Recovering the ℓ-tree. It does not suffice to know the value of $\left|forest^i\right|$. We must also know the structure of the ℓ-tree, *i.e.*, the sons of each p^i.

Notice that $forest^j$ results from $forest^{j-1}$ by a single operation *combine*. This operation creates the next root node p^j and links two nodes, each of them a root of $forest^{j-1}$ or an original item, to p^j as its sons. We know also that the roots of $forest^j$ are $p^{i+1} \ldots p^j$, where $i = deroot(j)$. This imposes certain combinatorial relations between sets of roots and leaves of successive forests. The following fact and the next lemma are simple consequences of such combinatorial relations.

Let $\#(S)$ denote the cardinality of any set S.

Fact 5.6 *Let $i = deroot(j)$ and $i' = deroot(j-1)$. Then $\#(leaves^j) = \#(leaves^{j-1}) + 2 - (i - i')$.*
Furthermore, $i - i' \in \{0, 1, 2\}$.

Lemma 5.7 *Let $0 < j < m$, and let $i = deroot(j)$ and $i' = deroot(j-1)$. Then one of the following three situations holds.*

(i) *$i - i' = 2$, p^j has sons p^i and p^{i-1}, and $leaves^i = leaves^{i-1}$.*

(ii) *$i - i' = 1$, p^j has sons p^i and a_t for some t, and $leaves^i = leaves^{i-1} \cup \{a_t\}$.*

(iii) *$i - i' = 0$, p^j has sons a_t and a_r for some t, r, and $leaves^i = leaves^{i-1} \cup \{a_t, a_r\}$.*

We conclude that the ℓ-tree of α can be recovered in $O(\log m)$ time using m processors, provided the sets W_k and the function *deroot* are known, since $leaves^j = W_k$, where k $= \#(leaves^j)$.

Lemma 5.8 *All sets W_k can be computed in $O(\log m)$ time with m processors.*

Thus we can execute PROCESS(α, Δ) in $O(\log^2 m)$ time with $m^2/\log m$ processors. Our main result follows.

Theorem 5.9 (main result) *An optimal alphabetic tree can be constructed in $O(log^3 n)$ time with $n^2/\log n$ processors of a CREW PRAM.*

Open problems

I. Can we compute alphabetic trees in polylogarithmic time with $O(n)$ processors?
II. Can we solve the OBST problem in polylogarithmic time with $O(n^k)$ processors, for some $k < 6$?

References

[1] A. Apostolico, M. J. Atallah, L. L. Larmore, H. S. McFaddin. Efficient parallel algorithms for string editing and related problems, *SIAM Journal of Computing* **19** (1990), pp. 968–988.

[2] A. Aggarwal, J. Park. Notes on searching in multidimensional monotone arrays. *Proc. 29th IEEE Symposium on Foundation of Computer Science* (1988), pp. 497–513.

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

[4] H. N. Gabow, J. L. Bentley, R. E. Tarjan, Scaling and related techniques for geometry problems, *Proc. 16th ACM Symposium on Theory of Computing* (1984), pp. 135–143.

[5] A. M. Garsia and M. L. Wachs, A New algorithm for minimal binary search trees, *SIAM Journal of Computing* **6** (1977), pp. 622–642.

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

[7] T. C. Hu. A new proof of the T-C algorithm, *SIAM Journal of Applied Mathematics* **25** (1973), pp. 83–94.

[8] T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM Journal of Applied Mathematics* **21** (1971), pp. 514–532.

[9] D. A. Huffman. A Method for the constructing of minimum redundancy codes, *Proc. IRE* **40** (1952), pp. 1098–1101.

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

[11] D. E. Knuth. *The Art of computer programming*, Addison–Wesley (1973).

[12] D. E. Knuth. Optimum binary search trees, *Acta Informatica* **5** (1971), pp. 14–25.

[13] L. L. Larmore, and T. M. Przytycka, Parallel construction of trees with optimal weighted path length, *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures* (1991), pp. 71–80.

[14] L. L. Larmore, and T. M. Przytycka, A Fast algorithm for optimum height limited alphabetic binary trees, Submitted for publication.

[15] W. Rytter, Efficient parallel computations for some dynamic programming problems, *Theo. Comp. Sci.* **59** (1988), pp. 297–307.

