PARALLEL CONSTRUCTION OF TREES WITH OPTIMAL WEIGHTED PATH LENGTH

Lawrence L. Larmore and Teresa M.Przytycka †

Department of Computer Science,
University of California, Riverside, CA 92521

Abstract

This paper deals with the problem of parallel construction of trees with optimal weighted path length. We study both the unordered case, known as the *Huffman coding* problem and the ordered case known as the *optimal alphabetic binary tree* problem. The methods used in both cases are different. We reduce the Huffman coding problem to the *Concave Least Weight Subsequence* and give a parallel algorithm that solves the latter problem in $O(\sqrt{n}\log n)$ time with n processors on a CREW PRAM. This leads to the first sublinear time $o(n^2)$-total work parallel algorithm for the Huffman coding problem. The alphabetic binary tree problem is a special case of the *Optimum Binary Search Tree* problem and can be solved in $O(\log^2 n)$ time with n^4 processors using the dynamic programming technique. We show that an optimal height restricted alphabetic tree can be constructed in $O(L\log n)$ time on a CREW PRAM using only linearly many processors, where L is an upper bound on the height of the tree.

This implies that an alphabetic tree whose cost differs by at most $1/n^k$ from the cost of the optimal tree can be constructed in $O(k\log^2 n)$ time using linear number of processors. To achieve this result we use a parallel version of the *package merge* technique.

1 INTRODUCTION

The *level* of a node in a tree is its distance from the root. The problem of constructing a *Huffman tree* is, given a sequence of n real numbers, $x_1, x_2, \ldots, x_n$, construct a binary tree with n leaves such that the leaves of the tree are in one-to-one correspondence with elements of the sequence, and the following cost function, c, is minimized:

$$c(T) = \sum_{i=1}^{n} x_i \ell_i \tag{1.1}$$

where ℓ_i denotes the level of the leaf corresponding to the number x_i, not necessarily the i^{th} leaf. The value x_j associated with a leaf v is called the *weight* of v.

In the *optimal alphabetic binary tree* problem we additionally assume that the i^{th} leaf of the tree is assigned weight x_i. Without loss of generality, $\sum_{i=1}^{n} x_i = 1$ and $x_i \neq x_j$ for $i \neq j$. The *height restricted optimal alphabetic tree* problem is given an integer L, where $\log n \leq L \leq n-1$, construct an alphabetic tree of height at most L that minimizes the cost function c.

† and Instytut Informatyki, Uniwersytet Warszawski.

Both the Huffman tree problem and the optimal alphabetic tree problem can be solved in $O(n \log n)$ sequential time [8,13]. Yet, despite much effort, neither of the problems has a good parallel algorithm. Currently the best NC algorithm for the Huffman coding problem takes $O(\log^2 n)$ time with roughly n^2 CREW PRAM processors [4]. A close approximation for the solution to the Huffman coding problem can be computed in $O(\log n \log^* n)$ time using linear number of CREW processors [14,22]. In this paper we present the first sublinear time, $o(n^2)$-work parallel algorithm for the Huffman coding problem.

At the heart of our algorithm is the reduction of the Huffman coding problem to the *Concave Least Weight Subsequence* (CLWS) problem. This reduction leads to a brand new linear time (if the input sequence of weights is sorted) sequential algorithm for the Huffman coding problem.

The CLWS problem has a long list of applications, including paragraph breaking. A good parallel algorithm for this problem is thus interesting in its own right. Hirschberg and Larmore [10] define the *Least Weight Subsequence* (LWS) problem as follows: Given an integer n, and a real-valued *weight* function $w(i,j)$ defined for integers $0 \leq i < j \leq n$, find a sequence of integers $0 = \alpha_0 < \alpha_1 < \ldots < \alpha_{k-1} < \alpha_k = n$ such that $\sum_{i=0}^{k-1} w(\alpha_i, \alpha_{i+1})$ is minimized. The *Single Source* LWS problem is to find such a minimal sequence $0 = \alpha_0 < \alpha_1 < \ldots < \alpha_{k-1} < \alpha_k = m$ for *all* $m \leq n$. The LWS problem can be solved in $O(n^2)$ sequential time [10]. This complexity can be reduced by imposing certain restrictions on the weight function. The weight function is *concave* if for all $0 \leq i_0 \leq i_1 < j_0 \leq j_1 \leq n$,

(1.2) $w(i_0, j_0) + w(i_1, j_1) \leq w(i_0, j_1) + w(i_1, j_0)$.

The inequality (1.2) is also called the *quadrangle inequality* [25]. The LWS problem defined by a concave weight function is called the Concave Least Weight Subsequence (CLWS) problem. Hirschberg and Larmore [10] showed that the CLWS problem can be solved in $O(n \log n)$ time. Subsequently, Galil and Giancarlo [7] showed the $O(n \log n)$ complexity bound for the *convex* least weight subsequence problem (i.e. when the weight function satisfies the reverse of the inequality (1.2)). Later, Wilber [24] proposed an $O(n)$ algorithm for the CLWS subsequence problem. The best known sequential algorithm for the convex case was proposed by Klawe and Kleitman [16]. Their algorithm requires $O(n\alpha(n))$ time.

