1997

Time-Optimal Tree Computations on Sparse Meshes

D. Bhagavathi

V. Bokka
Old Dominion University

H. Gurla
Old Dominion University

S. Olariu
Old Dominion University

J. L. Schwing
Old Dominion University

Original Publication Citation

Bhagavathi, D., Bokka, V., Gurla, H., Olariu, S., & Schwing, J. L. (1997). Time-optimal tree computations on sparse meshes. *Discrete Applied Mathematics, 77*(3), 201-220. doi:10.1016/s0166-218x(97)00135-2

ELSEVIER

Discrete Applied Mathematics 77 (1997) 201–220

DISCRETE APPLIED MATHEMATICS

Time-optimal tree computations on sparse meshes☆

D. Bhagavathi[a], V. Bokka[b], H. Gurla[b], S. Olariu[b,*], J.L. Schwing[b]

[a] *Department of Computer Science, Southern Illinois University, Edwardsville, IL 62026, USA*
[b] *Department of Computer Science, Old Dominion University, Norfolk, VA 23529, USA*

Received 31 July 1995; revised 8 October 1996

Abstract

The main goal of this work is to fathom the suitability of the mesh with multiple broadcasting architecture (MMB) for some tree-related computations. We view our contribution at two levels: on the one hand, we exhibit time lower bounds for a number of tree-related problems on the MMB. On the other hand, we show that these lower bounds are tight by exhibiting time-optimal tree algorithms on the MMB. Specifically, we show that the task of encoding and/or decoding n-node binary and ordered trees cannot be solved faster than $\Omega(\log n)$ time even if the MMB has an infinite number of processors. We then go on to show that this lower bound is tight. We also show that the task of reconstructing n-node binary trees and ordered trees from their traversals can be performed in O(1) time on the same architecture. Our algorithms rely on novel time-optimal algorithms on sequences of parentheses that we also develop.

Keywords: Meshes with multiple broadcasting; Binary trees; Ordered trees; Encoding; Decoding; Traversals; Tree reconstruction; Parentheses matching; Parallel algorithms; Time-optimal algorithms

1. Introduction

Due to its simple and regular interconnection topology the mesh is particularly well suited for solving various problems in image processing, pattern recognition, graph theory, and computer graphics. At the same time, its large diameter renders the mesh less effective in computing with data spread over processing elements far apart. To overcome this problem, the mesh architecture has been enhanced by various types of bus systems. Early solutions involving the addition of one or more global buses, shared by all processors, have been implemented on a number of massively parallel machines. Recently, a more powerful architecture, referred to as mesh with multiple broadcasting,

☆ Work supported by NASA grant NAS1-19858, by NSF grant CCR-9522093, and by ONR grant N00014-95-1-0779;

*Corresponding author. E-mail: olariu@cs.odu.edu.
Correspondence to: Prof. S. Olariu

PII S0166-218X(97)00135-7

(MMB) has been obtained by adding one bus to every row and to every column of the mesh [7, 14]. The MMB has been implemented in VLSI and is used in the DAP family of computers [14].

An MMB of size $M \times N$ consists of MN synchronous processors positioned on a rectangular array overlaid with a bus system. The processors are connected to their nearest neighbors and are assumed to know their own coordinates within the mesh. In addition, in every row of the mesh the processors are connected to a horizontal bus; similarly, in every column the processors are connected to a vertical bus as illustrated in Fig. 1.

Processor $P(i,j)$ is located in row i and column $j, (1 \leqslant i \leqslant M; 1 \leqslant j \leqslant N)$, with $P(1,1)$ in the north–west corner of the mesh. Every processor has a constant number of registers of size $O(\log MN)$; in unit time, the processors perform an arithmetic or boolean operation, communicate with one of their neighbors using a local connection, broadcast a value on a bus, or read a value from a specified bus. All these operations involve handling at most $O(\log MN)$ bits of information. For practical reasons, only one processor is allowed to broadcast on a given bus at any one time. By contrast, all the processors on the bus can simultaneously read the value being broadcast. In accord with other researchers [7, 10, 14], we assume that communications along buses take $O(1)$ time. Although inexact, recent experiments with the DAP and the YUPPIE multiprocessor array system, seem to indicate that this is a reasonable working hypothesis [10, 14]. Being of theoretical interest as well as commercially available, the MMB has attracted a great deal of attention [1–3, 7, 8, 12, 14]

A PRAM [5] consists of synchronous processors, all having unit-time access to a shared memory. In the CREW-PRAM, a memory location can be simultaneously accessed in reading but not in writing. From a theoretical point of view, an MMB can be perceived as a restricted version of the CREW-PRAM machine: the buses are nothing more than *oblivious* concurrent read-exclusive write registers with the access restricted to certain sets of processors. Indeed, in the presence of p

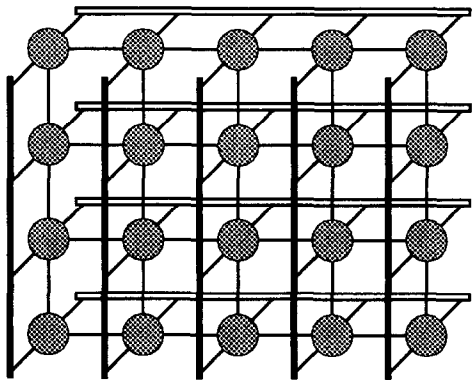


Fig. 1. A mesh with multiple broadcasting of size 4×5.

CREW-PRAM processors, groups of $\sqrt{p}$ of these have concurrent read access to a register whose value is available for one time unit, after which it is lost. Given that the MMB is, in this sense, *weaker* than the CREW-PRAM, it is very often quite a challenge to design algorithms in this model that match the performance of their CREW-PRAM counterparts. Typically, for the same running time, the MMB uses more processors.

Encoding the shape of an ordered tree is a basic step in a number of algorithms in integrated circuit design, automated theorem proving, and game playing [15]. The common characteristic of these applications is that the information stored at the nodes is irrelevant, as one is only interested in detecting whether two ordered trees have the same "shape". As it turns out, if we ignore the contents of the nodes of an n-node tree T, then the remaining shape information can be uniquely captured by a string of $2n$ bits, referred to as the *encoding* of T [12, 15]. Conversely, given a string of $2n$ bits, a number of practical applications ask to recover the unique n-node ordered tree (if any) corresponding to this encoding.

The main goal of this work is to fathom the suitability of the MMB architecture for some tree-related computations. Our contribution is to show tight time lower bounds and to provide time-optimal tree algorithms on the MMB architecture. Specifically, we show that the following tasks can be solved in $\Theta(\log n)$ time on an MMB of size $n \times n$:

- Encode an n-node binary tree into a $2n$-bitstring;
- Encode an n-node ordered tree into a $2n$-bitstring;
- Recover an n-node binary tree from its $2n$-bit encoding;
- Recover an n-node ordered tree from its $2n$-bit encoding.