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

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

Example. Let α =(1 7 8 9 2 10 11 3 17), from Figure 2. Then

$$\begin{aligned} W_2 &= \{1,7\} \\ W_3 &= \{1,7,8\} \\ W_4 &= \{1,7,9,2\} \\ W_5 &= \{1,7,8,9,2\} \\ W_6 &= \{1,7,9,2,11,3\} \\ W_7 &= \{1,7,8,9,2,11,3\} \\ W_8 &= \{1,7,8,9,2,10,11,3\} \\ W_9 &= \{1,7,8,9,2,10,11,3,17\} \end{aligned}$$

Lemma 5.2 *$leaves^i$ is lf-closed.*

For each $i > 0$, $leaves^i = W_k$ for some $k \geq 2$. (See Figure 2.)

The following lemma is the heart of the reduction to the CLWS problem.

Lemma 5.3 *Let $0 < j < m$. Then there exists $0 \leq i < j$ such that*

(i) *the roots of $forest^j$ are $p^{i+1} \ldots p^j$.*

(ii) *$leaves^j = W_{2j-i}$.*

(iii) *$\left|forest^j\right|$ is the weight of the minimum weight path in $\mathcal{G}(\alpha)$ from 0 to j.*

For $0 < j < m$, we define $deroot(j) = i$ where i is as given in Lemma 5.3. For completeness, let $deroot(0) = 0$. If all roots of $forest^j$ are removed, and all items which thereby become singletons are removed, the resulting forest is $forest^i$.

Proof:(sketch of proof of Lemma 5.3.) The correctness of (i) and (ii) is straightforward. (We use the fact that for any two packages p_i, p_j if $i < j$ then p_i is combined first.) By Lemma 5.2, $\left|forest^j\right| \geq \left|forest^i\right| + w(i,j)$.

Claim: For any $0 \leq k < j$, $\left|forest^j\right| \leq \left|forest^k\right| + w(k,j)$.

We prove the claim by induction on lexically ordered pairs (j,k), $0 \leq k < j < m$. The basis, $k = 0, j = 1$ follows immediately.

Let X be an *lf*-closed subsequence of a sequence α. We define the *closure* of X, $cl(X)$ to be the smallest prefix of α that contains all elements of X. We say that Y *covers* X if $cl(X) \subseteq Y$.

From [14], $\left|forest^i\right| = \left|opt\_forest^i\right|$, where opt_forest^i is the optimal alphabetic forest of i internal nodes.

Now assume that the inequality (iii) is true for any pair $(k', j') < (k, j)$. (Where "<" is defined lexically.)

Let $h = deroot(k)$. We consider two cases.

Case 1: W_{2j-k} covers W_{2k-h}.

We explicitly construct an alphabetic forest with j internal nodes whose weight is $\left|forest^k\right| + w(k,j)$.

Write $W_{2j-k} = U \cup U'$, where U is the maximal prefix of α contained in W_{2j-k}, and U' is the disjoint union of locally minimal pairs. By our hypothesis, $W_{2k-h} \subseteq U$. Combine the consecutive pairs of roots of the trees of opt_forest^i whose leaves belong to U. Then combine the locally minimal pairs in U'. The resulting forest has j internal nodes, thus its cost is at most $\left|opt\_forest^j\right| = \left|forest^j\right|$.

Case 2: W_{2j-k} does not cover W_{2k-h}.

In this case the following fact holds:

Fact 5.4

$$w(h,k) + w(k,j) \geq w(h,k-1) + w(k-1,j).$$

Proof: Let z be the smallest element of $cl(W_{2k-h})$ not in W_{2j-k} and let (x,y) be the largest locally minimal pair in W_{2j-h}, where y is a local minimum. Since x is an ancestor of z in the Cartesian tree, $x + y > z$. By minimality of the W_k,

$$\begin{aligned} w(h,k-1) &= |W_{2k-h-2}| \leq |W_{2k-h} - \{x,y\}| \\ w(k-1,j) &= |W_{2j-k+1}| \leq |W_{2j-k} \cup \{z\}| \end{aligned}$$

proving Fact 5.4. □

Applying the inductive hypothesis twice,

$$\begin{aligned} \left|forest^k\right| + w(k,j) &= \left|forest^h\right| + w(h,k) + w(k,j) \\ &\geq \left|forest^h\right| + w(h,k-1) + w(k-1,j) \\ &\geq \left|forest^{k-1}\right| + w(k-1,j) \\ &\geq \left|forest^j\right|. \end{aligned}$$

Proving the claim. Part (iii) follows. □

Reduction to the CLWS. The $\{w(i,j)\}_{0 \leq i < j < m}$ define an instance of the LWS. Let $\mathcal{T}$ be the tree of shortest paths starting at 0 in the graph $\mathcal{G}(\alpha)$. Then, for $j > 0$, lemma 5.3 implies:
if $deroot(j) = i$ then $|forest^j| = |forest^i| + w(i,j)$;
i is the father of j in the tree $\mathcal{T}$.