All of the above algorithms for the LWS and CLWS problems actually solve the single source versions. Henceforth, when we refer to any variant of the LWS problem, we shall mean the single source version.

In the parallel setting, the CLWS problem seems to be more difficult than the corresponding convex problem. Lam and Chan [6] presented an $O(\log^2 n \log\log n)$-time, $n/\log\log n$ - processor CREW PRAM algorithm to solve the convex problem. On the other hand, the best current NC algorithm for the CLWS problem can be obtained using the concave matrix multiplication techniques [2,3,4] and requires $O(\log^2 n)$ time with $n^2/\log n$ processors. In this paper we present an $O(\sqrt{n} \log n)$-time n-processors CREW PRAM algorithm to solve the CLWS problem.

The optimal alphabetic tree problem is a special case of the Optimum Binary Search Tree (OBST) problem. Thus it can be solved using parallel dynamic programming techniques in $O(\log^2 n)$ time with roughly n^4 processors, using the concave matrix multiplication algorithm [4]. An $O(\log^2 n)$-time, n^2-processor algorithm that uses dynamic programming techniques to compute an approximately optimal binary search tree was given in [4]. This algorithm can be used to obtain an approximately optimal alphabetic tree. On the other hand, the best currently known sequential algorithm to construct an optimal alphabetic binary tree does not use dynamic programming. Thus one can ask whether, in the parallel setting of the problem, one can also find a non-dynamic programming algorithm whose complexity is less than the complexity of parallel OBST algorithms that are currently known. In this paper we give a partial answer to

the above question. We present an $O(L \log n)$-time n-processor CREW PRAM algorithm which constructs an optimal alphabetic tree subject to height restriction L. The best known sequential algorithm for this problem [17] requires $O(nL \log n)$ time. As a consequence, we obtain an $O(k \log^2 n)$-time, n-processor algorithm that constructs an almost optimal alphabetic tree (more precisely a tree whose cost differs by at most $1/n^k$ from the cost of an optimal tree). Furthermore if the input sequence of weights does not contain two consecutive elements of weight less than $1/n^k$ the algorithm produces an optimal tree. We base our algorithm on the *package-merge* technique introduced in [17] (see also [18,19,20]).

The paper is organized as follows. In the next section, we present a parallel algorithm for the CLWS problem. In the third section, we show the reduction of the Huffman coding problem to the CLWS problem. In the fourth section, we describe briefly the package merge technique. In the fifth section, we outline a parallel implementation of the package merge algorithm.

2 A PARALLEL SINGLE SOURCE CLWS ALGORITHM

Consider an instance of the least weight subsequence problem over $[0, n]$ defined by a weight function w. Let $lws(i, j)$ denote the solution to the least weight subsequence problem on $[i, j]$ defined by the same function w. (This "solution" includes backpointers as well as the values of the minimum weight paths.) Let $g(i, j)$ be the weight of the least weight subsequence on $[i, j]$. We will compute $lws(0, j)$ for all $0 \leq j \leq n$ under the assumption that the weight function satisfies the quadrangle inequality (1.2).Let $f(j) = g(0, j)$ Define

$$pred(j) = \begin{cases} undefined & \text{for } j = 0 \\ min\{i | i < j, f(j) = f(i) + w(i, j)\} & \text{for } 0 < j \leq n \end{cases}$$

Thus $f(j)$ is equal to the weight of the least weight subsequence for the interval $[0, j]$ and $pred(j)$ is equal to the index of the last but one element of such a sequence (in the case of ties we choose the leftmost). Let $F(j) = (f(j), pred(j))$. To solve the LWS problem it suffices to compute the function $F(j)$ for all $j \in [0, n]$.

Given an interval $I = [i_1, i_2] \subset [0, n]$ we can also consider a restricted version of the LWS problem in which we require that for all $j > i_2$, $pred(j) \in I$ (i.e. for any $j > i_2$ the last but one element of the solution to $lws(0, j)$ belongs to I). To solve this problem it suffices to compute, for all $0 < j \leq n$, functions $f^I(j)$ and $pred^I(j)$ where

$$f^I(j) = \begin{cases} f(j) & \text{for } j \leq i_2 \\ \min_{i \in I}(f(i) + w(i, j)) & \text{for } i_2 < j \leq n \end{cases}$$

$$pred^I(j) = \begin{cases} pred(j) & \text{for } j \leq i_2 \\ min\{i | i \in I, i < j, f^I(j) = f(i) + w(i, j)\} & \text{for } i_2 < j \leq n \end{cases}$$

