More general parallel tree contraction: Register allocation and broadcasting in a tree

Krzysztof Diks Torben Hagerup

MPI–I–96–1-024 October 1996

More general parallel tree contraction: Register allocation and broadcasting in a tree

Krzysztof Diks* and Torben Hagerup†

Abstract

We extend the classic parallel tree-contraction technique of Miller and Reif to handle the evaluation of a class of expression trees that does not fit their original framework. We discuss applications to the following problems: (1) Register allocation, i.e., computing the number of registers needed to evaluate a given expression if all intermediate results must be kept in registers; and (2) Broadcasting in a tree, i.e., computing the number of steps needed to transmit a message from the root to all other nodes in a given rooted tree if each node is a processor that can communicate with a single neighbor in each step. We show that on inputs of size n, both problems can be solved with optimal speedup in $O((\log n)^2)$ time on an EREW PRAM, in $O(\log n \log\log n)$ time on a CREW PRAM, and in $O(\log n)$ time on a CRCW PRAM.

1 Introduction

Register allocation for the evaluation of an expression is a central concern in code generation. Atomic operands (variables and constants) occurring in the expression are generally assumed to reside in main memory initially, while operators can take their operands from either registers or main memory. For reasons of speed, all intermediate results are to be kept in registers. Since registers are a scarce resource, the goal is to use as few registers as possible (this can be shown to optimize other criteria as well). Before an expression can be evaluated, its nonatomic immediate subexpressions must be evaluated and their values left in registers. The order in which subexpressions are evaluated, which the compiler is free to choose, is what determines the number of registers used.

The problem of *information dissemination* or *broadcasting* in a tree was introduced and motivated in [21]. At time 0, the root of the tree generates a message to be distributed to all other nodes in the tree. The goal is to minimize the *broadcast time*, the earliest instant in time at which all nodes have received the message. At each integral time instant after it receives the message, a node can pass it on to exactly one of its children. The order in which children are notified, which can be chosen freely, is what determines the broadcast time.

The problems of register allocation for an expression and broadcasting in a tree show strong similarities. These become most evident when one considers recursive characterizations of optimal solutions. Suppose that the immediate subexpressions of an expression E are $E_1, \ldots, E_k$

*Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, 02–097 Warszawa, Poland. diks@mimuw.edu.pl
†Max-Planck-Institut für Informatik, Im Stadtwald, D–66123 Saarbrücken, Germany. torben@mpi-sb.mpg.de

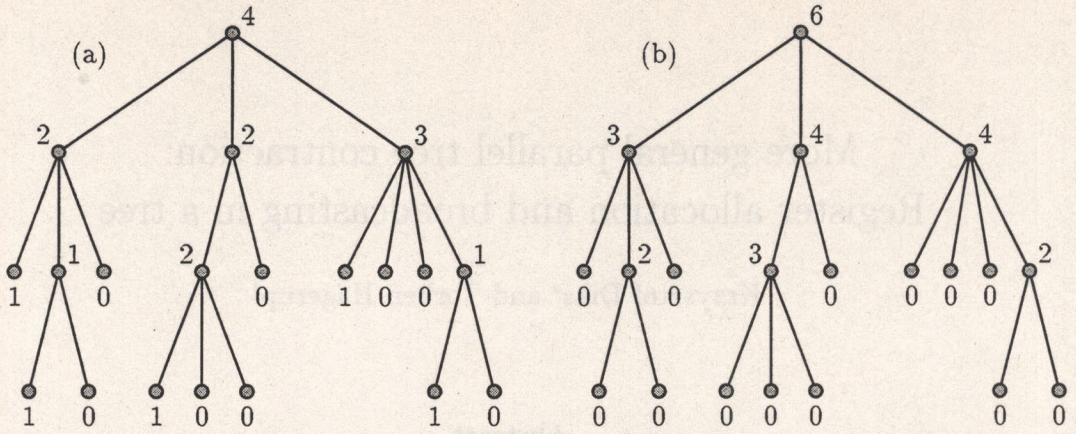

Fig. 1. A tree in which each node u is shown with (a) the register requirements (b) the optimal broadcast time of the maximal subtree rooted at u.