Lemma 5.5 (quadrangle) *The $\{w(i,j)\}$ satisfy the* quadrangle inequality (2)*. Thus, the the single-source shortest path problem for the graph $\mathcal{G}(\alpha)$ is an instance of the CLWS. The function deroot can be constructed in $O(\log^2 m)$ time with $m^2/\log m$ processors.*

Procedure SHRINK(X, β);
Let $X = \{(u_1, w_1), (u_2, w_2), ..., (u_k, w_k)\}$;
for each $1 \le i \le k$ **parallel do**
r_i:= *combine*(u_i, w_i);
mark r_i as a *package* which should be shifted;
Comment: As a side effect a forest of two-leaf subtrees is created through new links *father* and each pair u_i, w_i is replaced by a single *package* ;
end procedure;

Lemma 4.1 (predestination) *Let* $X \subseteq$ PRE(β). *Then the tree constructed by* MAIN(β) *is isomorphic to the tree constructed by the following algorithm:*

SHRINK(X, β); MERGEALLPACKAGES(β); MAIN(β).

Regular valleys. We define a *regular valley* to be a 1-valley α where, if u, v are leaves in the Cartesian tree of α and $u < v$, then $father(u) \le father(v)$.

A regular valley is *sorted* if the global minimum is the first item, and if all leaves of the valley tree, except for the global minimum, are right children of their *father* nodes in the valley tree.

Lemma 4.2 *For each regular valley* α *there is a unique sorted regular valley* α' *such that,* $\alpha \equiv \alpha'$, *and thus* ℓ*-Tree(*α*)* $\equiv$ ℓ*-Tree(*α'*). Furthermore, this sorted regular valley can be found in* $O(\log m)$ *time with* $m/\log m$ *processors, where* $m =$ *length of* α.

By Lemma 4.2 and Lemma 2.1 we can assume without loss of generality that each regular valley is sorted.

Special pairs. We say that a pair of adjacent items (u, v) is a *special* pair of a sequence α if

1. u is a leaf and v is the father of u in the Cartesian tree of α, and
2. the subtree of the Cartesian tree rooted at v has a leaf larger than u.

Let SPECIAL(α) be the special pairs of a sequence α.

Lemma 4.3 SPECIAL$(\alpha) \subseteq$ PRE(α).

Lemma 4.4 (transformation lemma) *A 1-valley becomes a regular valley after executing the following algorithm:*
SHRINK(SPECIAL$(\alpha), \alpha)$; MERGEALLPACKAGES(α).
which can be computed in $O(\log m)$ *time with* $m/\log m$ *processors, where* m *is the length of* α.

5 Parallel preprocessing of regular valleys

In this section, we implement PROCESS(α, Δ) in parallel for any regular valley α, in $O(\log^2 m)$ time with $m^2/\log m$ processors, where m is the length of α. The heart of this implementation is the reduction to the CLWS problem.

Packages and intermediate forests. Throughout this section, let $\alpha = (a_1, \ldots a_m)$ be a regular valley. Without loss of generality, by Lemma 4.2, α is sorted.

Let p^i be the i^{th} package (round node) created by the algorithm MAIN during construction of the ℓ-tree for α. Let $forest^i$ be the i^{th} ℓ-forest created in the process, the forest whose internal nodes are $p^1, \ldots p^i$, and whose leaves are those items of α which are descendants of those nodes. (Thus, $forest^i$ has no singleton trees.) We say that i is the *index* of $forest^i$. Let $leaves^i$ be the set of leaves of $forest^i$.

Let ℓ_k^i be the distance from a_k to its root in $forest^i$, which we take to be zero if $a_k \notin leaves^i$.

We define the *weight* of $forest^i$ to be

$$\left|forest^i\right| = \sum_{k=1}^{i} a_k \ell_k^i.$$

Similarly, if A is any subset of $\{a_1, \ldots a_n\}$, we define the *weight* of A, which we write $|A|$, to be the sum of the weights of members.