We also show that the following tasks can be performed in $O(1)$ time:

- Reconstruct an n-node binary tree from its preorder and inorder traversals;
- Reconstruct an n-node ordered tree (forest) from its preorder and postorder traversals.

Our algorithms rely on novel time-optimal algorithms involving sequences of parentheses, that we also develop. Specifically, we show that each of the tasks can be solved in $\Theta(\log n)$ time:

- finding all the matching pairs in a well-formed sequence of parentheses;
- determining the closest enclosing pair for every matching pair in a well-formed sequence.

The remainder of the paper is organized as follows. Section 2 presents our lower bound arguments. Section 3 discusses a number of fundamental results that are instrumental in our algorithms. Section 4 discusses the details of our parentheses algorithms. Section 5 presents time-optimal algorithms for encoding and decoding binary and ordered trees. Section 6 addresses the problem or reconstructing binary and ordered trees from their traversals. Finally, Section 7 offers concluding remarks and open problems.

2. Lower bounds

The purpose of this section is to provide time lower bounds for a number of fundamental problems that establish the optimality of our algorithms. Our lower bounds will be stated first in the CREW-PRAM and then extended to the MMB using a recent result of Lin et al. [9]. All our arguments for the CREW-PRAM rely directly or indirectly on the following fundamental result of Cook et al. [4].

Proposition 2.1 (Cook et al. [4]). *The task of computing the logical OR of n bits has a time lower bound of* $\Omega(\log n)$ *on the* CREW-PRAM, *regardless of the number of processors and memory cells used.*

LEFTMOST ONE: Given a sequence of n bits, find the position of the leftmost 1 bit in the sequence.

It is a trivial observation that OR reduces to LEFTMOST ONE in $O(1)$ time. Therefore, Proposition 2.1 implies the following result.

Corollary 2.2. *LEFTMOST ONE has a time lower bound of* $\Omega(\log n)$ *on the* CREW-PRAM, *regardless of the number of processors and memory cells used.*

To obtain lower bounds for the problems of interest to us on the MMB, we rely on the following recent result of Lin et al. [9].

Proposition 2.3 (Lin et al. [9]). *Any computation that takes* $O(t(n))$ *computational steps on an n-processor MMB can be performed in* $O(t(n))$ *computational steps on an n-processor* CREW-PRAM *with* $O(n)$ *extra memory.*

It is important to note that Proposition 2.3 guarantees that if $T_M(n)$ is the execution time of an algorithm for solving a given problem on an n-processor MMB, then there exists a CREW-PRAM algorithm to solve the same problem in $T_P(n) = T_M(n)$ time using n processors and $O(n)$ extra memory. In other words, "too fast" an algorithm on the MMB implies "too fast" an algorithm for the CREW-PRAM.

In the remaining part of this section the general scheme for proving lower bounds is as follows; given a problem A we either reduce the OR problem or the LEFTMOST ONE problem to A. Using the input to the OR, which is a sequence of bits $b_1, b_2, \ldots b_n$, an input to A is constructed, generally the processor which holds the bit b_i, in parallel, generates a portion of the input for A thus taking $O(1)$ time for the reduction process. After that there is an argument which relates the output of A to the result of OR, i.e. depending on the output of A we can determine in $O(1)$ time the output for OR. This shows that the lower bound for A is $\log n$.

A sequence $\sigma = x_1 x_2 \ldots x_n$ of parentheses is said to be *well-formed* if it contains the same number of left and right parentheses and in every prefix of σ the number of right

parentheses does not exceed the number of left parentheses. Next we define the classic parentheses matching problem.

MATCHING: Given a well-formed sequence $\sigma = x_1 x_2 \ldots x_n$ of parentheses, for each parenthesis in σ, find its match.

Lemma 2.4. *MATCHING has a time lower bound of* $\Omega(\log n)$ *on the* CREW-PRAM, *regardless of the number of processors and memory cells used.*

Proof. We reduce OR to MATCHING. For this purpose, let the input to OR consist of n bits $b_1, b_2, \ldots, b_n$. We convert this input to a sequence $c_0 c_1 c_2 \ldots c_{2n+1}$ of parentheses by writing $c_0 =$ '(' and $c_{2n+1} =$ ')', and by setting for all j $(1 \leqslant j \leqslant n)$:

- $c_{2j-1} =$ '(' and $c_{2j} =$ ')', whenever $b_j = 0$;
- $c_{2j-1} =$ ')' and $c_{2j} =$ '(', whenever $b_j = 1$.

It is easy to see that this construction can be performed in O(1) time by n CREW processors. Now, an easy inductive argument on the number of 1's in $b_1, b_2, \ldots, b_n$ shows that the sequence $c_0 c_1 c_2 \ldots c_{2n+1}$ is always well-formed. Furthermore, the matching pair of c_0 is c_{2n+1} if and only if the answer to the OR problem is 0. The conclusion follows by Proposition 2.1. □

One is often interested in the solution of the following problem.

ENCLOSING PAIR: Given a well-formed sequence $\sigma = x_1 x_2 \ldots x_n$ of parentheses, for every matching pair of parentheses in σ, find the closest pair of parentheses that encloses it.

Lemma 2.5. *ENCLOSING PAIR has a time lower bound of* $\Omega(\log n)$ *on the CREW-PRAM, regardless of the number of processors and memory cells used.*

Proof. We reduce LEFTMOST ONE to ENCLOSING PAIR. Assume that the input to LEFTMOST ONE is $b_1, b_2, \ldots, b_n$. Construct a sequence $c_1 c_2 \ldots c_{4n+2}$ of parentheses as follows:

- $c_{2n+1} =$ '('; $c_{2n+2} =$ ')'; furthermore, for all j $(1 \leqslant j \leqslant n)$ set.
- $c_{2n-2j+1} =$ '('; $c_{2n-2j+2} =$ ')'; $c_{2n+2j+1} =$ '('; $c_{2n+2j+2} =$ ')', whenever $b_j = 0$;
- $c_{2n-2j+1} =$ '('; $c_{2n-2j+2} =$ '('; $c_{2n+2j+1} =$ ')'; $c_{2n+2j+2} =$ ')', whenever $b_j = 1$.

