac{1}{2^{k+\lceil \log(k) n \rceil - 1}} n^{k-1}{}_{i-\lceil \log(k) n \rceil + 1} \rfloor .$$

But by the inductive hypothesis we have:

$$n^{k-1}{}_{i-\lceil \log^k n \rceil + 1} \ge \frac{n^{k-1}{}_{i-\lceil \log(k) n \rceil} - 3\lceil \log^{(k-1)} n \rceil - 2^k}{2^{k-1}}.$$

So
$$2^{k-1} \lfloor \tfrac{1}{2^{k+\lceil \log(k) n \rceil - 1}} n^{k-1}{}_{i-\lceil \log(k) n \rceil + 1} \rfloor$$
$$\ge 2^{k-1} \lfloor \frac{1}{2^{k+\lceil \log(k) n \rceil - 1}} \frac{n^{k-1}{}_{i-\lceil \log(k) n \rceil} - 3\lceil \log^{(k-1)} n \rceil - 2^k}{2^{k-1}} \rfloor$$
$$\ge \lfloor \frac{n^{k-1}{}_{i-\lceil \log(k) n \rceil} - 3\lceil \log^{(k-1)} n \rceil - 2^k}{2^{k+\lceil \log(k) n \rceil - 1}} \rfloor - 2^{k-1} \qquad \text{(by Fact 1(a))}$$
$$\ge \lceil \tfrac{1}{2^{k+\lceil \log(k) n \rceil - 1}} n^{k-1}{}_{i-\lceil \log(k) n \rceil} \rceil - \lceil \frac{3\lceil \log^{(k-1)} n \rceil + 2^k}{2^{k+\lceil \log(k) n \rceil - 1}} \rceil - 2^{k-1} - 2.$$

Therefore

$$n_i{}^k - 2^k n_{i+1}{}^k \le 2^k + 3\lceil \log(k) n \rceil - 2 + \lceil \frac{3\lceil \log^{(k-1)} n \rceil + 2^k}{2^{k+\lceil \log(k) n \rceil - 1}} \rceil + 2^{k-1} + 2 \le 2^k + 3\lceil \log(k) n \rceil + 1 + 2^{k-1}$$

$$\le 3\lceil \log(k) n \rceil + 2^{k+1}. \blacksquare$$

Now we can continue the proof of the Lemma 16. Let a,b be the real numbers which bound weights of elements in a gap, say on level j. Consider the last sampling step. In this step

level j receives only elements from level j-1. After the last sampling step in any gap on level j there are at most $2^{\log^* n}$ elements from level j. By Fact 2, level j-1 has at most $2^{2\log^* n} + 3\lceil \log(\log^* n) n \rceil + 2^{\log^* n+1} < 2^{2\log^* n+1}$ interval elements. Every $2^{\log^* n}$ of them is sent to level j. So in a gap on level j there are at most $2^{\log^* n+1}$ elements originated at level j-1 and $2^{\log^* n}$ elements from level j. So the number of elements in the gap is bounded by $2^{\log^* n+2}$. ■

Lemma 17: After log*n sampling steps the total number of elements is bounded by 3n.

Proof: Let a_i be the number of elements after i^{th} sampling step. Obviously:

$$a_0 = n \quad \text{and}$$

$$a_i \le a_{i-1} + \frac{a_{i-1}}{2^i} = a_{i-1}(1+\frac{1}{2^i})$$

But it is easy to check (by induction) that $a_i \le 3n(1 - \frac{1}{2^i})$. ■

Now we show how to implement a sampling process in O(log*n logn) time with n processors. By Lemma 16, the gap size is bounded by $2^{\log^* n+2}$. Since to merge two sequences it suffices to merge every basic sequence with the corresponding gap, we can implement one iteration of step 5 of BCS in O(log*n) time using Valiant's merging algorithm. This allows the entire algorithm to be implemented in O(logn log*n) time on a CREW PRAM using n processors. The details of such an implementation are given below.

IMPLEMENTATION OF SAMPLING PROCESS: Since, by Lemma 17, in any sampling step we have at most 3n elements we have one processor per constant number of elements. However, we must perform some computation which assigns elements to processors. Initially we have one processor per real element. Inductively assume that that processors 1,...,j have been assigned r elements and processors j+1,...,n have assigned r-1 elements. In the k^{th} sampling step the processor associated with a given element checks whether this element originates a sequence of sampling elements and if so, how long this sequence is.

(Note that in the k^{th} sampling step the l^{th} element in a sequence originates a chain of length $\max_i\{l \text{ is divided by } 2^{k+i-1}\}$). Since the length of a sampling sequence is bounded by logn the generation of such a sequence can be done in O(logn) time using one processor per sequence.