Remark 5.1 *The node* p^j *is a root of* $forest^j$. *If this root is deleted, and items that become singletons as a result of this deletion are also deleted, the resulting forest is* $forest^{j-1}$.
Assume that after removing all roots of $forest^j$ *and all items that become singletons as a result, we obtain a forest with* i *internal nodes. Then* $forest^j$ *has* $(2j - i)$ *leaves.*

The graph $\mathcal{G}(\alpha)$. A set of items A is defined to be *lf-closed* if whenever $a_i \in A$ and a_i is a leaf in the Cartesian tree of α, then $father(a_i) \in A$. ("*lf*" = "leaf-father.")

Let $\mathcal{G}(\alpha)$ be the weighted directed acyclic graph whose nodes are the integers $0, 1, \ldots m-1$, and where there is an edge of weight $w(i, j) = |W_{2j-i}|$ from i to j for all $0 \le i < j < m$ such that $2j - i \le m$, where W_k is the minimum weight *lf*-closed set of cardinality k.

Since α is sorted, W_k always consists of a prefix of α together with zero or more locally minimal pairs of α.

Proof:(sketch) The proof follows by induction on the number of nodes in the valley tree of the valley α. The base case follows by results of [7]. The inductive step is a natural consequence of the choice of the value Δ. □

3 Parallelization of algorithm MAIN

One straightforward method for parallelizing the algorithm MAIN would be to perform $combine(u, w)$ for many locally minimal pairs in parallel, *e.g.*, as many as possible, followed by MERGEALLPACKAGES, and then repeat these two operations until the list consisted of a single item. Unfortunately, in the worst case, the number of iterations needed is linear. For example, if β is sorted, there will be only one locally minimal pair at every step, and $(n-1)$ steps will be required. We thus need to develop a more sophisticated approach to obtain an $\mathcal{NC}$ algorithm.

We define a *1-valley* to be a valley α such that each node of val-Tree(α) of degree two has at least one son which is a leaf.

A 1-valley is a weighted list whose non-singleton valleys form a chain under inclusion. For example, 57234 is a 1-valley, while 45723 is not a 1-valley, since 45 and 23 are non-singleton valleys that are disjoint.

Algorithm PARALLELMAIN(β)

1. **while** $|\beta| \geq 2$ **do**
   - 1.1 find the set VAL of all maximal 1-valleys
   - 1.2. **for each** 1-valley $\alpha \in$ VAL **parallel do** replace α by PROCESS(α, Δ_α); **Comment:** as a side effect many links *father* are created
   - 1.3. MERGEALLPACKAGES(β);
2. **return** the tree given by the links *father*

Consider our example sequence $\beta =$ (17 8 24 10 16 14 18 12 19 22 27 11 21 23 15 25 20 26). There three maximal 1-valleys are:

$$\begin{aligned} val_1 &= (17, 8) \\ val_2 &= (10, 16, 14, 18, 12, 19, 22) \\ val_3 &= (11, 21, 23, 15, 25, 20, 26) \end{aligned}$$

We have

$$\begin{aligned} \text{VAL} &= \{val_1, val_2, val_3\} \\ \text{PROCESS}(val_1, 24) &= (25') \\ \text{PROCESS}(val_2, 24) &= (22, 26', 30', 33') \\ \text{PROCESS}(val_3, 27) &= (26, 32', 38', 45'). \end{aligned}$$

After processing these valleys we have the sequence:

$$\beta = (25', 24, 22, 26', 30', 33', 27, 26, 32', 38', 45').$$

Next we execute MERGEALLPACKAGES(β) and obtain:

$$\beta = (24, 22, 25, 26, 27, 26, 30, 32, 33, 38, 45).$$

At the same time an ℓ-forest is computed as an effect of applying many operations *combine* (which produce *marked* elements). This forest is illustrated in Figure 1.c. This completes one iteration of PARALLELMAIN.

Lemma 3.1 (tree contraction) *The number of iterations of the algorithm* PARALLELMAIN *is logarithmic.*

Proof:(sketch) If we look at each iteration from the perspective of the valley tree then each iteration is like a tree contraction. At each iteration, we remove all 1-chains: subtrees corresponding to maximal 1-valleys. and insert some number of new packages into the tree. Note, however that a package can only subdivide an edge of the "old" tree and never creates a node of degree 2. More formally, define a *branching node* of the valley tree to be an internal node that has two non-leaf sons. At each iteration the number of branching nodes in the current valley tree is reduced to fewer than half the previous number, and if there are no branching nodes, the entire sequence consists of a single 1-valley. Hence the number of iterations is logarithmic. □

4 Parallel preprocessing of 1-valleys: regular valleys

To complete the parallel implementation of the algorithm MAIN it is sufficient to execute PROCESS(α) in $O(log^2 m)$ time with $m^2/\log m$ processors for a 1-valley α, where m is the number of elements of the 1-valley.