It is easy to see that this construction can be performed in O(1) time by n CREW processors. Moreover, our construction guarantees that the sequence is well-formed and that every parenthesis knows its match; in particular, c_{2n+1} and c_{2n+2} are a matching pair. Furthermore, the closest enclosing parentheses for the pair (c_{2n+1}, c_{2n+2}) is $(c_{2n-2k+2}, c_{2n+2k+1})$, if and only if k is the position of the leftmost 1 in $b_1, b_2, \ldots, b_n$. Now the conclusion follows by Corollary 2.2. □

A binary tree T is either empty or consists of a root and two disjoint binary trees, called the left subtree, T_L and the right subtree, T_R. For later reference, we assume that every node in a binary tree maintains pointers to its left and right children. In many

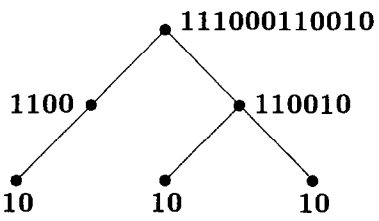


Fig. 2. A binary tree and its encoding.

contexts, it is desirable to encode the shape of T as succinctly as possible. In this paper, we are interested in one such encoding scheme recursively defined as follows:[1]

$$\sigma(T) = \begin{cases} \varepsilon & \text{if } T \text{ is empty,} \\ 1\sigma(T_L)0\sigma(T_R) & \text{otherwise.} \end{cases} \quad (1)$$

Note that under (1) an arbitrary n-node binary tree is encoded into $2n$ bits, as illustrated in Fig. 2.

BINARY TREE ENCODING: Given an n-node binary tree, find its encoding.

Lemma 2.6. *BINARY TREE ENCODING has a time lower bound of* $\Omega(\log n)$ *on the* CREW-PRAM, *regardless of the number of processors and memory cells used.*

Proof. We reduce OR to BINARY TREE ENCODING. Assume that $b_1, b_2, \ldots, b_n$ is an arbitrary input to OR. We assume an extra bit b_{n+1} with value 0 (so as not to change the answer). Convert this bit sequence to an $n+1$-node binary tree T with nodes $1, 2, 3, \ldots, n+1$. Specifically, we associate with every bit $b_j, (1 \leqslant j \leqslant n+1)$ the node j of T, such that

- 1 is the root of T;
- for every $i, (1 \leqslant i \leqslant n)$, node $i+1$ is the unique child of i. Moreover, $i+1$ is the left child of i if $b_i = 1$ and the right child otherwise. An illustration of the construction if featured in Fig. 3.

It is easy to see that this construction can be performed in O(1) time by n CREW processors. To see this notice that each processor $P(1, i)$ holding the bit b_i creates the node i of T and sets its child (left or the right child depends on the value of b_j) pointer to node $i+1$ which is stored in processor $P(1, i+1)$. Let $\sigma(T) = c_1 c_2 \ldots c_{2n+2}$ be the $2n+2$-bit encoding of T. We claim that

$$c_{2n+1} = 1 \text{ if and only if the answer to OR is 0.} \quad (2)$$

To justify (2), observe that if all the bits in the string $b_1, b_2, \ldots, b_{n-1}$ are 0 then, by our construction and (1), combined, $c_{2n+1} = 1$. Conversely, let j be the position of the leftmost 1 in $b_1, b_2, \ldots, b_{n-1}$. Clearly, (1) guarantees that the encoding of the subtree rooted at j is $1\sigma(T_{j+1})0$ and, since by construction node j has no right child, this is

[1] This scheme is similar to the one reported in [15].

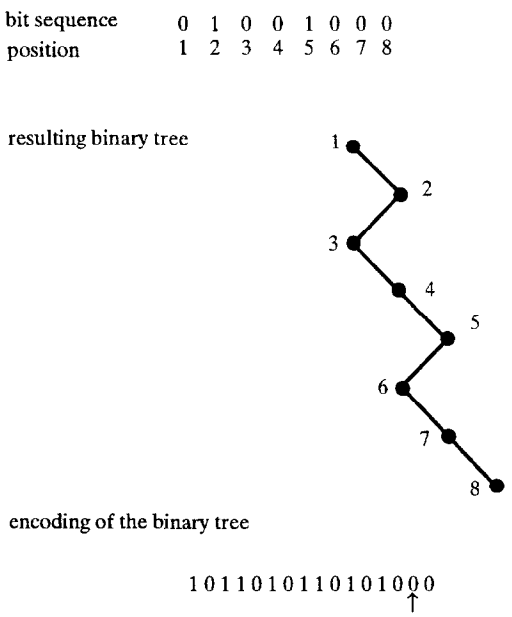


Fig. 3. Illustrating the construction in Lemma 2.6.

a suffix of the encoding of T. Since $\sigma(T_{j+1})$ must end in a 0, it follows that $c_{2n+1}=0$ and the conclusion follows.

By virtue of (2), once the encoding is available, one can determine in O(1) time the answer to OR. Now the conclusion follows by Proposition 2.1. □

The converse operation of recovering a binary tree from its encoding is of interest in a number of practical applications. We state the problem as follows.

BINARY TREE DECODING: Recover an n-node binary tree from its encoding.

Lemma 2.7. *BINARY TREE DECODING has a time lower bound of* $\Omega(\log n)$ *on the* CREW-PRAM, *regardless of the number of processors and memory cells used.*

Proof. We reduce OR to BINARY TREE DECODING. Let $b_1, b_2, \ldots, b_n$ be an arbitrary input to OR. We add a new bit b_0 with a value 0 (this will not change the OR of the bits). Construct a well-formed sequence of parentheses, $c_0 c_1 \ldots c_{2n+3}$ as described below:

- $c_0 =$ '('; $c_{2n+3} =$ ')'.
- $c_{2i+1} =$ '('; $c_{2i+2} =$ ')', whenever $b_i = 0$;
- $c_{2i+1} =$ ')'and $c_{2i+2} =$ '(', whenever $b_i = 1$ and $b_{i-1} = 0$;
- $c_{2i-1} =$ '('and $c_{2i} =$ ')', whenever $b_i = 1$ and $b_{i-1} = 1$.

It is easy to see that this construction can be performed in O(1) time by n CREW processors, and that the resulting sequence is well-formed. Consequently, interpreting

every '(' as a 1 and every ')' as a 0, $c_0c_1 \ldots c_{2n+3}$ is the encoding of a binary tree T with $n+2$ nodes. Notice that root(T) has two children if and only if the OR of the input sequence is 1. Therefore, once the decoding is available, one can solve the OR problem in O(1) time. Now Proposition 2.1 implies that any algorithm that performs the decoding must take $\Omega(\log n)$ time. □

An ordered tree T is either empty or it contains a root and disjoint ordered subtrees, $T_1, T_2, \ldots, T_k$. For later reference, we assume that ordered trees are specified by parent pointers. The encoding $\sigma(T)$ of T is defined as follows:

$$\sigma(T) = \begin{cases} \varepsilon & \text{if } T \text{ is empty} \\ 1\sigma(T_1)\sigma(T_2)\ldots\sigma(T_k)0 & \text{otherwise.} \end{cases} \tag{3}$$

Note that the encoding of an n-node ordered tree is a sequence of $2n$ bits. Refer to Fig. 4 for an example.

Next, we are interested in the following problem.

ORDERED TREE ENCODING: Given an ordered tree, find its encoding.