To assign processors to new sampling elements we number those elements in such a way that if v and u are two sampling elements and v is originated at level higher than u, or v and u are originated at the same level but the element which originated v proceeds the element which originated u, then v receives smaller number than u. If u and v are originated by the same element and u is on higher level than v then u receives smaller number than v. Note that every "old" element knows the number of new sampling elements it has originated. So using prefix sum computation we can compute for every "old" element number of new sampling elements preceding it. Since for every new sampling element its position in the sequence of new sampling elements originated by the same element, say x, is known in order to obtain the number of given new sampling element, it suffices to add this position number to the number of new sampling elements preceding x. After numbering all new sampling elements element numbered m is assigned to the processor numbered (j+m)modn. This solves the problem of processor allocation.

It remains to insert new sampling elements into proper position of the sequences produced in the previous sampling step. This can be done by a global sorting algorithm. Note that it is important that the sorting procedure which we are using is stable. If it is not, we can number all elements (in a way similar to the way we have numbered all new sampling elements) and sort lexicographically pairs (number of the element, its value). Since the number of sampling steps is O(log*n), the whole sampling process can be implemented in O(log*n logn) time with n processors.

To implement efficiently the rest of the algorithm we represent the sequences that result from the sampling phase in the following way:

REPRESENTATION OF SEQUENCES: Each sequence is represented by a list formed by pointers *succ*. Also for every element of a sequence the following information is given:

nr_sa - number of sampling elements in the sequence which precede given element,
nr_r - number of real elements in the sequence which precede given element,
left_sa - pointer to the sampling element closest to the left,
left_r - pointer to the real element closest to the left,
left_so - pointer to the element closest to the left which is a source element,
nr_so - number of sampling elements in the sequence which precede given element.

and for every sampling element we have:

source - pointer to the source of the given element.

To simplify the description we will also assume the following pointers (these pointers can be computed in parallel using the information provided by the pointers defined above):

right_r -points to the real element closest to the right,
pred - is the reverse of *succ* pointer,
$source^{-1}$ - is the reverse of pointer *source*,
dist - is the sum *nr_so*+*nr_r*.

INITIALIZATION: Pointers *source* are build in the sampling process. All the other information can be computed by applying the parallel prefix technique.

PAIR ELEMENTS: We should pair every real element x with even value *nr_r* with the element pointed by *left_r*(x). However between such a pair of real elements there can occur sampling elements. We first rearrange the sequence (not changing the relative order of real or sampling elements) in such a way that real elements which are going to be paired occur as consecutive elements. Furthermore positions of sampling elements are chosen in such a way that the list obtained by pairing real elements and doubling of weights of sampling elements is sorted. The details are as follows:

1. For every sampling element x compute y1(x):=*left_r*(x); y2(x):=*right_r*(x);
   Every sampling element x for which y1(x) and y2(x) are non none checks parity of *nr_r*(*left_r*(x)) If it is odd then x appears between two real elements which should be paired. Such a sampling element is called *skipped*. For every skipped element the following two tests are performed:

$$\text{AFTER}(x) \Leftrightarrow 2w(x) \leq w(y1(x)) + w(y2(x))$$

INSERT(x) $\Leftrightarrow$ AFTER and (w(y1(x))+ w(y2(x)) < 2w(*succ*(x))
or *succ*(x) =*right_r*(x))

The first test says whether the two real elements should be inserted somewhere after x and the second test says whether the two real elements should be inserted immediately after x. However it may happened that the pair of real elements should be inserted before all sampling elements which occur between them. To detect this case every real element y performs the following test:

FIRST(y) $\Leftrightarrow$ *nr_r*(y) is odd and w(y)+ w(*right_r*(y)) < 2w(*succ*(y))

2. For every skipped element x update *left_r* and *nr_r* :

   if AFTER(x) then *left_r*(x):=*left_r*(*left_r*(x)); *nr_r*(x):=*nr_r*(x)-1

   otherwise *left_r*(x):=y2(x); *nr_r*(x):=*nr_r*(x)+1

3. Rearrange the elements on the list:

   For every skipped element x for which INSERT(x) is true do:

   succ(*pred*(y1(x))):=*succ*(y1(x)); *succ*(*pred*(y2(x))):=*succ*(y2(x));

   succ(y1(x)):=y2; *succ*(y2(x)):=*succ*(x); *succ*(x):=y1(x);

   For every real element y for which FIRST(y) is true do:

   succ(*right_r*(y):=*succ*(y);*succ*(*pred*(*right*(y))):=*succ*(*right*(y)); *succ*(y):=*right_r*(y)

4. For every real element which changed its place compute *left_sa*, *nr_sa*:

   Let x be a skipped element for which INSERT was true then

   left_sa(*succ*(x)), *left_sa*(*succ*(*succ*(x))):=x;

   nr_sa(*succ*(x)), *nr_sa*(*succ*(*succ*(x))) :=*nr_sa*(x);

   (we don't need pointer *left_so* for the list which is currently at the lowest level)

   For every real element y do for which FIRST(y) is true do

   left_sa(*succ*(y)):=*left_sa*(y); *nr_sa*(*succ*(y)):=*nr_sa*(y)

5. Form a new sorted list by pairing real elements and doubling weights of sampling elements. Elements obtained from pairing real elements are considered as real. All the functions (*nr_r*, *nr_sa*, *left_r*, *left_sa*, *succ*, *pred*) can be easily computed from the corresponding functions of the old list.