Let $F^I = (f^I, pred^I)$. For any interval $I = [i_1, i_2]$, where $i_2 = i_1 + k$ the function $pred^I$ partitions the interval $[i_2 + 1, n]$ into $k + 1$ subsets $D_0, D_1, \ldots, D_k$, such that $j \in D_t$ if and only if $pred^I(j) = i_1 + t$. Some of the sets D_i may be empty. We say that element i_t *dominates* elements in D_t. This partition of the interval $[i_2 + 1, n]$ *dominance partition yielded by interval* I. If the weight function is concave then the dominance partition satisfies the properties stated by the following lemma:

LEMMA 2.1. *If the weight function w satisfies the quadrangle inequality (1.2) then, for any interval $I = [i_1, i_2] \subset [0, n]$, all nonempty subsets of the dominance partition yielded by I are intervals. Furthermore, if $D_i = [a, b]$ is the set of elements dominated by i and $D_j = [c, d]$ is the set of elements dominated by j, and $i < j$ then $b < c$.*

Assume that $n = m^2$ for some integer m. Let $f^i = f^{[o,im]}$ and $pred^i = pred^{[o,im]}$. The basic idea of our algorithm is to iteratively compute $F^0, F^1, \ldots, F^m = F$

where $F^i = (f^i, pred^i)$. For $0 < i \leq m$, define $I_i = [(i-1)m, im]$. The algorithm can be described as follows:

ALGORITHM 1: Concave Least Weight Subsequence

1. (*preprocessing*) **for all** i, j such that $i < j$ and $j - i \leq m$ **do** compute $lws(i, j)$;
2. (*initialize*) **for all** $0 \leq j \leq n$ **do** $f^0(j) = 0, pred^0(j) = 0$ **od** ; $D_0 = [1, n]$;
3. **for** $i := 1$ **to** m **do** (*iteratively compute* F^i)
   - *3.1* **for all** $k \in I_i$ **do** *using the results of Step 1 and* F^{i-1} *compute* $lws(0, k)$
   - *3.2 compute* F^{I_i};
   - *3.3 from* F^{I_i} *and* F^{i-1} *compute* F^i.

LEMMA 2.2. *The algorithm Concave Least Weight Subsequence can be implemented in* $O(\sqrt{n} \log n)$ *time with* n *CREW PRAM processors.*

Proof: To prove the lemma we present an implementation of the consecutive steps of the algorithm and discuss their complexity. Some of the implementation details we leave to the reader.

STEP 1. Assign one processor to each element of the sequence. Compute $lws(i, j)$ for all $0 \leq i < n - m$ and $j \leq i + m$ using a linear-time algorithm for one instance, of size m, of the single source CLWS problem for each i. This step requires $O(m)$ time. $\diamond$

STEP 3.1. Let $pred_{(i,j)}$ be the *pred* function for the least weight subsequence on the interval $[i, j]$. Then for all $k \in I_i$, we can compute $f(k)$ and $pred(k)$ using the following algorithm:

for all r in the solution of some $lws(j, s)$ where $j, s \in I_i$ **do** $pred(r) = pred_{(j,s)}(r)$ **od** ;

for all $k \in I_i$ **do**

$f(k) = \min_{j \in I_i, j \leq k}(f^{i-1}(j) + g(j, k))$;

$t := min\{j \mid j \in I_i, j \leq k, f^{i-1}(j) + g(j, k) = f(k)\}$;

$pred(t) = pred^{i-1}(t)$.

To implement this algorithm we assign one processor for each pair $j, k \in I_i, j < k$ and compute, for each such j, k, the value $f^{i-1}(j) + g(j, k)$. This requires $O(1)$ time using n processors (or in $O(\log n)$ time using $n/\log n$ processors). Then we compute the corresponding minima in $O(\log n)$ time using $n/\log n$ processors. $\diamond$

STEP 3.2. We implement this step by computing the dominance partition yielded by the interval I_i. To do this, for each $j \in I_i$, we compute $left(j)$ and $right(j)$ equal, respectively, to the left and to the right boundary of D_j ($left(j) > right(j)$ signals empty D_j). Assume that $j < k$. Let $boundary(j, k)$ be the leftmost element in the interval $[im + 1, n]$ for which j is a better candidate for $pred^{I_i}$ than k. We compute the dominance partition using the following algorithm:

For $j < k$, $boundary(j, k) = \min\{t | t \in [im + 1, n]$, and $f(j) + w(j, t) \leq f(k) + w(k, t)\}$;

for all $j \in I_i$ **do**

$right(j) = \min_{k \in I_i, k > j} boundary(j, k)$;

$left(j) = \max_{k \in I_i, k < j} boundary(k, j)$;

if $right(j) < left(j)$ **then** $D_j = \emptyset$.