Lemma 2.8. *ORDERED TREE ENCODING has a time lower bound of* $\Omega(\log n)$ *on the* CREW-PRAM, *regardless of the number of processors used.*

Proof. We reduce OR to ORDERED TREE ENCODING. Let $b_1, b_2, \ldots, b_n$ be an arbitrary input to OR. We add two bits $b_0 = 1$ and $b_{n+1} = 0$. The new sequence $b_0, b_1, \ldots, b_{n+1}$ is converted to an ordered tree T on nodes $\{0, 1, \ldots, n+1\}$ as follows:

- node 0 is the root;
- for all i, $(1 \leqslant i \leqslant n+1)$, the parent of node i is 0 if $b_i = 0$ and $n+1$ otherwise.

The reader is refered to Fig. 5 for an illustration of this construction.

It is easy to see that this construction can be performed in O(1) time by n CREW processors. Let $c_1c_2 \ldots c_{2n+4}$ be the $2(n+2)$-bit encoding of T. Observe that

$$c_{2n+2} = 1 \text{ if and only if the OR of the input sequence is 0,} \tag{4}$$

To justify (4), note that if all bits in $b_1, b_2, \ldots, b_n$ are 0 then, by (3) and our construction, $c_{2n+2} = 1$. Conversely, if there exist 1's in the sequence $b_1, b_2, \ldots, b_n$, then the node of T corresponding to b_{n+1} is not a leaf. Therefore, (3) guarantees that $c_{2n+2} = c_{2n+3} = c_{2n+4} = 0$ and (4) follows.

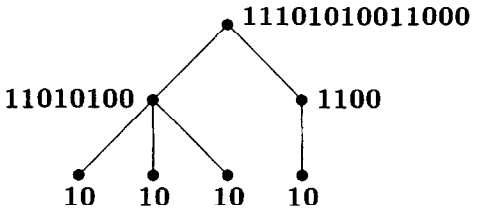


Fig. 4. An ordered tree and its encoding.

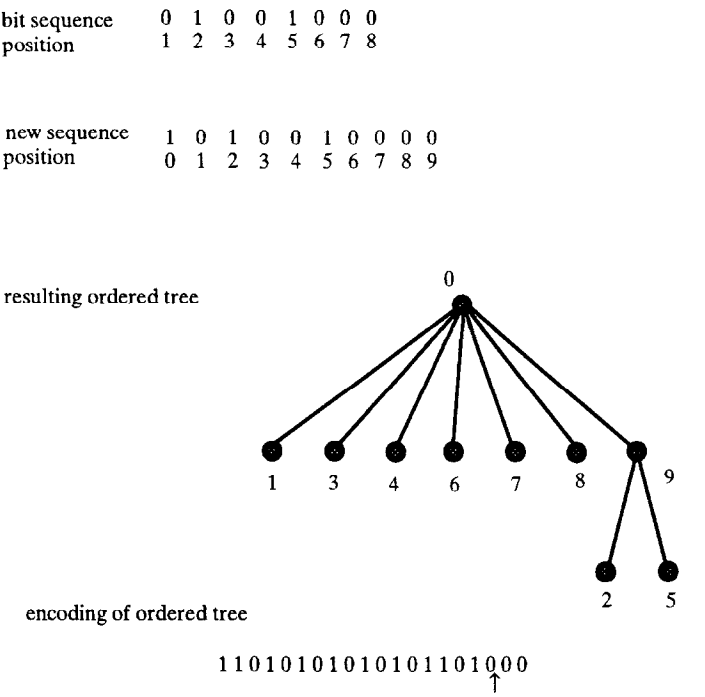


Fig. 5. Illustrating the construction in Lemma 2.8.

Now (4) guarantees that the answer to OR can be obtained in O(1) time, once the encoding is available. Therefore, by Proposition 2.1, the encoding algorithm must take $\Omega(\log n)$ time. □

The converse operation requires recovering an ordered tree from its encoding. Specifically, the tree is assumed specified in parent pointer representation. We state the problem as follows.

ORDERED TREE DECODING: Recover an ordered tree from its encoding.

Lemma 2.9. *ORDERED TREE DECODING has a time lower bound of* $\Omega(\log n)$ *on the* CREW-PRAM, *regardless of the number of processors used.*

Proof. We reduce ENCLOSING PAIR to ORDERED TREE DECODING. Let the input to ENCLOSING PAIR be $s_1 s_2 \ldots s_{2n}$. Augment this sequence with $s_0 =$ '(' and $s_{2n+1} =$ ')'. Thus, interpreting every '(' as a 1 and every ')' as a 0 we obtain the valid encoding of some ordered tree T under (2). Now, consider any algorithm that correctly recovers T from the encoding above. It is easy to see that the setting of parent pointers gives exactly the solution to the ENCLOSING PAIR problem for the augmented sequence. The conclusion follows from Lemma 2.5. □

Now Proposition 2.3 together with Lemmas 2.4–2.9 imply the following result.

Theorem 2.10. *MATCHING, ENCLOSING PAIR, BINARY TREE ENCODING, BINARY TREE DECODING, ORDERED TREE ENCODING, and ORDERED TREE DECODING have a lower bound of* $\Omega(\log n)$ *on an MMB of size* $n \times n$.

3. Basics

The purpose of this section is to review a number of fundamental results for the MMB that will be instrumental in the design of our algorithms.
The problem of list ranking is to determine the rank of every element in a given list, stored as an unordered array, that is, the number of elements following it in the list. Recently, Olariu et al. [12] have proposed a time-optimal algorithm for list ranking on MMB's.

Proposition 3.1 (Olariu et al. [12]). *The task of ranking an n-element linked list stored in one row of an MMB of size* $n \times n$ *can be performed in* $O(\log n)$ *time. Furthermore, this is time optimal.*

The All Nearest Smaller Values problem (ANSV) is formulated as follows: given a sequence of n real numbers $a_1, a_2, \ldots, a_n$, for each a_i $(1 \leqslant i \leqslant n)$, find the nearest element to its left and the nearest element to its right. Recently, Olariu et al. [12] have proposed a time-optimal algorithm for the ANSV problem.

Proposition 3.2 (Olariu et al. [11]). *Any instance of size n of the ANSV problem can be solved in* $O(\log n)$ *time on an MMB of size* $n \times n$. *Furthermore, this is time-optimal.*

The *prefix sums* problem is a key ingredient in many parallel algorithms and is stated as follows: given a sequence $a_1, a_2, \ldots, a_n$ of items, compute all the sums of the form $a_1, a_1 + a_2, a_1 + a_2 + a_3, \ldots, a_1 + a_2 + \cdots + a_n$.

Proposition 3.3 (Kumar and Raghavendra [7], and Olariu et al. [12]). *The prefix sums (also maxima or minima) of a sequence of n real numbers stored in one row of an MMB of size* $n \times n$ *can be computed in* $O(\log n)$ *time. Furthermore, this is time-optimal.*

Merging two sorted sequences is one of the fundamental operations in computer science. Recently, Olariu et al. [12] have proposed a constant time algorithm to merge two sorted sequences of total length n stored in one row of an MMB of size $n \times n$.