$MERGE_k(C,V)$ for k=3,6 is implemented as one or two insertion steps (we use concurrent read facility to find the proper place for the inserted element).

$MERGE_5(C,V)$ is implemented as follows:

1. For every element of V decide (basing on *left_so*) to which gap it belongs.

2. For every real element of C decide (basing on *left_sa*) to which basic subsequence it belongs.
3. Merge every basic subsequence with corresponding gap using Valiant's merging algorithm. (Existing information allows us to treat gaps and basic sequences as arrays). Let x be an element from a merged list and let d(x) be the number of elements in a merged list which precede x which are from sequence other than the sequence to which x belonged before merging. This value can be computed as the difference between the current position in the (merged) subsequence and the previous position in the gap or basic subsequence.
4. Compute functions *nr_r*, *nr_sa*, *left_sa* Since we know position of every element in V we may assume that we have an immediate access to every element of V. Denote by V(i) i^{th} element of sequence V.

   For every x∈ V:

   nr_sa(x), *left_sa* -remains unchanged,

   nr_r(x):=d(x)+*nr_r*(x)+*nr_r*(*source*$^{-1}$(*left_so*(x));

   For every x∈ C:

   y:=V(*dist*(*source*(*left_sa*(x)))+d(x));

   - - - *find the closest element from V preceding x in the merged sequence*

   nr_sa(x) :=*nr_sa*(y); *left_sa*(x):=*left_sa*(y);*nr_r*(x):=*nr_r*(x)+*nr_r*(y));
5. For every x: *dist*(x):=*nr_sa*(x)+*nr_r*(x)
6. Computing of *left_r:* Let R be an auxiliary array. For every real element x do R(*nr_r*(x)):=x (assume R(0)=null). If x is a real element then *left_r*(x):=R(*nr_r*(x)-1) otherwise *left_r*(x):=R(*nr_r*(x))

   The remaining functions can be easily computed in O(1) steps.

This finishes the description of the algorithm. We can summarize the main result of this section in the following theorem:

Theorem 18: A tree whose cost differs at most $\frac{1}{n^k}$ by from the cost of an optimal tree can be constructed in O(logn log*n) time using n CREW processors.

5.3. O(k^2logn) time n^2 processor parallel interpretation of GCS with $\frac{1}{n^k}$ construction error

In this interpretation we use an approximate merging algorithm which combines two sequences in a constant time. The idea of the merging algorithm is taken from Example 11. First we describe an algorithm which works in O(klogn) time with n^{6k} processors and then we show a hierarchical data structure which allows an $O(k^2 \log n)$ time $n^2/\log n$ processors implementation.

The idea is to partition main subsequences (i.e. sequences with tails or heads excluded) into n^{6k} subsequences s.t. element x belongs to the subsequence j iff $w(x) - \frac{1}{2^i} \in \left[\frac{j-1}{n^{6k}} \frac{1}{2^i}, \frac{j}{n^{6k}} \frac{1}{2^i}\right)$ where i is the level of the subsequence (we treat heads and tails separately). If an element, say x, belongs to the j^{th} sublist we say that its *subrank* is equal to j (denote subrank(x)=j). In order to merge two sequences of the same level we concatenate corresponding subsequences (cf. Example 11). To allow fast implementations of MERGE and PAIR_ELEMENTS we represent the sequences in the following way:

REPRESENTATION OF SEQUENCES:

For every element, x, we have:

dist(X,x) - position of element x in the sequence X,
succ(X,x) - successor of element x in the sequence X,

If $\mathbf{X}^j$ is a subsequence of a basic sequence $\mathbf{X}$ then

SUCC($\mathbf{X}^j$) - the closest nonempty subsequence following $\mathbf{X}^j$.
FIRST($\mathbf{X}^j$) - the first element of the subsequence $\mathbf{X}^j$.
LAST($\mathbf{X}^j$) - the last element of the subsequence $\mathbf{X}^j$.
(If a subsequence j is empty then FIRST($\mathbf{X}^j$)=LAST($\mathbf{X}^j$)=0.

INITIALIZATION: It is not difficult to construct the above data structure in O(klogn)time with n^{6k} processors.