and that the number of registers needed to evaluate E_i is b_i, for $i = 1, \ldots, k$. If the subexpressions of E are evaluated in the order $E_1, \ldots, E_k$, the number of registers used in the evaluation of E will be $b = \max_{1 \le i \le k}(b_i + i - 1)$, the reasoning being that while E_i is under evaluation, $i - 1$ registers are needed to hold the values of $E_1, \ldots, E_{i-1}$. A moment's thought reveals that b is minimized, over all permutations of $E_1, \ldots, E_k$, if $b_1 \ge b_2 \ge \cdots \ge b_k$ (see [2, Fig. 9.23]). Similarly, it is shown in [21, Theorem 1] that if $T_1, \ldots, T_k$ are the maximal subtrees of a tree T rooted at the children of the root of T, if b_i is the optimal broadcast time of T_i, for $i = 1, \ldots, k$, and if $b_1 \ge b_2 \ge \cdots \ge b_k$, then the optimal broadcast time of T is $\max_{1 \le i \le k}(b_i + i)$. Thus the only formal difference between the two problems is an additive constant, plus different boundary conditions: The broadcast time of every 1-node tree is 0, whereas the register count assigned to a leaf in [2] is 1 if the leaf is the leftmost child of its parent, and 0 otherwise (see Fig. 1); we follow the latter convention, but note that different assignments of 0 and 1 to leaves may be appropriate for different instruction sets. Both problems can be solved in linear time; in the case of computing optimal broadcast times, this was shown in [21]. The register-allocation and broadcasting problems make sense in a more general setting: One can ask for the minimum number of registers needed to execute a straight-line program, represented as a directed acyclic graph, or for the optimal broadcast time of an arbitrary connected, undirected graph with a distinguished start node. The reason for restricting attention to expressions and trees, however, is compelling: (Decision versions of) the more general problems are NP-hard [19, 21].

Let $\mathbb{N} = \{1, 2, \ldots\}$ and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. Motivated by the applications described above, we take $\odot_\mu$, for $\mu \in \{0, 1\}$, to be the function from $\mathbb{N}_0^+$, the set of nonempty sequences of nonnegative integers, to $\mathbb{N}_0$ defined as follows: For all $k \in \mathbb{N}$ and all $a_1, \ldots, a_k \in \mathbb{N}_0$,

$$\odot_\mu(a_1, \ldots, a_k) = \max_{1 \le i \le k} (b_i + i - \mu),$$

where $\{b_1, \ldots, b_k\}$ is the same multiset as $\{a_1, \ldots, a_k\}$ and $b_1 \ge b_2 \ge \cdots \ge b_k$. In other words, the arguments of $\odot_\mu$ are first sorted into nonincreasing order, then the position of each argument in the sorted sequence minus μ is added to the argument, and the value of $\odot_\mu$ is the maximum among the resulting numbers. We define the *$\odot$-problem* as follows: An instance of the problem is given by a rooted tree T in which each leaf is labeled with an element of $\mathbb{N}_0$ and each internal

node is labeled with one of the functions $\odot_0$ and $\odot_1$, and the task is to compute the *value* of T, defined as the value of its root; the value of a leaf is its associated label, and the value of an internal node u is $\odot_\mu(b_1, \ldots, b_k)$, where $\odot_\mu$ is the function labeling u and $b_1, \ldots, b_k$ are the values of its children. (We have no use for "mixed" trees with both $\odot_0$ and $\odot_1$ labels, but the generalization comes for free.)

While it is not difficult to show that the $\odot$-problem is in NC, obtaining our best results requires some care and rather intricate variants of tree contraction. Denoting the number of nodes in the input tree by n and assuming that all leaf values are bounded by 1 (as in our applications) and that $n \geq 4$ (so that $\log\log n$ is well-defined and at least 1), we achieve the following time bounds, in all cases together with a linear time-processor product, i.e., with optimal speedup: $O((\log n)^2)$ on the EREW PRAM, $O(\log n \log\log n)$ on the CREW PRAM, and $O(\log n)$ on the CRCW PRAM. Using a reduction from the problem of computing the parity of n bits, the latter two time bounds can be shown to be a factor of at most $\Theta(\log\log n)$ larger than the minimum achievable (in the case of the CRCW PRAM, with any polynomial number of processors). For the EREW PRAM, the actual time bound is $O(\log n \log(d+1))$, where d is the maximum degree of any node in the input tree (the *degree* of a node in a rooted tree is defined as the number of its children). For $d = 2$, our (EREW PRAM) result was obtained previously by Miller and Teng [17].

2 Tree contraction

The $\odot$-problem is a special case of the more general problem of evaluating an expression tree over a domain D and a set Ω of operators from D^+ to D. An instance is here given by a rooted and ordered tree T in which each leaf is labeled with an element of D and each internal node is labeled with an operator in Ω. The goal again is to compute the value of the root, where the value of a leaf is its label, and the value of an internal node with label $\diamond$, whose k children, in the order from left to right, have values $b_1, \ldots, b_k$, is $\diamond(b_1, \ldots, b_k)$.