Proposition 3.4 (Olariu et al. [12]). *Let* $S_1 = (a_1, a_2, \ldots, a_r)$ *and* $S_2 = (b_1, b_2, \ldots, b_s)$, *with* $r + s = n$, *be sorted sequences stored in one row of an MMB of size* $n \times n$,

with $P(1,i)$ holding a_i, $(1 \leqslant i \leqslant r)$ and $P(1,r+i)$ holding b_i, $(1 \leqslant i \leqslant s)$. S_1 and S_2 can be merged in $O(1)$ *time.*

Recently, the simple merging algorithm of Proposition 3.4 was used to derive a time-optimal sorting algorithm for MMBs [12].

Proposition 3.5 (Olariu et al. [12]). *An n-element sequence of items from a totally ordered universe stored one item per processor in the first row of an MMB of size $n \times n$ can be sorted in* $O(\log n)$ *time. Furthermore, this is time-optimal.*

4. Time-optimal parentheses algorithms

The purpose of this section is to present two time-optimal algorithms involving sequences of parentheses on an MMB of size $n \times n$. In addition to being interesting in their own right, these algorithms are instrumental in our subsequent tree algorithms.

Consider a sequence $\sigma = x_1 x_2 \ldots x_n$ of parentheses stored one item per processor in the first row of an MMB of size $n \times n$, with x_k, $(1 \leqslant k \leqslant n)$, stored by $P(1,k)$. Assuming that the sequence is well-formed, we present an algorithm to find all the matching pairs. The idea is as follows. First, we compute a sequence $w_1, w_2, \ldots, w_n$ obtained from σ by setting $w_1 = 0$ and by defining w_k, $(2 \leqslant k \leqslant n)$, as follows:

$$w_k = \begin{cases} 1 & \text{if both } x_{k-1} \text{ and } x_k \text{ are left parentheses;} \\ -1 & \text{if both } x_{k-1} \text{ and } x_k \text{ are right parentheses;} \\ 0 & \text{otherwise.} \end{cases}$$

We now compute the prefix sums of $w_1, w_2, \ldots, w_n$ and let the result be $e_1, e_2, \ldots, e_n$. By Proposition 3.3, this operation is performed in $O(\log n)$ time. It is easy to see that left and right parentheses x_i and x_j are a matching pair if and only if x_j is the first right parenthesis to the right of x_i for which $e_i = e_j$.

Further, with each parenthesis x_k, $(1 \leqslant k \leqslant n)$, we associate the tuple (e_k, k). On the set of these tuples we define a binary relation $\prec$ by setting

$$(e_i, i) \prec (e_j, j) \quad \text{whenever} \quad (e_i < e_j) \quad \text{or} \quad [(e_i = e_j) \quad \text{and} \quad (i < j)].$$

It is an easy exercise to show that $\prec$ is a linear order. Now, sort the sequence $(e_1, 1), \ldots, (e_n, n)$ in increasing order of $\prec$. By Proposition 3.5, sorting the ordered pairs can be done in $O(\log n)$ time. The key observation is that, as a result of sorting, the matching pairs occur next to one another. For a worked example the reader is referred to Fig. 6. To summarize our findings we state the following result.

Theorem 4.1. *Given a well-formed sequence of n parentheses as input, all matching pairs can be found in* $O(\log n)$ *time on an MMB of size $n \times n$. Furthermore, this is time-optimal.* □

```
(  (  (  )  (  )  )  (  )  )  (  )
1  2  3  4  5  6  7  8  9  10 11 12
```

the w sequence:

```
0  1  1  0  0  0  −1 0  0  −1 0  0
```

the prefix sums:

```
0  1  2  2  2  2  1  1  1  0  0  0
```

the tuple sequence:

(0,1) (1,2) (2,3) (2,4) (2,5) (2,6) (1,7) (1,8) (1,9) (0,10) (0,11) (0,12)

same sequence sorted :

(0,1) (0,10) (0,11) (0,12) (1,2) (1,7) (1,8) (1,9) (2,3) (2,4) (2,5) (2,6)

the matching pairs:

1 − 10, 11 − 12, 2 − 7, 8 − 9, 3 − 4, 5 − 6

Fig. 6. Illustrating the parentheses matching algorithm.

```
(  (  (  )  (  )  )  (  )  )  (  )
1  2  3  4  5  6  7  8  9  10 11 12
```

the matching pairs:

1 − 10, 11 − 12, 2 − 7, 8 − 9, 3 − 4, 5 − 6

the p sequence:

```
10 7  4  3  6  5  2  9  8  1  12 11
```

the solution to the ANLV problem:

```
0  1  3  −  2  −  −  1  −  −  0  −
```

Fig. 7. Illustrating the solution to the ENCLOSING PAIR problem.

Next, we are interested in a time-optimal solution to the ENCLOSING PAIR problem stated in Section 2. Consider a well-formed sequence $\sigma = x_1 x_2 \ldots x_n$ of parentheses, stored one item per processor in the first row of the mesh. The details of the algorithm follow. For a worked example the reader is referred to Fig. 7

Step 1. Find the match of every parenthesis in σ; every processor $P(1,i)$ stores in a local variable the position j of the match x_j of x_i.

Step 2. Solve the corresponding instance of the ANSV problem.

It is not hard to see that at the end of Step 2 every processor knows the identity of the closest enclosing pair. By Proposition 3.2 and Theorem 4.1, the running time of this simple algorithm is bounded by $O(\log n)$. By Theorem 2.10, this is the best possible on this architecture. Thus, we have proved the following result.

Theorem 4.2. *Given a well-formed sequence of n parentheses stored one item per processor in the first row of an MMB of size $n \times n$, the ENCLOSING PAIR problem can be solved in $O(\log n)$ time. Furthermore, this is time-optimal.*

5. Encoding and decoding trees

The purpose of this section is to show that the task of encoding n-node binary and ordered trees into a $2n$-bitstring can be carried out in $O(\log n)$ time on an MMB of size $n \times n$. By virtue of Theorem 2.10, this is time-optimal.

Consider an n-node binary tree T with left and right subtrees T_L and T_R, respectively. We assume that the nodes of T are stored, one item per processor, in the first row of an MMB of size $n \times n$. First, we show how to associate with T the unique encoding $\sigma(T)$ defined in (1). Our encoding algorithm can be seen as a variant of the classic Euler-tour technique [5]. We proceed as follows. Replace every node u of T by 3 copies, u^1, u^2, and u^3. If u has no left child, then set $\text{link}(u^1) \leftarrow u^2$, else if v is the left child of u, set $\text{link}(u^1) \leftarrow v^1$ and $\text{link}(v^3) \leftarrow u^2$. Similarly, if u has no right child, then set $\text{link}(u^2) \leftarrow u^3$ else if w is the right child of u then set $\text{link}(u^2) \leftarrow w^1$ and $\text{link}(w^3) \leftarrow u^3$. It is worth noting that the processor associated with node u can perform the pointer assignments in $O(1)$ time. What results is a linked list starting at $root(T)^1$ and ending at $root(T)^3$, with every edge of T traversed exactly once in each direction. It is easy to confirm that the total length of the linked list is $O(n)$. Finally, assign to every u^1 a 1, to every u^2 a 0 and delete all elements of the form u^3. It is now an easy matter to show that what remains represents the encoding of T specified in (1).