PAIR_ELEMENTS: First we show how to compute subrank of the parent, say u, of two elements u_1, u_2 in time $|\text{rank}(u_2)-\text{rank}(u_1)|$. We assume that for every element u we know the boundary values $\frac{1}{2^{\text{rank}(u)}}$ and $\frac{1}{2^{\text{rank}(u)-1}}$. If u_1, u_2 have the same rank then

$$\lfloor \frac{subrank(u_1)+subrank(u_2)}{2} \rfloor \leq subrank(u) \leq \lfloor \frac{subrank(u_1)+subrank(u_2)}{2} \rfloor +1$$

We can compute in O(1) time the boundary values of the two possible subranks of element u and in this way determine one of two possible values. So assume that $rank(u_1)$-$rank(u_2)$ $= \Delta r > 0$. In this case we normalize the subrank of the smaller element by dividing it by $2^{\Delta r}$. Note that u has rank equal to $rank(u_2)$ or $rank(u_2)$-1. In the first case we have

$$\lfloor \frac{subrank(u_1)}{2^{\Delta r}} + subrank(u_2)) \rfloor \leq subrank(u) \leq \lfloor \frac{subrank(u_1)}{2^{\Delta r}} + subrank(u_2)) \rfloor +1$$

and in the second case we have:

$$\lfloor \frac{subrank(u_1)}{2^{\Delta r+1}} + \frac{subrank(u_2)}{2} \rfloor \leq subrank(u) \leq \lfloor \frac{subrank(u_1)}{2^{\Delta r+1}} + \frac{subrank(u_2)}{2} \rfloor +1$$

so again we have to determine one of two possible values. To be able to do this we shall compute the boundary values of the two subranks possible for u. We can compute them from the boundary values of subranks of elements u_1, u_2 using a method similar to the one described above.

Procedure $PAIR\_ELEMENTS_2$ has only two elements to pair. We simply create a common parent for both of them and compute the subrank of the new element. Procedure $PAIR\_ELEMENTS_5$ can be implemented as follows:

1. Create a common father u for every element u_1 whose value $dist(U,u_1)$ is odd and the element $u_1 = succ(U,u_1)$.
2. For each element v, $dist(C,v) := \lceil dist(U,left(v))/2 \rceil$.
3. For every element v from the resulting list C compute its subrank (i.e. divide C into sublists).
4. To compute FIRST and LAST decide for every element of list C if it is the first and/or the last element of this subrank by comparing the subrank of the given element with subranks of its neighbors.
5. To compute PRED and SUCC:

   If $FIRST(C^j) \neq 0$

   then $PRED(C^j) := IN(parent(prec(U,left(FIRST(C^j)))))$

   else $PRED(C^j) := IN(parent(LAST(PRED(U^j))))$

   where IN(x) is a function which returns the pointer to the subsequence containing element x

Function SUCC can be computed similarly.

So the time to implement the above procedure with n^{6k} processors is $O(\Delta r_i+1)$ where Δr_i is equal to the difference of ranks of the first two elements on list U_{2i}. But $\sum_{j=1}^{I} \Delta r_i = O(K(n))$ so the total time spent on pairing step is O(K(n)).

$MERGE_k(C,V)$: For k=3,6 this procedure is implemented as one or two insertion steps. If the inserted element has rank greater or smaller than the level of the sequence then it forms tail or head of the sequence and is treated separately. For k=5 procedure $MERGE_k(C,V)$ can be implemented as follows.

1. Obtain the resulting list U by putting, for every j, elements of subrank j from list C before elements of subrank j from list V.
2. For each element v of subrank j in list U compute $dist(U_i,v)$ as follows:

   if v is an element from list C then:

   dist(U,v) = $dist$(C,v) + $dist$(V, LAST(PREC(V^j))

   else if FIRST(C^j)≠0

   then $dist$(U,v) = $dist$(V,v) + $dist$(C,LAST(C^j))

   else $dist$(U,v)= $dist$(V,v)+$dist$(C,LAST(PREC(C^j))).
3. Compute functions FIRST, LAST, PRED, SUCC for list U:

   if FIRST(C^j)≠0 then FIRST(U^j):=FIRST(C^j)

   else FIRST(U^j):=FIRST(V,j);

   PRED((U^j):=max(PRED(C^j),PRED((V^j));

   if LAST(C^j)=0 then LAST(U^j):=LAST(C^j)

   else LAST(U^j):=LAST(V^j);

   SUCC((U^j):=min(SUCC((C^j),SUCC((V^j)).

It is easy to see that procedure MERGE can be implemented in O(1) time with n^{6k} processors. This leads to the following lemma:

Lemma 19: A tree whose cost differs at most by $\frac{1}{n^{k-3}}$ from the cost of an optimal tree can be constructed in O(klogn) time using n^{6k} CREW processors.