As in the step 3.1 assign one processor for each pair $j, k \in I_i, j < k$. Then, by Lemma 2.1, for each such pair j, k, where $j < k$, we compute $boundary(j, k)$ using binary search. Since we have one processor per boundary value, for each j, functions $left(j)$ and $right(j)$ can be computed in $O(\log n)$ time. time using n processors. $\diamond$

STEP 3.3. To compute F^i we simply have to check, for every element $j \in [im + 1, n]$, which of $f^{i-1}(j)$ and $f^{I_i}(j)$ is smaller, using $O(1)$ time with n processors. $\diamond$

$\square$

3 REDUCTION OF THE HUFFMAN TREE PROBLEM TO CLWS

A binary tree where each internal node has two children can be uniquely described by listing the levels of its leaves, in left to right order. A sequence of n integers $\ell_1, \ell_2, \ldots, \ell_n$ for which there exists a tree with n leaves whose levels when read from the left to the right are $\ell_1, \ell_2, \ldots, \ell_n$ is called a *leaf pattern.* A tree is said to be *left-justified* [4] if its leaf pattern is non-increasing. It can be shown (see for example Hu and Tan [11]) that a Huffman tree can be realized as a tree whose whose leaf pattern is sorted. Thus, by the definition of the cost function, c, we have the following lemma:

LEMMA 3.1. ([4]) *Given an arbitrary input sequence X, a Huffman tree for X can be realized by a left-justified binary tree whose leaf weights form a non-decreasing sequence.*

By the above lemma, sorting reduces our initial problem to that of constructing a left-justified binary tree which minimizes the cost function c. We show that the last problem can be reduced to the CLWS problem. A left-justified tree T of height k can be uniquely described by listing the numbers of internal nodes on each level of the tree. Thus T can be described by a strictly monotone increasing sequence of integers $\overline{\alpha_T} = \alpha_0, \alpha_1, \ldots, \alpha_{k-1}$, $\alpha_0 = 0$, $\alpha_k = n - 1$, where, for any $j > 0$, $\alpha_j - \alpha_{j-1}$ is equal to the number of internal nodes on level $k - j$. We call $\overline{\alpha_T}$ the *level sequence* of the tree T. However, not every strictly monotone sequence starting at 0 and ending at $n - 1$ is the level sequence for a left-justified tree.

The main result of this section is stated in the following theorem:

THEOREM 3.1. *Let $s_1, s_2, \ldots, s_n$ be the sequence of prefix sums for a nondecreasing sequence $X = x_1, x_2, \ldots, x_n$ i.e. $s_i = \sum_{j=1}^{i} x_j$ and let $w(i,j)$ be the weight function defined as follows:*

$$w(i,j) = \begin{cases} s_{2j-i} & \text{if } 2j - i \leq n \\ \infty & \text{otherwise} \end{cases}$$

then the LWS problem defined by w is concave and the solution to this problem is equal to the level sequence of the left justified Huffman tree for X.

Proof: (sketch) The fact that w is concave follows easily from the fact that $x_i \leq x_{i+1}$. Let $\overline{\alpha} = \alpha_0 < \ldots < \alpha_k$ be a sequence which is a candidate for being the level sequence of some tree – *i.e.*, $\alpha_0 = 0$, $\alpha_k = n - 1$, and $2\alpha_i - \alpha_{i-1} \leq n$ for all $0 < i \leq k$. We construct a tree $T_{\overline{\alpha}}$ which satisfies the following properties:

a) If $\overline{\alpha} = \overline{\alpha_T}$ for some left-justified binary tree T, then $T_{\overline{\alpha}} = T$ and $weight(\overline{\alpha}) = weight(T)$.

b) Otherwise, $weight(\overline{\alpha}) > weight(T_{\overline{\alpha}})$.

It follows that if $\overline{\alpha}$ has minimum weight, $\overline{\alpha} = \overline{\alpha_T}$, where T is the Huffman tree.

The construction of $T_{\overline{\alpha}}$ is as follows. Start with the ordered forest of n one-element trees. In step i, first take $2(\alpha_i - \alpha_{i-1})$ trees from the left and, for every j such that $2j \leq 2(\alpha_i - \alpha_{i-1})$, create a common parent for the trees which are on positions $2j - 1$ and $2j$ in the forest (counting from the left). □

4 THE PACKAGE MERGE ALGORITHM

First, we introduce a geometric interpretation of binary trees and forests. A more detailed description of the properties of such interpretation is given in [20].

Consider an $n \times n$ square in the plane, divided into n^2 unit squares, which we call *tiles.* (See Figure 4.1) We refer to the tile in the l^{th} row and the i^{th} column as $s_{l,i}$. We refer to l as the *level* of $s_{l,i}$, and to i as its *index.* We adopt the conventions that levels range from 0 to $n-1$, from the top down, while indices range from 1 to n, from left to right.