The correctness of this simple algorithm being easy to see, we turn to the complexity. Computing the Euler tour amounts to setting pointers. Since all the information is available locally, this step takes $O(1)$ time. The task of eliminating every node of the form u^3 can be reduced to list ranking, prefix computation, and compaction in the obvious way. By virtue of Propositions 3.1 and 3.3 these tasks can be performed in $O(\log n)$ time. By Theorem 2.10, this is the best possible. Consequently, we have the following result.

Theorem 5.1. *The task of encoding an n-node binary tree can be performed in $O(\log n)$ time on an MMB of size $n \times n$. Furthermore, this is time-optimal.*

It is worth noting here that the encoding algorithm described above is quite general and can be used for other purposes as well. For example, the *preorder-inorder* traversal of T is obtained by replacing for every node u of T, u^1 and u^2 by the label of u (see [13] for details). We will further discuss properties of the preorder-inorder traversal in the context of reconstructing binary trees from their preorder and inorder traversals in Section 6.

Our encoding algorithm for ordered trees is very similar to the one described for binary trees. Consider an n-node ordered tree T. It is well-known [11] that for the

purpose of getting the encoding (3) of T we only need to convert T into a binary tree BT as in [6] and then to encode BT using (1). It is easy to confirm that the resulting encoding is exactly the one defined in (3). The conversion of T into BT can be performed in O(1) time since it amounts to resetting pointers only. By Theorem 5.1, the encoding of BT takes $O(\log n)$ time. By Theorem 2.10 this is the best possible. Consequently, we have the following result.

Theorem 5.2. *The task of encoding an n-node ordered tree can be performed in* $O(\log n)$ *time on an MMB of size* $n \times n$. *Furthermore, this is time-optimal.*

Before addressing the task of recovering binary and ordered trees from their encodings, we introduce some notation and review a few technical results. Let T be a binary tree and let v be a node of T. We let T^v stand for the subtree of T rooted at v. A bitstring τ is termed *feasible* if it contains the same number of 0's and 1's and in every prefix the number of 0's does not exceed the number of 1's. Recently, Olariu et al. [11] have shown that every feasible bitstring is the encoding of some binary tree. For later reference, we state the following technical result [11].

Proposition 5.3. *A nonempty bitstring* τ *is feasible if and only if for every* 1 *in* τ *there is a unique matching* 0 *such that* τ *can be written as* $\tau_1 1 \tau_2 0 \tau_3$, *with both* τ_2 *and* $\tau_1\tau_3$ *feasible.*

Proposition 5.3 motivates us to associate with every 1 and its matching 0, a node v in T. The following simple observation [11] will justify our decoding procedure.

Observation 5.4. The corresponding decomposition of τ as $\tau_1 1 \tau_2 0 \tau_3$ has the property that $\sigma(T_L^v) = \tau_2$, and $\sigma(T_R^v)$ is a prefix of τ_3.

Observation 5.4 motivates our algorithm for recovering a binary tree from its encoding. Let τ be a feasible bitstring. For every 1 in τ we find the unique matching 0 guaranteed by Proposition 5.3. The corresponding (1,0) pair is associated with a node v in the binary tree T corresponding to τ. We then compute the left and right children of v. The details of the algorithm are spelled out as follows. Begin by ranking the 1's of τ and use the ranks as indices in T. For every 1, find its unique matching 0. Let v_i be the node of T corresponding to the 1 of rank i and to its matching 0; let p_i and q_i denote the positions in τ of the 1 of rank i and that of its matching 0, respectively. The processor in charge of v_i sets pointers as follows:

- $\text{left}(v_i) \leftarrow$ **nil** in case $q_i = p_i + 1$, and $\text{left}(v_i) \leftarrow v_{i+1}$ otherwise;
- $\text{right}(v_i) \leftarrow v_j$ if $p_j = q_i + 1$, and **nil** otherwise.

The correctness follows immediately from Proposition 5.3 and Observation 5.4. Therefore, we turn to the complexity. Note that to rank all the 1's we need to compute their prefix sum. By Proposition 3.3, this task can be performed in $O(\log n)$ time.

By Theorem 4.2, the matching takes $O(\log n)$ time. Finally, the setting of pointers can be done in $O(1)$ time. Thus, we have the following result.

Theorem 5.5. *The task of recovering an n-node binary tree from its encoding takes* $O(\log n)$ *time on an MMB of size* $n \times n$. *Furthermore, this is time-optimal.*

The task of recovering an n-node ordered tree T from its $2n$-bit encoding is similar. We begin by perceiving the encoding of T as the encoding of a binary tree BT. Once, this tree has been recovered as we just described, we proceed to convert BT to T using the classic ordered-to-binary conversion [6]. As it turns out, this latter task can be carried out in $O(\log n)$ time using the sorting algorithm of Proposition 3.5. Due to space limitations the details are omitted.

Theorem 5.6. *The task of recovering an n-node ordered tree from its* $2n$*-bit encoding can be performed in* $O(\log n)$ *time on an MMB of size* $n \times n$. *Furthermore, this is time-optimal.*

6. Reconstructing trees from their traversals

The purpose of this section is to present $O(1)$ time algorithms for reconstructing binary and ordered trees from their traversals. It is well-known that a binary tree can be reconstructed from its inorder traversal along with either its preorder or its postorder traversal [6]. Our goal is to show that this task can be performed in $O(1)$ time on the MMB. The main idea of our algorithm is borrowed from Olariu et al. [13], where the reconstruction process was reduced to that of merging two sorted sequences.