Proof: The processor and time bounds follow directly from the description of the algorithm. The truncation error is $\frac{1}{n^{k-2}}$. We will show that the construction error is bounded by $\frac{8}{n^k}$. Let $d_i=\Delta_{2i}+\Delta_{2i-1}$. We prove that $d_i \le \frac{4}{2^i n^k}$ which, by Theorem 5, implies the result. More precisely we show, by induction, that $\Delta_{2i} \le \frac{2}{2^i n^k} \frac{2^{5(I-i)}}{2^{5k\log n}}$ and $\Delta_{2i-1} \le \frac{2}{2^i n^k} \frac{2^{5(I-i)+2}}{2^{5k\log n}}$.

For every j the sequence U^A_j constructed by the algorithm is an approximation of the corresponding sequence U^H_j constructed by the Huffman tree algorithm. It suffices to show that U^A_{2i} is a $\frac{1}{2^i n^k} \frac{2^{5(I-i)}}{2^{5k\log n}}$- approximation of the sequence U^H_{2i} and U^A_{2i-1} is a $\frac{1}{2^i n^k} \frac{2^{5(I-i)+2}}{2^{5k\log n}}$- approximation of the sequence U^H_{2i-1}. For U^H_{2I} this fact is obvious. For other values of i note that, by Lemma 13, each application of MERGE at most doubles the approximation error. Also each application of PAIR_ELEMENTS at most doubles the approximation error. Since in GCS between creation of a sequence U_{2i} and U_{2i-1} we have two calls of MERGE and PAIR_ELEMENTS and between creation of a sequence U_{2i-1} and U_{2i-2} we have three calls of MERGE and PAIR_ELEMENTS the result follows. ■

Note that in the above algorithm the high number of processors follows from the fact that we use one processor for each subsequence. But in any sequence there are at most n nonempty subsequences. To avoid this inefficient utilization of both space and processors, we divide a sequence into subsequences in the following recursive way: A sequence is divided into n subsequences, then every nonempty subsequence is divided into n subsubsequences and so on (6k times). The partition of sequences is reflected by a hierarchical data structure. As we will see the number of subsequences at every level of the hierarchy is bounded by n^2. To merge two sequences we concatenate corresponding subsequences. (Note that heads and tails of sequences have to be treated separately). More formally:

Let X be a main subsequence of level i and let t=6k. Then

1. X is divided into n subsequences $X^1,...,X^n$ (some of them may be empty) according to the weights such that $x \in X^j$ iff $w(x) - \frac{1}{2^i} \in [\frac{j-1}{n}\frac{1}{2^i}, \frac{j}{n}\frac{1}{2^i})$.

2. Each nonempty sequence $X^{i_1 i_2 ... i_k}$, where, $k<t$ is divided into n subsequences $X_i^{i_1 i_2 ... i_k 1}$ ...,$X_i^{i_1 i_2 ... i_k n}$ such that $x \in X_i^{i_1 i_2 ... i_k j}$ iff $w(x) - \frac{1}{2^i}(1 + \frac{i_1-1}{n} + \frac{i_2-1}{n^2} + ... + \frac{j-1}{n^k}) \in [\frac{j-1}{n^{k+1}}\frac{1}{2^i}, \frac{j}{n^{k+1}}\frac{1}{2^i})$.

We say that X^α is a *k^{th} order subsequence* iff α is a sequence of k indices. If $x \in X^\alpha$ and α is a sequence of k indices such that $\alpha=\beta j$ then we say that $subrank_k(x)=j$. We maintain the subsequences of each order in lexicographical order of their upper index. The sequences are represented by the following data structure:

REPRESENTATION OF SEQUENCES:

For each element, v , from the sequence X we have

dist(X,v) - position of v in the sequence X

Let $\mathbf{X}$ be the main subsequence of the sequence X.

For every subsequence $\mathbf{X}^\alpha$ we have:

FIRST($\mathbf{X}^\alpha$) - a pointer to the first element of the subsequence.

LAST($\mathbf{X}^\alpha$) - a pointer to the last element of the subsequence.

(If subsequence is empty then FIRST($\mathbf{X}^\alpha$)=LAST($\mathbf{X}^\alpha$)=0).

For a each subsequence $\mathbf{X}^\alpha$ of order t we have:

SUCC($\mathbf{X}^\alpha$) - a pointer to the closest nonempty subsequence of order t following $\mathbf{X}^\alpha$.

PRED($\mathbf{X}^\alpha$) - a pointer to the closest nonempty subsequence of order t preceding $\mathbf{X}^\alpha$.

The subsequences which differ only by last index are kept in an array (called block) ordered according to the last index.

For each subsequence of order smaller than t we have:

DOWN($\mathbf{X}^\alpha$) - a pointer to the block of subsequences into which $\mathbf{X}^\alpha$ is divided

For each subsequence $\mathbf{X}^\alpha$ of order greater than zero (where zero is the order of whole sequence) we have:

UP($\mathbb{X}^{\alpha}$) - a pointer to the subsequence of one order lower than order of $\mathbb{X}^{\alpha}$ which contains given subsequence.