Parallel processing of a 1-valley is in two steps. First, all "special" pairs are identified and processed. Second, the resulting valley, which is now a "regular valley," is processed by reduction to the CLWS.

Predestination. We say that a pair of adjacent items in the sequence β are *predestined* if they are brothers (sons of the same node) in the ℓ-tree constructed by the algorithm MAIN. In other words, (u, w) is a predestined pair if, at some iteration of the algorithm MAIN(β), "$r :=$ $combine(u, w)$" will be executed. For example, all locally minimal pairs are predestined. Let PRE(β) be the set of predestined pairs of β.

The key property of predestined pairs is that any number of them can be processed in parallel using the following procedure.

valley. Each valley is of the form $\alpha_1 b_k \alpha_2$, where each α_i is either the empty list or is itself a valley which is a *son* of $val(b_k)$. In this way we define the valley tree of β, denoted val-Tree(β).

The subtree of val-Tree(β) corresponding to a valley α is denoted by val-Tree(α).

The *Cartesian tree* [4] of a sequence is defined very similarly, and in fact is isomorphic to the valley tree. The nodes of the Cartesian tree are the list items themselves. This isomorphism is obtained by replacing each valley by its representative (*i.e.*, maximum) item. [See Figure 1.]

Example. Let us take the following sequence $\beta =$ (17 8 24 10 16 14 18 12 19 22 27 11 21 23 15 25 20 26).

Then $val(19) =$ (10 16 14 18 12 19) and $val(23) =$ (11 21 23 15). The valleys and the Cartesian tree for this example are presented in Figure 1.

Each proper valley α has a *smaller neighbor*, Δ_α, defined to be the parent of the representative of α in the Cartesian tree. In our example, for instance, the smaller neighbor of $val(19)$ is 22.

Equivalence. We define a *rotation* of a binary tree to be the exchange of the left and right sons of some node. Then two binary trees are *isomorphic* in the sense of unordered trees if one can be changed into the other in a sequence of rotations. Denote this relation by "$\equiv$."

An important relation between isomorphism of ℓ-trees and valley trees is given in the following lemma:

Lemma 2.1 *If val-Tree(β) $\equiv$ val-Tree(β') then*

$$\ell\text{-}Tree(\beta) \equiv \ell\text{-}Tree(\beta').$$

Computing Packages. Let α be a valley subsequence of a sequence β. Let $\Delta = \Delta_\alpha$, the smaller neighbor of α. The following algorithm, PROCESS, processes α independently of the rest of the sequence. It iteratively computes packages until at most one item less than Δ remains. PROCESS returns the sequence of roots of the forest it computes.

We illustrate the algorithm for our example sequence. Consider val_2 from Figure 1. The smaller neighbor of val_2 is $\Delta = 24$. The computation of PROCESS(val_2, Δ) (where *prime* means a marked element) is:

$$\begin{aligned}(10,16,14,18,12,19,22) &\Rightarrow (14,18,12,19,22,26') \Rightarrow \\ (14,19,22,26',30') &\Rightarrow (22,26',30',33')\end{aligned}$$

Hence:
PROCESS($(10,16,14,18,12,19,22),24$) $= (22,26',30',33')$.

Algorithm PROCESS(α, Δ);

Comment: α is a valley;

Comment: $\Delta = \Delta_\alpha$
MakeEmpty(*Queue*);

1. **while** $|\alpha| \geq 2$ **do**

   1.1. select any *locally minimum* pair (u, w)

   1.2. $r :=$ *combine*(u, w);

   1.3. **if** *weight*$(r) \leq \Delta$ **then**
   $\alpha :=$ *MergeSinglePackage*(r, α)
   else *mark* r as a package to be processed later; *insert*$(r,$ *Queue*$)$;

Comment: denote the forest defined by computed links *father* in the combination steps by ℓ-Forest(α, Δ);
Queue = the sequence of roots of this forest (left-to-right);

2. **return** the concatenation $\alpha \bullet$ *Queue*.