The weight, $w(s_{l,i})$, of a tile $s_{l,i}$ is defined to be x_i. The weight of a set of tiles is equal to the sum of

the weight of its members.

A tree, or a forest, can be represented as a connected union of "tiles ". Namely, if a $\ell_1, \ell_2, \ldots, \ell_n$ is a leaf-pattern of a tree T then tile $s_{l,i}$ belongs to the geometrical representation of T if and anly if $l \leq \ell_i$. Thus the cost of a tree is equal to weight of its geometric representation.

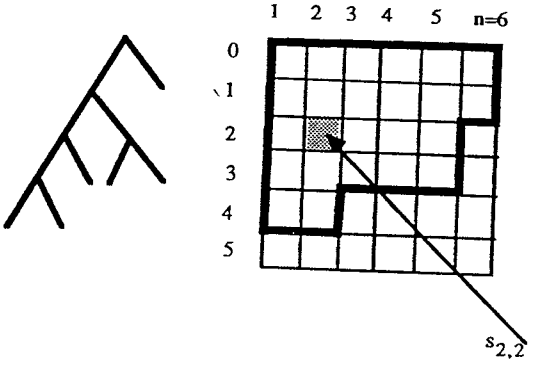

Figure 4.1 A tree and its geometric representation

We now give a jointly recursive (but still fairly simple) definitions of two concepts: *level i item* and *level i package*. A *level l item* is either a level l package or a level l tile. A *level l package* is a union of two level $l-1$ items. By this recursive definition it follows that a package is a union of tiles.

Two sets of tiles A, B are *tentatively connected* with respect to a set of tiles C (denoted T-C$(A, B; C)$) if and only if for any tile a from A and any tile b from B there exists a path of tiles between a and b which is contained in $A \cup B \cup C$.

An optimal alphabetic tree can be constructed iteratively as a union of certain packages. Let $\log n \leq L \leq n-1$. At each level l, starting from level L, the algorithm receives 0 or more *l-packages* from previous iteration. These are inserted (merged) into the list of level l tiles; both packages and tiles are called level l-items. These items are paired according to tentative-connectivity rules similar to [12], and resulting *packages* are exported to the next higher level; the odd item, if any is discarded. The set of 0-items defines the optimla tree.

ALGORITHM 2: **Package Merge(L)**

1. $PK_L := \emptyset; IT_L := \{s_{L,1}, s_{L,2}, \ldots, s_{L,n}\}$;

comment: PK_l is a set of level l tiles and IT_l is a set of level l items

2. **for** $l := L$ **downto** 1 **do**

   2.1. $IT_{l-1} := \{s_{l-1,1}, s_{l-1,2}, \ldots, s_{l-1,n}\}$; $PK_{l-1} := \emptyset; j := 1$;

   2.2. **while** $|IT_l| \geq 2$ **do**

      2.2.1. Pick $x, y \in IT_l$ of minimal total weight such that T-C$(x, y; PK_{l-1})$;

      2.2.2. $P_{l-1,j} := x \cup y$;

      2.2.3. $PK_{l-1} := PK_{l-1} \cup P_{l-1,j}$;

      2.2.4. $IT_l := IT_l - \{x, y\}; IT_{l-1} := IT_{l-1} \cup \{P_{l-1,j}\}$;

      2.2.5. $j := j + 1$;

3. *geom_repr* $:= PK_0 \cup \{s_{0,1}, s_{0,2}, \ldots s_{0,n}\}$

An $O(nL \log n)$-time sequential implementation of the Package Merge algorithm was presented in [17] (see also [18,19,20]). In the next section we present a new parallel implementation of this algorithm.

5 PARALLEL CONSTRUCTION OF OPTIMAL HEIGHT RESTRICTED ALPHABETIC TREES

In this section we show an efficient parallel implementation of a single iteration of the main loop of the Package Merge algorithm. Let $P_{l-1,j}$ be a package created at some iteration of Step 2.2 and let $s_{k,d}$ be an arbitrary tile in this package. A tile $s_{l-1,d}$ is called the *left dominating tile* of the package $P_{l-1,j}$ if $d = \max_{i<k}\{i | w(s_{l-1,i}) > w(P_{l-1,j})\}$. The left dominating tile may be undefined. Informally, the left dom-

inating tile for a package, P, is the closest to the left (with respect to any tile of the package) level l tile, which has weight greater than $w(P)$. For any l we represent the set of level l items (IT_l) as what we call a *normalized sequence of level l items.* With this representation we can compute efficiently, in parallel, all packages created at one iteration of step 2 of the Package Merge algorithm. Below we give a constructive definition of normalized sequence of level l items:

1. For each level l-package compute its left dominating tile;
2. Sort all packages which have the same left dominating tile according to weight in decreasing order,
3. Insert each (sorted) sequence of packages computed at Step 2. into $s_{l,1}, s_{l,2}, \ldots, s_{l,n}$; after its dominating tile (the sorted sequence of elements which do not have a left dominating tile is inserted before $s_{l,1}$).