For every element we know its t subranks.

INITIALIZATION: Assign n processors to every element. We sort the input sequence using Cole's parallel merge sort. For every element compute all its subranks. This can be done in O(klogn) time using n processors by k applications of binary search (performed for every element in parallel). Use the first subrank to divide sequences into first order subsequences. To compute functions FIRST and LAST it suffices to compare the subrank of every element with the subranks of its neighbors. Compute (using prefix sum computation) the number of nonempty subsequences preceding a given subsequence. Divide every nonempty first order subsequence into second order subsequences according to the value subrank_2 and construct pointer DOWN. So we obtain, for every nonempty subsequence, n second order subsequences (some of them possibly empty). Assign one processor for every n elements of the second order subsequence (say one of n processors associated with the first element of the subsequence of first order). For every second level sequence construct pointer UP (we can do it in O(1) time time with n^2 processors). Since the number of first order subsequences preceding a given subsequence is known and every first order subsequence is divided into exactly n second order subsequences, we can treat second order subsequences as consecutive elements of some array. (We can compute the position of every second level subsequence in such an array in O(1) time) So we can use a prefix sum computation to compute, for every second order subsequence, the number of nonempty subsequences preceding it. Similarly we compute subsequences of next orders, corresponding functions FIRST, LAST, UP, DOWN, and the number of nonempty subsequences of given order preceding given subsequence. From the last information we can compute PRED and SUCC for subsequences of order t in the following way: Use an array, say A, and assign to A(i) i^{th} nonempty subsequence. For a subsequence X with index j in A do PRED(X):=A(j-1); SUCC(X):=A(j+1).

The initialization step can be implemented in O(klogn) time with n^2 CREW processors.

PAIR_ELEMENTS: To show the implementation of this procedure we first show how to compute in $O(t\,|\text{rank}(u_1)-\text{rank}(u_2)|)$ for any two neighboring elements u_1, u_2 the subranks of

their parent, say u. But it is easy to compute in O(t) time the value subrank(u) from the sequence $subrank_1(u)$, $subrank_2(u)$,....,$subrank_t(u)$ and the opposite. So we can use the method presented for n^{6k} algorithm.

PAIR_ELEMENTS$_2$(U): In this case we have only two elements to pair. We simply create a common parent for both of them and compute all subranks of the new element.

PAIR_ELEMENTS$_5$(U):

1. Create a common father u for every element u_1 whose value $dist(U,u_1)$ is odd and the element $u_1=succ(U,u_1)$. Let C be the resulting list.
2. For each element v in list C: $dist(C,v):= \lceil dist(U,left(v))/2 \rceil$.
3. Let u_1 and u_2 be a pair of elements which obtain a common father, say u. To construct the data structure of the new sequence C modify the data structure of sequence U by removing elements u_1 and u_2 and inserting element u. To do this first compute all subranks of every newly created element u.
4. Find, using pointers UP, the subsequence of the highest order, say s, to which both of u_1, u_2 belong. If u belongs to a subsequence of order k+1 which was previously empty then build subsequences on orders s+2, ...,t (together with pointers UP, DOWN). Since the sequence is a one-element sequence we can do it, for every new element, in O(s) time with n processors (using information about subranks).
6. Compute function FIRST and LAST for every level and every subsequence using a method similar as in the initialization step.
7. For every subsequence check whether it is an empty subsequence. If yes and if the higher level subsequence containing the given subsequence is also empty then remove this subsequence from the data structure.
8. Compute PRED and SUCC. Let $IN_t(u)$ be the reference to the subsequence order t containing u. For any order t subsequence $C^{\alpha i}$ of the sequence C do

   If $FIRST(C^{\alpha i})\neq 0$ then $PRED(C^{\alpha i}):=IN_t(parent(prec(U,left(FIRST(C^{\alpha i}))))$

   else if the subsequence $U^{\alpha i}$ was represented in the data structure for then

   then $PRED(C^{\alpha i}):= IN_t(parent(LAST(PRED(U^{\alpha i}))))$

   else there is exactly one element, x, of in the subsequence C^{α}. Let $x\in C^{\alpha j}$.

   if i>j then $PRED(C^{\alpha i})=IN_t(x)$ else $PRED(C^{\alpha i})=IN_t(pred(C,x))$.

Function SUCC can be computed similarly.

$MERGE_k(C,V)$ (for k=3,6): This procedure is implemented as one or two insertion steps. If the inserted element has rank equal to the level of sequence to which it is inserted then it is added to the hierarchical data structure. Since the ranks of the element are known it can be done in O(k) time. If the rank of the inserted element is smaller or greater than the level of the sequence then the inserted element is added to the head or tail of the sequence. (Note that we never have more than two elements in a head nor more than one element in a tail).