Let T be an n-node binary tree. For simplicity, we assume that the nodes of T are $\{1,2,\ldots,n\}$. Let $c_1,c_2,\ldots,c_n$ and $d_1,d_2,\ldots,d_n$ be the preorder and inorder traversals of T, respectively. We may think of $c_1,c_2,\ldots,c_n$ as $1,2,\ldots,n$, the case where $c_1,c_2,\ldots,c_n$ is a permutation of $1,2,\ldots,n$ reducing easily to this case [13]. In preparation for merging, we construct two sequences of triples. The first sequence is $(1,j_1,c_1),(1,j_2,c_2),\ldots,(1,j_n,c_n)$ such that $d_{j_i}=c_i$, $(i=1,2,\ldots,n)$. In other words, the second coordinate j_i of a generic triple represents the position of c_i in the inorder sequence $d_1,d_2,\ldots,d_n$. The second sequence consists of the triples $(2,1,d_1),(2,2,d_2),\ldots,(2,n,d_n)$. Denote by $\prod$ the set of triples

$$\{(1,j_1,c_1),(1,j_2,c_2),\ldots,(1,j_n,c_n),(2,1,d_1),(2,2,d_2),\ldots,(2,n,d_n)\},$$

and define a binary relation $\prec$ on $\prod$ as follows: for arbitrary triples (α,β,γ) and (α',β',γ') in $\prod$ we have:

Rule 1. $((\alpha=1)\wedge(\alpha'=1))\rightarrow(((\alpha,\beta,\gamma)\prec(\alpha',\beta',\gamma'))\leftrightarrow(\gamma<\gamma'))$;

Rule 2. $((\alpha=2)\wedge(\alpha'=2))\rightarrow(((\alpha,\beta,\gamma)\prec(\alpha',\beta',\gamma'))\leftrightarrow(\beta<\beta'))$;

Rule 3. $((\alpha=1)\wedge(\alpha'=2))\rightarrow(((\alpha,\beta,\gamma)\prec(\alpha',\beta',\gamma'))\leftrightarrow((\beta\leqslant\beta')\vee(\gamma\leqslant\gamma')))$.

In view of the rather forbidding aspect of Rules 1–3, an explanation is in order. First, note that Rules 1 and 2 confirm that with respect to the relation $\prec$ both sequences $(1,j_1,c_1),(1,j_2,c_2),\ldots,(1,j_n,c_n)$ and $(2,1,d_1),(2,2,d_2),\ldots,(2,n,d_n)$ are sorted. Intuitively, Rule 3 specifies that in the preorder-inorder traversal any pair of distinct labels u and v must occur in the order '$\ldots u\ldots v\ldots v\ldots u\ldots$' or '$\ldots u\ldots u\ldots v\ldots v\ldots$' [13].

Consider the sequence $e_1,e_2,\ldots,e_{2n}$ obtained by extracting the third coordinate of the triples in the sequence resulting from merging the two sequences above. As argued in [13], the sequence $e_1,e_2,\ldots,e_{2n}$ is the preorder-inorder traversal of T.

Let $c_1=1,c_2=2,\ldots,c_n=n$ and $d_1,d_2,\ldots,d_n$ be the preorder and inorder traversals of a binary tree. We assume that these sequences are stored in the first row of an MMB of size $n\times 2n$ in left to right order, with the c_i's stored to the left of the d_i's. It is easy to modify the algorithm to work on a mesh of size $n\times n$. To construct the sets of triples discussed above, every processor storing c_i needs to determine the position of the second copy of c_i in the inorder traversal. Notice that every processor storing a d_j can construct the corresponding triple without needing any further information. The details follow.

Step 1. Begin by replicating the contents of the first row throughout the mesh. This is done by tasking every processor $P(1,i)$ to broadcasts the item it stores on the bus in its own column. Every processor reads the bus and stores the value broadcast.

Step 2. Every processor $P(i,i)$, $(1\leqslant i\leqslant n)$, broadcasts c_i on the bus in row i. The unique processor storing the second copy of label c_i will inform $P(i,i)$ of its position in the inorder sequence. A simple data movement now sends this information to $P(1,i)$. Clearly, at the end of Step 2, every processor in the first row of the mesh can construct the corresponding triple.

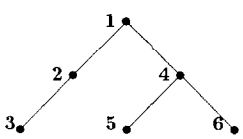
Step 3. Merge the two sequences of triples using Proposition 3.4 and store the result in the first row of the mesh. Finally, every processor retains the third coordinate of the triple it receives by merging. For an example of how this algorithm works the reader is referred to Fig. 8.

The correctness of the algorithm is easy to see. Since all steps take O(1) time, we have proved the following result.

Lemma 6.1. *Given the preorder and inorder traversals of an n-node binary tree, the corresponding preorder-inorder traversal can be constructed in* O(1) *time on an MMB of size* $n\times n$.

Our next goal is to show that once the preorder-inorder traversal $e_1,e_2,\ldots,e_{2n}$ is available, the corresponding binary tree can be reconstructed in O(1) time. Recall, that every label of a node in T occurs twice in the preorder-inorder traversal. Furthermore, by virtue of Step 2 above, the first copy of a label knows the position of its duplicate, and vice versa.

We associate a node u with every pair of identical labels in $e_1,e_2,\ldots,e_{2n}$. Let e_i and e_j be the first and second copy of a given label. The processor holding e_i assigns children pointers as follows:



Preorder Traversal: 1, 2, 3, 4, 5, 6

Inorder Traversal: 3, 2, 1, 5, 4, 6

Sequence 1: (1,3,1),(1,2,2),(1,1,3),(1,5,4),(1,4,5),(1,6,6)

Sequence 2: (2,1,3),(2,2,2),(2,3,1),(2,4,5),(2,5,4),(2,6,6)

Merged Sequence: (1,3,1),(1,2,2),(1,1,3),(2,1,3),(2,2,2),(2,3,1)

(1,5,4),(1,4,5),(2,4,5),(2,5,4),(1,6,6),(2,6,6)

Preorder-Inorder Traversal: 1, 2, 3, 3, 2, 1, 4, 5, 5, 4, 6, 6

Fig. 8. Illustrating the reconstruction of a binary tree.

- if e_{i-1} is the first copy of a label v, then left(u) $\leftarrow v$; otherwise, left(u) $\leftarrow$ **nil**;
- if e_{j+1} is the first copy of a label w, then right(v_i) $\leftarrow w$; otherwise, right(v_i) $\leftarrow$ **nil**.

The setting of pointers takes O(1) time. Therefore, Lemma 6.1 implies the following result.

Theorem 6.2. *An n-node binary tree can be reconstructed from its preorder and inorder traversals in constant time on an MMB of size $n \times n$.*

An ordered tree is an object that is either empty, or it consists of a root along with a possibly empty list $T_1, T_2, \ldots, T_k$ of subtrees, enumerated from left to right. Every node in an ordered tree stores a pointer to its leftmost child along with a pointer to its right sibling. The purpose of this section is to show that given its preorder and postorder traversals, an n-node ordered tree can be reconstructed in O(1) time on an MMB of size $n \times n$. We are presenting a slightly more general result, namely we show how to reconstruct an ordered forest from its preorder and postorder traversals.

Our algorithm relies on the well-known one-to-one correspondence between n-node ordered forests and n-node binary trees [6]. Specifically, let $F = (T_1, T_2, \ldots, T_m)$ be an ordered forest. The binary tree $B(F)$ corresponding to F is either empty (in case F is empty), or else is defined as follows:

- the root of $B(F)$ is $root(T_1)$;
- the left subtree of $B(F)$ is $B(T_{11}, T_{12}, \ldots, T_{1k})$, where $T_{11}, T_{12}, \ldots, T_{1k}$ are the subtrees of $root(T_1)$;
- the right subtree of $B(F)$ is $B(T_2, \ldots, T_m)$.