LEMMA 5.1. *The normalized sequence for IT_{l-1} can be computed in $O(\log n)$ time with n processors, where n is the length of the sequence.*

Proof: Step 1 of the above construction can be implemented in $O(\log n)$ time with $n/\log n$ processors on a CREW PRAM using a parallel algorithm that for any element in a sequence computes its closest bigger neighbor ([5,15]). An implementation of the remaining steps is routine. □

Our algorithm to construct an optimal height restricted alphabetic tree can be described as follows:

ALGORITHM 3: **Parallel Package Merge**

1. **for** $l := L$ **downto** 1 **do**
   - 1.1 **package construction:** given a normalized sequence of level l items construct the $l-1$ level packages: $P_{l-1,1}, P_{l-1,2}, \ldots$;
   - 1.2 **package merge:** insert the set of constructed packages into the sequence $s_{l-1,1}$, $s_{l-1,2}$, ..., $s_{l-1,n}$ in such a way that the resulting sequence is normalized; these packages and slots become the level $(l-1)$ items.
2. *geom_repr* := $P_{0,1} \cup P_{0,2} \cup \ldots \cup P_{0,n-1} \cup \{s_{0,1}, s_{0,2}, \ldots s_{0,n}\}$;
3. Compute the actual alphabetic tree.

We need to show how to implement the package construction step. We base our implementation on the concept of *level tree* introduced in [15,22]. Let $V = v_1, v_2, \ldots, v_m$ be a normalized sequence of level l items. We add to this sequence two elements: $v_0, v_{m+1} = BR$ where BR (Big Real) is a real number greater than the cost of the optimal tree.

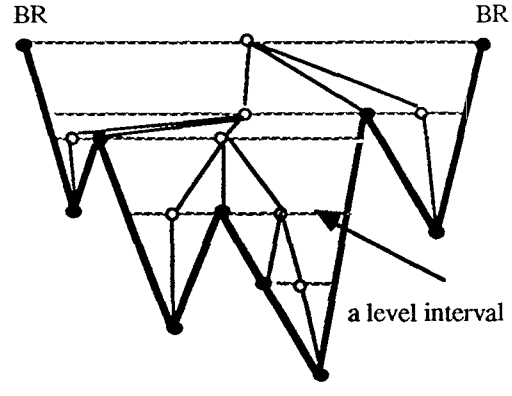

Figure 5.1 Construction of a level tree

Consider the following geometric construction (see Figure 5.1). For every i, there will be an associated point, (i, v_i), in the Cartesian plane. For every $i = 0, \ldots m$ there will be a line segment which connects the points (i, v_i) and $(i+1, v_{i+1})$, forming a polygonal path of $(m+1)$ segments. For every i such that $v_i < v_{i+1}$ (resp., $v_i > v_{i+1}$), draw a horizontal line going from (i, v_i) to the left (resp.,to the right) until it hits the polygonal path. The intervals defined in this way are called *level intervals.* We also consider the interval $[(0, BR), (m+1, BR)]$ and the degenerate

intervals $[(i, v_i), (i, v_i)]$ to be level intervals. The *level tree* is a binary tree whose nodes are the level intervals. The degenerate intervals are the leaves of the tree, the parent of any level interval is the unique closest level interval which is above it, and $[(0, BR), (m+1, BR)]$ is the root. A level treee is closely related to the *cartesian tree* introduced in [23]. In the contrasct to a cartesian tree a level tree need not to be binary.

LEMMA 5.2. ([15,22]) *Given a sequence of n weights the level tree can be constructed in $O(\log n)$ time using $n/\log n$ processors on CREW PRAM.*

Let v be a node of a level tree T_{lev}. The *pseudovalley defined by v* is the subsequence of a normalized sequence whose elements are the leaves of the minimal rooted subtree of T_{lev} that contains v. A pseudovalley that has an even (resp., odd) number of elements is called an *even pseudovalley* (resp., an *odd pseudovalley*). If U is a pseudovalley defined by v then $U = W_1 v W_2$ where W_i is a (possibly empty) pseudovalley. An empty pseudovalley is considered to be even.

For any element v of the normalized sequence of level l items we need to find the unique element v' such that v and v' are combined by the algorithm Package Merge into one package. We observe that elements from an even pseudovalley must be paired among themselves. Each odd pseudovalley is responsible for exporting one element which is paired with an element outside this pseudovalley. The "exported element" is the largest element of the valley, thus tis is the element that defines the valley. This observation leads to the following *Package Construction Rules*:

Package Construction Rules:

1. If v defines a pseudovalley $U = W_1 v W_2$ where both W_i are even then U is odd and v is the element exported by U.
2. If v defines a pseudovalley $U = W_1 v W_2$ where one of W_i (say W_1) is odd and the other is even then U is even and v is paired with the element exported by W_1.
3. Assume that v defines a pseudovalley $U = W_1 v W_2$ where both of W_i are odd. Let w_1 (resp. w_2) be the element exported by W_1 (resp., W_2) and $w_1 < w_2$. Then v is paired with w_1 and w_2 is the element exported by U.