$MERGE_5(C,V)$: Note that at this step neither of merged sequences have nonempty tail or head. Let U be the resulting sequence. We merge the sequences C and V in a top-down fashion:

1. Using n processors combine (for each j) V^j with C^j. If one the subsequences is empty then U^j is represented by the nonempty one. If both V^j and C^j are nonempty then we call U^j an active subsequence of level order 1. Note that we may have at most n active subsequences.
2. *current_order*:=1;
3. While *current_order*<t do
   3.1. For each active subsequence of current order combine recursively (using n processors) subsequences of next order (use pointer DOWN to find the proper block of subsequences). Let V^α and C^α be the merged subsequences. If *current_order*+1 = t then put elements of V^α before C^α . If *current_order*+1 <t and if one of the subsequences is empty then U^α is equal to the nonempty sequence otherwise U^α is an active subsequence of order *current_order+1*.
   3.2. *current_order*:=*current_order*+1;
4. Compute FIRST, LAST, PRED, SUCC for the lowest level as in step 4 of $PAIR\_ELEMENTS_5$.

Theorem 20: A tree whose cost differs at most by $\frac{1}{n^k}$ from the cost of an optimal tree can be constructed in $O(k^2 \log n)$ time using n^2 CREW processors.

Proof: The algorithm is a modification of the n^{6k} processor algorithm in which we use n^2 processors and the time of each call of MERGE or PAIR_ELEMENT is multiplied by the depth of the hierarchical data structure. So the time complexity is $O(k^2 \log n)$. The bound for the construction error follows from Lemma 19. ■

6. Sequential interpretations of the GCS

A natural sequential interpretation of the general construction scheme is to obtain an ε-sorted sequence using an integer sorting algorithm and implement MERGE as standard merging procedure (recall Example 11). Since the cost of merging using the standard merging procedure is proportional to the sum of lengths of the merged sequences, the total time which is spent on merging is O(n). To obtain a linear time implementation of the general construction scheme we must be able to compute ranks in linear time. Assume that the ranks are bounded by cf(n) and that we can sort in linear time integers in range $[0,2^{cf(n)}]$. Then we can compute ranks in $O(n\log\frac{cf(n)}{n})$ time by the following algorithm:

1. Sort the sequence according to $\lceil w(v_i)2^{cf(n)} \rceil$,
2. Merge the resulting sequence with the following sequence of elements 1, $2^{1\lceil \frac{f(n)}{n} \rceil c}$, $2^{2\lceil \frac{f(n)}{n} \rceil c}$, ..., $2^{n\lceil \frac{f(n)}{n} \rceil c}$ called *boundary elements*. Let X_1 be the resulting sequence.
3. i:=1;
4. while $i \le \log(\lceil \frac{f(n)}{n} \rceil c)$:
   4.1. Let X'_{i+1} be the sequence obtained from X_i by removing those boundary elements which have immediately before it and immediately after them boundary elements;
   4.2. Each boundary element belonging to X'_{i+1} which has immediately before it a real element defines a new boundary element equal to the geometric average of its value and the value of the closest boundary element which occurs in X'_{i+1} before the given boundary element. Denote by B_{i+1} the sorted sequence of such defined new boundary elements;
   4.3. Obtain X_{i+1} by merging X'_{i+1} and B_{i+1};
   4.4. i:=i+1;
5. Now every non-boundary element is between two boundary elements which are consecutive powers of two. Elements which are between 2^i and 2^{i+1} have rank cf(n) - i.

If we assume f(n)=logn and $c=11\cdot 2^k$ we can use integer sorting algorithm of Kirkpatrick and Reisch [KR84] which sorts integers in range $[0,n^c]$ in O(n(1+logc)) time. As the result we obtain $\frac{1}{n^c}$-sorted sequence. If we apply GCS with $K(n)=2^k\log n$ then we

can compute ranks of heavy elements in O(kn) time with the help of the algorithm described above. This interpretation leads to the following theorem:

Theorem 21: A tree T such that $\Delta T \leq \frac{1}{n^{2^k}}$ can be constructed in O(kn) time.

Proof: It is obvious that the algorithm presented above runs in O(kn) time. The truncation error is at most $\frac{1}{n^{2^k-2}}$. It suffices to show that the construction error in this algorithm is at most $\frac{8}{n^{2^k}}$. Using the same technique as in the proof of Lemma 19 we can show that $\Delta_{2i} \leq \frac{2}{2^i n^{2^k}} \frac{2^{5(I-i)}}{2^{10 \cdot 2^k \log n}}$ and $\Delta_{2i-1} \leq \frac{2}{2^i n^{2^k}} \frac{2^{5(I-i)+2}}{2^{10 \cdot 2^k \log n}}$ which implies the result. ■

If we assume as a computation model a RAM with unbounded register capacity then we can sort n integers in range $[0,2^{cn})$ in O(n(1+logc)) time ([KR84]). If we assume $c=7 \cdot 2^k$, $f(n)=n$, and $K(n)=2^k n$ then we can use the algorithm described at the beginning of this section to obtain $\frac{1}{2^{cn}}$-sorted sequence of heavy elements and to compute ranks of heavy elements in O(kn) time. This construction leads to the following theorem:

Theorem 22: A tree T such that $\Delta T \leq \frac{1}{2^{n2^k}}$ can be constructed in O(kn) time on a RAM is unbounded register capacity.