The following algorithm, MERGEALLPACKAGES(β), generalizes the operation *MergeSinglePackage*(r, β) to the set of all marked elements of the sequence β.

Algorithm MERGEALLPACKAGES(β);
Let Q be the set of marked elements of β
for each $r \in Q$ **do**
MergeSinglePackage(r, β);

Assume that F' is a forest whose roots are contained in the set of leaves of the tree T'. Denote by $T' \oplus F'$ the tree resulting by connecting the subtree of F' whose root is any r to the leaf r of T'.

Let T be the ℓ-tree of some sequence. Denote by *Cutoff*(T, Δ) the forest resulting by cutting off all *uplinks* from nodes larger than Δ.

The following lemma shows that each valley can be processed independently of the other ones. This permits the parallelization of the algorithm MAIN.

Lemma 2.2 (Independence lemma)
Assume that $\alpha_1, \alpha_2, ..., \alpha_k$ are disjoint valleys of β. Let β' be the sequence obtained from β by replacing each α_i with PROCESS(α_i, Δ_i)*, where Δ_i is the smaller neighbor of α_i. Let $T' =$* MAIN(MERGEALLPACKAGES(β')) *and F_i be the forest created in the call* PROCESS(α_i, Δ_i) *Then*

1. *ℓ-Tree(β) $\equiv T' \oplus F_1 \oplus F_2 ... \oplus F_k$.*
2. *$F_i \equiv$ Cutoff$(\ell\text{-}Tree(\alpha_i), \Delta_i)$*

cessed by reduction to the CLWS, resulting in at most one item of the original 1-valley together with some number of "marked" items. These marked items are then reinserted into the list by moving each to the place just to the left of its nearest larger right neighbor, completing one top-level iteration.

Using standard tree-contraction techniques, it can be shown that the number of maximal 1-valleys is at least halved at each iteration. If the list has just one maximal 1-valley, it is processed to a single item, which is then the root of the ℓ-tree.

Complexity. Maximal 1-valleys can be found in $O(\log n)$ time using $n/\log n$ processors. Processing a 1-valley of length m requires $O(\log^2 m)$ time using $m^2/\log m$ processors. The total length of all 1-valleys processed during the entire algorithm is $O(n)$. The number of top-level iterations is $O(\log n)$. Finally, the OAT can be constructed in $O(\log n)$ from the ℓ-tree using $n/\log n$ processors [10].

We conclude that our algorithm takes $O(\log^3 n)$ time and uses $n^2/\log n$ processors.

2 A Sequential algorithm

In this section, we introduce a slight modification of the Garsia-Wachs algorithm, and in section 3 we give a parallelization.

Let $\beta = (b_1, \ldots b_k)$. We refer to $b_i + b_{i+1}$ as the i^{th} 2-sum, for $1 \leq i < k$. A pair of consecutive elements (b_i, b_{i+1}) is said to be *locally minimal* if the i^{th} *2-sum* is a local minimum in the sequence of 2-sums. Every locally minimal pair must eventually be combined ("packaged").

Combining one pair. The algorithm performs $(n-1)$ iterations. At each iteration, a new node, r, called a *package*, is created through the operation *combine* defined below. This node is *marked*, which means that it must later be moved and unmarked.

Function *combine*(u, w);
  Comment: u, w are adjacent elements in β
  $r := u + w$;
  create links
    $father(u) = r$ and $father(w) = r$;
  mark r as a package to be processed;
  remove u and w from β and replace by
    the single element r;
  return r;
end function.

We call this step *combining* the pair (u, w) and denote it as a function (with side-effects):

$$r := combine(u, w).$$

We define the operation

$$MergeSinglePackage(r, \beta).$$

as follows: A marked package r is moved in the actual working sequence β to the place immediately before the first element r' which is to the right of r and whose weight is not smaller than weight of r. After this reinsertion, r is unmarked.

In order for *MergeSinglePackage* to be defined in all cases, we assume the existence of a fictitious infinite item after the last item in the list.

The sequential algorithm MAIN. The difference between our sequential algorithm and Garsia-Wachs algorithm is that we take *any* locally minimal pair, rather than necessarily the leftmost minimal pair.