The *tree-contraction* technique, introduced by Miller and Reif [14] and developed further in a series of papers [12, 15, 16, 17, 18], is a fundamental tool for the parallel evaluation of expression trees. A very simple variant of tree contraction for the case of binary operators was indicated in [1, 13]. The simple algorithm can also be applied in the case of unbounded-degree trees whenever, for each operator $\diamond \in \Omega$, $\diamond(b_1, \ldots, b_k)$ can be construed as $b_1 \diamond b_2 \diamond \cdots \diamond b_k$ for some binary associative operator $\diamond$. This is not the case for our operators $\odot_0$ and $\odot_1$, however, so that we have to deal directly with high degrees; this is what makes the problem challenging.

Tree contraction evaluates an expression tree by repeatedly shrinking it, while modifying labels stored in the tree in such a way as to preserve the value of the tree. Miller and Reif formulate the technique in terms of two operations, RAKE and COMPRESS. Described at a high level that, in particular, ignores the calculations involving labels, RAKE removes a leaf from the (current) tree, while COMPRESS *merges* two adjacent degree-1 nodes (the result of merging a node v with its parent u is a new node with the same parent as u, if any, and a set of children consisting of all children of v and all children of u except v). If RAKE and COMPRESS operations can be carried out in constant time (this depends on the label calculations), then tree contraction can reduce an n-node tree to a constant size in $O(\log n)$ time without changing its value, after which the value of the tree can be determined in constant time. In our case, COMPRESS operations can be executed in constant time, as required, but simultaneously raking

k leaves with the same parent involves sorting k items, which cannot be done in constant time. Miller and Reif were faced with a similar problem in [16], where tree contraction is used to compute so-called canonical labelings of trees. In their case, if $k \geq 2$ children of the same node simultaneously become leaves, due to the execution of RAKE operations, it is necessary to spend $\Theta(\log k)$ time sorting labels associated with the k leaves, after which they can be removed. Miller and Reif showed that even with this complication, the tree contraction finishes within $O(\log n)$ time. What makes our problem more difficult is that even after any amount of processing, we cannot remove any leaf children of a node u as long as u has two or more nonleaf children; this is a reflection of the fact that even if all but one of the arguments of $\odot_\mu$ are known, the value of $\odot_\mu$, once the final argument is revealed, could still depend on all the other arguments individually. We show how to cope with such a situation within the framework of tree contraction.

The lemma below, which describes generic tree contraction, was essentially proved in a less general form by Miller *et al.* [15, 16, 12, 17]. Their description does not provide a clean interface between generic tree contraction and its various applications, however, so that the lemma can be extracted from their exposition only with some effort. For this reason we provide a self-contained proof of the lemma in an appendix. An informal explanation of the lemma is given after its statement. We restrict attention to operators $\diamond$ that are *commutative*, i.e., whose value is invariant under permutation of arguments. This is a genuine restriction, but the operators $\odot_0$ and $\odot_1$ of interest here are clearly commutative.

Let k and l be integers with $0 \leq l \leq k$, and let D be a set. An *l-dimensional projection* of a function $\diamond$ from D^k to D is the function from D^l to D obtained from $\diamond$ by fixing $k - l$ arguments to be constants in D, while leaving the remaining l arguments as indeterminates. E.g., the l-dimensional projection of $\diamond$ obtained by fixing the $k - l$ first arguments at the values $b_1, \ldots, b_{k-l}$ maps $(x_1, \ldots, x_l)$ to $\diamond(b_1, \ldots, b_{k-l}, x_1, \ldots, x_l)$ for all $(x_1, \ldots, x_l) \in D^l$. Given sets D and Ω, where each element of Ω is a function from D^+ to D, let $\mathcal{T}_{D,\Omega}$ be the set of all rooted trees in which each leaf is labeled with an element of D and each internal node is labeled with an element of Ω.

Lemma 2.1 *Let D be a set, Ω a set of commutative functions from D^+ to D, and $\mathcal{T}$ a subset of $\mathcal{T}_{D,\Omega}$. If there are τ, p and $\mathcal{F}$ satisfying Conditions (A)-(E) below, then, for all integers $n \geq 2$, every n-node tree $T \in \mathcal{T}$ can be evaluated in $O(\tau(T) \log n + \log(p(T)))$ time on a PRAM with $np(T)$ processors.*

(A) $\mathcal{F}$ is a set of functions from D to D, and for all $f \in \mathcal{F}$ and all $x \in D$, $f(x)$ can be computed from f and x in constant time with one processor.

(B) For any two functions f and g in $\mathcal{F}$, $g \circ f$ belongs to $\mathcal{F}$ and can be computed from f and g in constant time with one processor.