LEMMA 5.3. *Given a normalized sequence of level l items, IT_l, the Package Construction Rule defines exactly the same set of level $l-1$ as it is constructed in step 2 of algorithm Package Merge.*

Proof: We leave the proof to the full version of the paper. The following two observations gives useful hints.

First, two minimal items of any pseudovalley defined by IT_l are tentatively connected with respect to PK_{l-1}. In Step 2. of the Package Merge algorithm, we take the minimal such pair remove it form IT_l, and insert their union into PK_{l-1}. The sequence IT_l remains normalized. Furthermore, two minimal items of any pseudovalley of the modified sequence IT_l are tentatively connected with respect to modified set PK_{l-1}. Thus we can repeat this step iteratively.

Second, the above construction can be performed independently in disjoint pseudovalleys. □

With the Package Construction Rules the package construction step can be implemented in $O(\log n)$ time with $n/\log n$ processors using tree contraction techniques [1,21].

To implement Step 2 of the Parallel Package Merge algorithm we need to identify packages $P_{0,1}, P_{0,2}, \ldots, P_{0,n-1}$ and compute their union. To do this, for each tile we determine whether if belongs to the geometric representation of the optimal tree. This can be done in $O(k \log^2 n)$ time with n processors by a level-by-level computation.

Finally step 3 can be implemented as follows. If tile $s_{i,j}$ belongs to the geometric representation and tile $s_{i+1,j}$ does not, then the j^{th} leaf of the optimal tree has depth i. Given a sequence of leaf depths one can construct the corresponding tree in $O(\log n)$ time

with $n/\log n$ processors [15,22].

Thus the algorithm can be implemented in $O(L \log n)$ time with linear number of processors. If L is polylogarithmic this gives immediately a polylogarithmic time algorithm. The algorithm can be also used to find an approximately optimal tree. This observation is made precise in the following theorem.

THEOREM 5.1. *Given a sequence of reals $X = x_1, x_2, \ldots, x_n$ such that $\sum_{i=1}^{n} x_i = 1$, and constant k, an alphabetic tree whose cost differs at most by $1/n^k$ from the cost of an optimal alphabetic tree for X can be constructed in $O(k \log^2 n)$ time with n CREW PRAM processors. Furthermore, if X does not contain two consecutive elements of weight less than $1/n^k$ then the resulting tree is optimal.*

Proof: We use the following lemma, which follows from the results of [9].

LEMMA 5.4. *An optimal alphabetic tree for a sequence of reals $X = x_1, x_2, \ldots, x_n$ such that $\sum_{i=1}^{n} x_i = 1$ and no two consecutive elements are less than $1/n^k$ has height at most $k\lceil \log n \rceil$.*

Let T be an optimal tree for the sequence X. The above lemma leads immediately to an $O(k \log^2 n)$-time n-processor CREW PRAM which computes T if the sequence does not contain two consecutive elements of weight less than $1/n^k$. Thus, assume that X contains one or more subsequences of consecutive elements of weight less than $1/n^k$. We replace each such subsequence with an arbitrary one of its element called the *representative* of the subsequence. By Lemma 5.4, the tree, T', which is optimal for the new sequence, has height at most $k\lceil \log n \rceil$. Now replace every leaf of T' that is a representative of a subsequence of elements of small weight by the full binary tree build on the subsequence. The resulting tree T'' has height at most $(k+1)\lceil \log n \rceil$. Let T_k be an optimal height $(k+1)\lceil \log n \rceil$ restricted tree. Observe that

$$c(T_k) \leq c(T'') \leq c(T') + n^2/n^k \leq c(T) + 1/n^{k-2}$$

which together with the parallel implementation of the Package Merge Algorithm implies the theorem. □

6 Open Questions

The best sequential algorithm for the Concave Least Weight Subsequence problem takes $\Theta(n)$ time, and (of course) $\Theta(n)$ work, while every known NC algorithm takes $\Omega(n^2)$ work.

If we define "work-time" to be the product of the time complexity and the work complexity of an algorithm, every known algorithm for the CLWS has $\Omega(n^2)$ "work-time" complexity. The algorithm we have presented in this paper is no exception, having a work-time complexity of $\Theta(n^2 \log^2 n)$.

We conjecture that there should exist a family of parallel algorithms exhibiting various tradeoffs between work and time, ranging between the best sequential and the best known NC algorithms, but all with quadratic times polylog work-time. (Our new algorithm is just in the middle of this hypothetical family.) We also suggest that finding a parallel algorithm whose work-time is less than quadratic will be a very challenging problem.