Algorithm MAIN(β);

1. **while** $|\beta| \geq 2$ **do**
   - 1.1 Choose an arbitrary locally minimal pair (u, w);
   - 1.2. $r := combine(u, w)$;
   - **Comment:** u, w are replaced by r, and two links *son-father* are created;
   - 1.3. $MergeSinglePackage(r)$;
2. ℓ-Tree := tree defined by the created *father*-links
3. **return** ℓ-Tree

After each iteration of the main loop in Algorithm MAIN the function *father* computed so far defines a forest. The roots of this forest are the elements of the current sequence, and its leaves are the elements of the initial sequence. It is possible that some of the leaves are roots of a single-node components.

Valleys. It was first observed by Hu [7] that disjoint "valley"-shaped subsequences can be processed to a certain extent independently of each other. This fact is central to our algorithm and it motivates the definitions below.

Formally we define a *valley* as follows. Given an item b_k in the list β, let $vall(b_k, \beta)$ be the largest (contiguous) subsegment of β containing b_k which contains no item larger than b_k. We say b_k is the *representative* of its

in the class $\mathcal{NC}$ up to now.

The Concave Least Weight Subsequence Problem. We make use of a reduction to the *Least Weight Subsequence* (LWS) problem: given a triangular array of weights $\{w(i,j)\}_{0\le i<j<n}$, choose a monotonic sequence of integers $0 = i_0, i_1, \ldots i_m = n$ which minimizes the sum $\sum w(i_k, i_{k+1})$. This problem can be solved in $O(n)$ sequential time if the weight function is *concave*, *i.e.*, if for all $0 \le i \le i' < j \le j' \le n$,

$$w(i,j) + w(i',j') \le w(i,j') + w(i',j) \qquad (2)$$

The inequality (2) is also called the *quadrangle inequality* [17]. This condition specifies the so-called Concave Least Weight Subsequence (CLWS) problem [6]. The first linear time algorithm for the CLWS problem was given by Wilber [16], but there is no known efficient parallelization. The best known $\mathcal{NC}$ algorithm for the CLWS requires $O(\log^2 n)$ time with $n^2/\log n$ CREW processors [1, 2]. The use of such an algorithm is the bottleneck of our algorithm for the OAT problem, *i.e.*, if the CLWS problem is solvable in polylogarithmic time with n^α processors, $\alpha \ge 1$, then our algorithm can be also implemented to work in polylogarithmic time with those same n^α processors.

The Hu-Tucker (or Garsia-Wachs) algorithm is considerably more complicated than the Huffman algorithm. We can cite Knuth's opinion about the problem of proving it correct (see page 443 in [11]), "No simple proof is known, and it is quite possible that no simple proof will ever be found!" The Hu-Tucker algorithm works in a "mysterious way" and appears to be inherently sequential. One of the minor difficulties is caused by the use of two different types of nodes in the algorithm (square and round nodes). We use the Garsia-Wachs version of the Hu-Tucker algorithm which uses only one type of nodes, and is technically easier to parallelize.

Uniqueness hypothesis. In order to avoid dealing with ties, we will, throughout this paper, assume that all weights that arise during any computation are distinct. It suffices to assume that the item weights $b_1, \ldots b_n$ are linearly independent over the integers. In theory, the uniqueness hypothesis can be forced by adding appropriate infinitesimals to the weight of each item. In practice, the uniqueness hypothesis can be forced without increasing the asymptotic complexity by careful use of a tie-breaking strategy. This technique will be explained in the full paper.

For the rest of this paper, for simplicity of exposition, each item's weight will also be its name. By the uniqueness hypothesis, no confusion will result.

Overview of the algorithm. Our algorithm has the same two phases as the Hu-Tucker and the Garsia-Wachs algorithms. In the first phase, a certain tree called the *ℓ-tree* (short for "level tree") is constructed. This is a binary tree whose leaves are the nodes in the list, and whose internal nodes are what Hu and Tucker call "round nodes," and what are called "packages" in [13]. In the second phase, we find the unique alphabetic tree where each item is at the same level as in the ℓ-tree. This will be the optimal alphabetic tree.

The sequential algorithm. The classical sequential algorithms compute the ℓ-tree as follows. Initialize the list $\beta = (b_1, \ldots b_n)$. After each step, β will be a list of roots of binary trees, and the totality of leaves of these trees will be the set $\{b_1, \ldots b_n\}$

Each step consists of "processing" just one pair. Two items of β are chosen. These two items are deleted from β, and become the left and right sons of the root of a new binary tree, which is then re-inserted into β. The length of β thus decreases by 1 at each step. After $(n-1)$ steps, β consists of a single item, the root of the ℓ-tree.