The following result is well-known [6].

Proposition 6.4. *F and $B(F)$ have the same preorder traversal. Furthermore, the postorder traversal of F is precisely the inorder traversal of $B(F)$.*

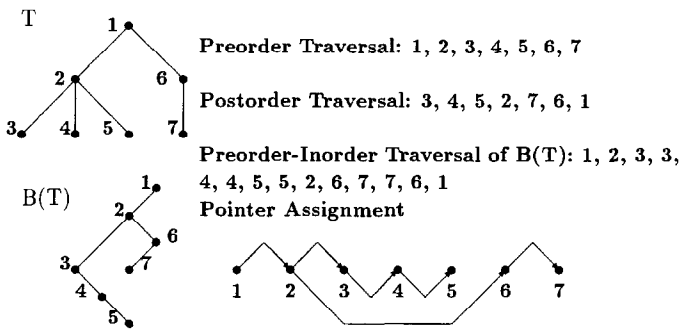


Fig. 9. Example of ordered tree reconstruction.

Proposition 6.4 motivates the following natural approach to reconstruct an ordered forest F from its preorder and postorder traversals. First, interpret the two traversals of F as the preorder and inorder traversals of the corresponding binary tree $B(F)$. Using the algorithm discussed in the previous section reconstruct $B(F)$. Finally, convert $B(F)$ to F.

We now present the details of the implementation of our forest reconstruction algorithm on an MMB of size $n \times 2n$. It is easy to modify the algorithm to work on an MMB of size $n \times n$. We assume that the preorder and postorder traversals of an ordered forest F are stored in the first row of the mesh in left to right order. Our algorithm proceeds as follows.

Step 1. Reconstruct the binary tree $B(F)$ having the same preorder traversal as F and whose inorder traversal corresponds to the postorder traversal of F.

Step 2. Let u be a generic node in $B(F)$; the processor in charge of u reinterprets pointers as follows:

- if $\text{left}(u) = v$ then set $\text{l\_child}(u) \leftarrow v$;
- if $\text{right}(u) = v$ then set $\text{r\_sibling}(u) \leftarrow v$.

Fig. 9 illustrates the reconstruction of an ordered tree from its traversals. The upper arrows indicate l_child pointers and the lower arrows indicate r_sibling pointers. The correctness of our algorithm is easy to see. Furthermore, by Theorem 6.3 the running time is O(1). Consequently, we have proved the following result.

Theorem 6.5. *An n-node ordered forest stored in the first row of an MMB of size $n \times n$ can be reconstructed from its preorder and postorder traversals in* O(1) *time.*

7. Concluding remarks and open problems

In this paper, we have presented a number of time-optimal tree algorithms on meshes with multiple broadcasting. Specifically, we have shown that the following tasks can

be solved in $\Theta(\log n)$ time:

- Encode an n-node binary tree into a $2n$-bitstring.
- Encode an n-node ordered tree into a $2n$-bitstring.
- Recover an n-node binary tree from its $2n$-bit encoding.
- Recover an n-node ordered tree from its $2n$-bit encoding.

We have also shown that the following tasks can be performed in $O(1)$ time:

- Reconstruct an n-node binary tree from its preorder and inorder traversals.
- Reconstruct an n-node ordered tree (forest) from its preorder and postorder traversals.

Our algorithms rely heavily on time-optimal algorithms for sequences of parentheses that we developed. Specifically, we have shown that each of the following tasks can be solved in $\Theta(\log n)$ time:

- Finding all the matching pairs in a well-formed sequence of parentheses.
- Determining the closest enclosing pair for every matching pair in a well-formed sequence.

A number of problems are open. In particular, it is not known whether reconstructing an ordered tree in parent-pointer format can be done in less than $O(\log n)$ time. It is clear that such an algorithm using the closest enclosing pair can be devised. Can one do better?

A very hard and important problem is to determine the *smallest* size MMB on which instances of size n of the above tree-related computations run in $O(\log n)$ time, that is as fast as possible. To the best of our knowledge this question is still open.

References

[1] D. Bhagavathi, P.J. Looges, S. Olariu, J.L. Schwing and J. Zhang, A fast selection algorithm on meshes with multiple broadcasting, IEEE Trans. Parallel Distributed Systems 5 (1994) 772–778.

[2] D. Bhagavathi, S. Olariu, J.L. Schwing, W. Shen, L. Wilson and J. Zhang, Convexity problems on meshes with multiple broadcasting, J. Parallel Distributed Comput. 27 (1995) 142–156.

[3] D. Bhagavathi, S. Olariu, W. Shen and L. Wilson, A time-optimal multiple search algorithm on enhanced meshes, with Applications, J. Parallel Distributed Comput. 22 (1994) 113–120.

[4] S.A. Cook, C. Dwork and R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, SIAM J. Comput. 15 (1986) 87–97.

[5] J. JáJá, An Introduction to Parallel Algorithms (Addison-Wesley, Reading, MA, 1991).

[6] D. Knuth, The Art of Computer Programming: Fundamental Algorithms, Vol. 1, 2nd ed. (Addison-Wesley, Reading, MA, 2nd ed., 1973).

[7] V.P. Kumar and C.S. Raghavendra, Array processor with multiple broadcasting, J. Parallel Distributed Comput. 2 (1987) 173–190.

[8] V.P. Kumar and D.I. Reisis, Image computations on meshes with multiple broadcast, IEEE Trans. Pattern Anal. Machine Intelligence 11 (1989) 1194–1201.

[9] R. Lin, S. Olariu, J.L. Schwing and J. Zhang, Simulating enhanced meshes, with applications, Parallel Process. Lett. 3 (1993) 59–70.

[10] M. Maresca and H. Li, Connection autonomy in SIMD computers: a VLSI implementation, J. Parallel Distributed Comput. 7 (1989) 302–320.

[11] S. Olariu, C.M. Overstreet and Z. Wen, Reconstructing binary trees in doubly logarithmic CREW time, J. Parallel Distributed Comput. 27 (1995) 100–105.

[12] S. Olariu, J.L. Schwing and J. Zhang, Optimal parallel encoding and decoding algorithms for trees, Internat. J. Found. Comput. Sci. 3 (1992) 1–10.

[13] S. Olariu, J. L. Schwing and J. Zhang, Optimal convex hull algorithms on enhanced meshes, *BIT* 33 (1993) 396–410.
[14] D. Parkinson, D.J. Hunt and K.S. MacQueen, The AMT DAP 500, Proceedings of the 33rd IEEE Comp. Soc. International Conference (1988) 196–199.
[15] S. Zaks, Lexicographic generation of ordered trees, Theoret. Comput. Sci. 10 (1980) 63–82.