References

[1] K.Abrahamson, N.Dadoun, D.G.Kirkpatrick, and T.M.Przytycka. A simple parallel tree contraction algorithm, *Journal of Algorithms* 10, (1989) pp. 287–302 also in Proc. 25th Allerton Conference on Communication, Control and Computing (1987).

[2] A.Apostolico, M.J.Atallah, L.L.Larmore, H.S.McFaddin. Efficient parallel algorithms for string editing and related problems, *Proc. 26nd Annual Allerton Conf. on Comm., Control, and Computing*, Monticello IL (Oct. 1988) pp. 253–263. Reprinted as CSD-TR-724, Purdue University (1988), reprinted in *SIAM Journal of Computing*, 19 pp.968–988 (1990).

[3] A.Aggarwal, J.Park. Notes on searching in multidimensional monotone arrays. In *29th Annual Symposium on Foundation of Computer Science* IEEE (1988).

[4] M.J.Atallah, S.R.Kosaraju, L.L.Larmore, G.L.Miller, and S-H. Teng. Constructing trees in parallel, Proc. 1st Symp. on Parallel Algorithms and Architectures (1989) pp. 499–533.

[5] O.Berkman, D.Breslauer, Z.Galil, B.Schieber, and U.Vishkin. Highly parallelizable problems, Proc. 21st ACM Symp. on Theory of Computing (1989) pp. 309–319.

[6] K-F Chan and T-W Lam. Finding Least-Weight Subsequences with Fewer Processors, *Proceedings of the LNCS* pp.318–327. (1990)

[7] Z. Galil and R. Giancarlo. Data Structures and Algorithms for Approximate String Matching, Tech. Report, CS Dept., Columbia University, NY (1987).

[8] M.C. Golumbic. Combinatorial Merging, *IEEE Trans. Comp.* 25, 11 (1976) 1164–1167.

[9] R. Güttler, K. Mehlhorn and W. Schneider. Binary search trees: average and worst case behavior, *Electron. Informationsverarb Kybernet*, 16 (1980) pp. 41–61.

[10] D.S.Hirschberg, and L.L.Larmore, The Least weight subsequence problem, *Proc. 26th Annual Symp. on Foundations of Computer Science*, Portland Oregon (Oct. 1985), pp. 137-143. Reprinted in *SIAM Journal on Computing* 16 (1987) pp. 628–638.

[11] T.C. Hu and K.C. Tan. Path length of binary search trees, *SIAM J. Appl. Math.* 22, pp. 225-234 (1972).

[12] T.C.Hu and C.Tucker. Optimum computer search trees. *SIAM J.Appl. Math.*, 21, 1971, pp 514–532.

[13] D.A. Huffman. A method for the constructing of minimum redundancy codes, Proc. IRE, 40, 1952, 1098–1101.

[14] D.G.Kirkpatrick and T.M.Przytycka. Parallel construction of binary trees with almost optimal weighted path length, Proc. 2nd Symp. od Parallel Algorithms and Architectures (1990).

[15] D.G.Kirkpatrick and T.M.Przytycka. An optimal parallel minimax tree algorithm, Proc. of the 2nd IEEE Symp. of Parallel and Distributed Processing (1990) 293–300.

[16] M.M. Klawe and D.J. Kleitman, An almost linear time algorithm for generalized matrix searching, RJ 6275, IBM - Research Division, Almaden Research Center, (1988)

[17] L.L. Larmore. Length limited coding and optimal height-limited binary trees, TR 88-01 University of California, Irvine (1988).

[18] L.L. Larmore, D.S. Hirschberg. A Fast algorithm for optimal length-limited codes *Journal of the ACM*, (1990) pp. 464–473.

[19] L.L. Larmore, and D.S.Hirschberg. Length-limited coding, *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms* San Francisco, CA. (January 1990) pp. 310–318.

[20] L.L.Larmore, and T.M.Przytycka, A Fast Algorithm for Optimum Height Limited Alphabetic Binary Trees, manuscript.

[21] G.L. Miller and J. Reif. Parallel tree contraction and its application, Proc. 26th IEEE Symp. on Foundation of Computer Science (1985) pp. 478–489.

[22] T.M. Przytycka. *Parallel Techniques for Construction of Trees and Related Problems.* Ph.D thesis, the University of British Columbia, Vancouver (1990).

[23] J.Vuillemin, A unifying look at data structures, *C. ACM*, 23, 4 1980, pp 229–239.

[24] R. Wilber. The concave least weight subsequence problem revisited, *Journal of Algorithms* 9, 3 pp.418–425 (1988).

[25] F.F. Yao, Efficient dynamic programming using quadrangle inequalities, *Proc. of the 12th ACM Symp. on Theory of Computing*, (1980) pp.429–435.