The Hu-Tucker and Garsia-Wachs algorithms differ in the manner in which they decide which pair to combine and where the combined node is reinserted, but the final ℓ-tree is the same.

In the Garsia-Wachs algorithm, which is more closely related to ours, a pair must be a locally minimum adjacent pair. When combined, the new node moves to the right (or left) as far is it can without passing a node which is heavier. It is then reinserted at this point.

Both the Hu-Tucker and Garsia-Wachs algorithms compute the ℓ-tree in $O(n \log n)$ time. The optimal alphabetic tree is constructed from the ℓ-Tree in $O(n)$ time.

Parallelization. The model of parallel computation we use in this paper is the CREW PRAM. In our parallel algorithm, we make use of the invariance of the ℓ-tree. We process any number of locally minimal adjacent pairs at once, and we also make use of what we call "predestination." If we can determine that a certain pair of adjacent nodes will eventually become locally minimal, we can process them immediately. (See Lemma 4.1.)

The construction of the ℓ-tree is then done in parallel in logarithmically many "top-level" iterations. At each iteration, a non-overlapping set of special subsequences, maximal "1-valleys," is identified. Each maximal 1-valley is processed in two steps. In the first step, certain predestined "special" pairs are identified and processed. The 1-valley is thereby transformed into a "regular valley." In the second step, the regular valley is pro-

PARALLEL CONSTRUCTION OF OPTIMAL ALPHABETIC TREES

Lawrence L. Larmore *
Department of Computer Science,
University of California, Riverside, CA 92521

Teresa M. Przytycka
Department of Mathematics and Computer Science
Odense University, DK-5239 Odense M, Denmark

Wojciech Rytter †
Institute of Informatics, Warsaw University
ul. Banacha 2, 02-097 Warsaw, Poland

Abstract

A parallel algorithm is given which constructs an optimal alphabetic tree in $O(\log^3 n)$ time with $n^2 \log n$ processors. The construction is basically a parallelization of the Garsia-Wachs version [5] of the Hu-tucker algorithm [8]. The best previous $\mathcal{NC}$ algorithm for the problem uses $n^6/\log^{O(1)} n$ processors. [15]

Our method is an extension of techniques used first in [3] and later used in [13] for the Huffman coding problem, which can be viewed as the alphabetic tree problem for the special case of a monotone weight sequence. In this paper, we extend to the case of certain "almost monotone" sequences, which we call "sorted regular valleys." The processing of such subsequences depends on a quadrangle inequality, while the total number of global iterations depends on a kind of tree contraction. Altogether we can view our algorithmic approach as (*quadrangle inequality* + *tree contraction*).

An optimal alphabetic tree is a special case of an optimal binary search tree where all the weights are in the leaves. Thus, the result gives a partial answer to the open problem posed in [3]: is there an $\mathcal{NC}$ algorithm which can find an optimal binary search tree and which uses $n^{6-\epsilon}$ processors for some $\epsilon > 0$?

* This research was supported by NSF grant CCR9112067.

† This work was done while the author was visiting University of California at Riverside. The work of this author was supported by the project *ALTEC*

1 Introduction

The construction of optimal binary search trees (OBST) is an important algorithmic problem which has so far resisted any really efficient $\mathcal{NC}$ implementation, though Knuth gives a quadratic time sequential algorithm [12] which uses a *quadrangle inequality*. The *optimal alphabetic tree* (OAT) problem is an important special case of the OBST problem. The best known sequential time for that problem is $O(n \log n)$ using a greedy approach [5, 8]. In this paper, we use a combination of that greedy approach and a quadrangle inequality, generalizing the results on Huffman coding from [13].

Statement of the optimal alphabetic tree problem. Let $\beta = (B_1, \ldots B_n)$ be a sequence of weighted items, where b_i is the *weight* of B_i. If T is a binary tree with n leaves, where the i^{th} leaf (in left-to-right order) is labeled B_i, we define the *cost* of T as follows:

$$cost(T) = \sum_{i=1}^{n} b_i \ell_i \qquad (1)$$

where ℓ_i is the *level* of B_i in T, defined to be the distance from the root. The OAT problem is then the problem of finding that tree T of minimum cost for a given sequence of items.

Although the restriction of the OBST problem to the OAT problem results in substantial savings in the sequential class [5, 8], no savings at all have been realized