Proof: Using exactly same technique as in the proof of Theorem 21. ■

7. Conclusions

We can summarize the results of this paper in the following corollary:

Corollary 23: One can construct a binary tree T, such that ΔT is bounded by the sum of construction error and truncation error, in t(n) time using p(n) CREW processors where p(n), t(n), truncation error, and construction error are given in the following table:

t(n)	p(n)	construction error	truncation error
O(klogn)	$n\frac{loglogn}{logn}$	0.172	$\frac{1}{n^{k-2}}$
O(klog*n logn)	n	0	$\frac{1}{n^{k-2}}$
$O(k^2logn)$	$n^2/logn$	$\frac{8}{n^k}$	$\frac{1}{n^{k-2}}$
O(kn)	1	$\frac{8}{n^{2^k}}$	$\frac{1}{n^{2^k-2}}$
O(kn)[1]	1	$\frac{8}{2^{n2^k}}$	$\frac{1}{2^{n2^k-2logn}}$

where k is an integer constant chosen by the algorithm.

We can observe that neither of the proposed parallel algorithms achieves the optimal speedup. The sequential algorithms presented in the last section have linear running time and produce a tree with an error smaller than the error of any parallel algorithm presented in the paper (even when restricted to a realistic computation model). However we have an O(klognlog*n) time and n processor algorithm which realizes an almost optimal speedup over the Huffman algorithm and produces a tree with a very small error. In this case we have only one type of error, namely truncation error. So if the input sequence does not contain elements of weight less than $\frac{1}{n^k}$ then the algorithm produces an optimal tree.

It would be interesting to see whether there exists an efficient parallel algorithm for construction of a binary tree with $\Delta T \leq \frac{1}{n^k}$ such that processor time product is linear in n. Also the question whether there exists an efficient parallel algorithm to construct an optimal binary tree such that processor time product is O(nlogn) remains open.

One should be aware of another source of error which we have not addressed in this paper, namely the error resulting from representation of real numbers on a computer. Our algorithms use only comparison, addition, division by 2,mod, and $\lceil\ \rceil$, with the exception of the third parallel algorithm which uses also division by n. One can show that Huffman's

[1]On RAM with unbounded register capacities

algorithm is numerically stable (i.e. a relative error of δ in the input data causes a relative error of at most $O(n)\delta$ in the cost of the constructed tree) Similarly our O(klognlog*n) algorithm is numerically stable. It is also worth noting that if the input sequence is given as a sequence of integers representing relative frequencies rather than probabilities then we can reformulate our algorithms (with the exception of the second integer sorting), so that they will perform only integer operations using words of size of the order of the size of maximal input element.

References

[AKLMT89] M.J.Atallah, S.R.Kosaraju, L.L.Larmore, G.L.Miller, S-H.Teng, "Constructing trees in parallel", Proc 1st ACM Symposium on Parallel Algorithms and Architectures, 1989, 421-431.

[BH85] A.Borodin, J.E.Hopcroft, "Routing, merging and sorting on parallel models of computation", J. Comp. Sys. Sci., Vol. 30, 1985, 130-145.

[C86] R.Cole, "Parallel merge sort", Proc 27th Annual IEEE Symp. on Foundation of Computer Science, 1986, 511-516.

[H73] T.C.Hu, "A new proof of the T-C algorithm", SIAM J.Appl. Math., Vol. 25, No 1, July 1973, 83-94.

[Huff52] D.A.Huffman, "A method for the construction of minimum redundancy codes", Proc. IRE, 40, 1952, 1098-1101.

[KR84] D.G.Kirkpatrick, S.Reisch, "Upper bound for sorting integers on random access machines", Theoretical Computer Science 28 (1984) 236-276.

[MRK88] G.L.Miller, V.Ramachandran, E.Kaltofen, "Efficient parallel evaluation of straight line code and arithmetic circuits", SIAM J. Comput. 1988, to appear.

[M85] K.Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer Verlag, (1984).

[T87] S-H.Teng, "The construction of Huffman equivalent prefix code is in NC", ACM SIGACT, 18(4) 1987, 54-61.

[V75] L.Valiant, "Parallelism in comparison problems", SIAM J.Comput., vol. 4, 1975, 348-355.