(C) Every 1-dimensional projection of a function in Ω belongs to $\mathcal{F}$.

(D) τ and p are functions from $\mathcal{T}$ to $\mathbb{N}$ such that for all integers $n \geq 2$ and all n-node trees $T \in \mathcal{T}$, $\tau(T)$ and $p(T)$ can be computed from T in $O(\tau(T) \log n)$ time with $np(T)$ processors.

(E) Fix an input tree $T^* \in \mathcal{T}$ with $n \geq 2$ nodes, let $\tau^* = \tau(T^*)$ and $p^* = p(T^*)$ and consider the following setting: A sequence $a_1^*, \ldots, a_s^*$ of s integers, where s is a nonnegative integer

bounded by the maximum degree of a node in T^* and where $1 \leq a_1^* \leq a_2^* \leq \cdots \leq a_s^* = O(\tau^* \log n)$, is available for preprocessing in $O(\tau^* \log n)$ time with s processors. Assume that the preprocessing finishes at time 0 (this is just a convention for fixing the origin of the time axis). Subsequently, at time a_i^*, for $i = 1, \ldots, s$, the value b_i^* of a node r_i in T^* becomes known, and $p^* n_i$ processors numbered $p^* \sum_{j=1}^{i-1} n_j + 1, \ldots, p^* \sum_{j=1}^{i} n_j$ become available, where n_j is the number of nodes in the maximal subtree T_j^* of T^* rooted at r_j, for $j = 1, \ldots, s$. Thus every value that becomes known contributes new processors, and the available processors at all times are consecutively numbered. Moreover, the trees $T_1^*, \ldots, T_s^*$ are disjoint.

In these circumstances, for some constant $C > 0$ and for all $\diamond \in \Omega$, the function $f : D \to D$ mapping x to $\diamond(b_1^*, \ldots, b_s^*, x)$, for all $x \in D$, must be computable with the available processors to be ready by time

$$\max_{a_1^* \leq j \leq a_s^*} (j + C\tau^* \log(|F_j| + 2)),$$

where $F_j = \{i : 1 \leq i \leq s \text{ and } a_i^* = j\}$, for $j = a_1^*, a_1^* + 1, \ldots, a_s^*$; for $s = 0$ we take this condition to mean that f must be computable in constant time with one processor.

The model of computation is specified in Lemma 2.1 only as a PRAM; the lemma holds for all variants of the PRAM at least as strong as the EREW PRAM. When there are no restrictions on concurrent reading, the term $\log(p(T))$ in the time bound can be removed. While Conditions (A)–(C) of Lemma 2.1 are readily interpreted in the context of [15, 16, 12, 17] and Condition (D) is a simple extension (allowing more time and processors), Condition (E) may look unfamiliar. Its closest analogue in the work of Miller and Reif is their requirement [16] that "the k leaf children of a node can be raked in $O(\log k)$ time", but this formulation is neither precise nor general enough for our needs.

Since a major concern in the remainder of the paper will be to show that Condition (E) is satisfied in a number of situations, we introduce terminology that facilitates the discussion. First, the processing required in Condition (E), i.e., the computation of the function mapping x to $\diamond(b_1^*, \ldots, b_s^*, x)$, for all $x \in D$, will be referred to as *raking*. The quantities $b_1^*, \ldots, b_s^*$ are called *keys* to distinguish them from other integers. For $1 \leq i < j \leq s$, we consider b_i^* and b_j^* to be distinct keys even if they have the same numerical value. For $i = 1, \ldots, s$, a_i^* is called the *arrival time* of the key b_i^*, and for $j = a_1^*, a_1^* + 1, \ldots, a_s^*$, the set of keys with arrival time j will be called a *family*, also considered to have arrival time j. We distinguish between families with distinct arrival times even if they happen to be empty. The *lead* of a family with arrival time j is $t - j$, where t is the time at which the raking finishes. Finally, for any finite set S, we define the *log-size* of S as $\log(|S| + 2)$. Condition (E) can now be expressed by saying that the lead of some family must be within a constant factor of τ^* times its log-size.

3 The algorithms

In our application of tree contraction to the $\odot$-problem, we take $D = \mathbb{N}_0$ and $\Omega = \{\odot_0, \odot_1\}$. For all $\alpha, \beta, \gamma \in \mathbb{N}_0$ with $\alpha < \beta$, denote by $\phi_{\alpha,\beta,\gamma}$ the function from $\mathbb{N}_0$ to $\mathbb{N}_0$ given by

$$\phi_{\alpha,\beta,\gamma}(x) = \begin{cases} \gamma, & \text{if } x < \alpha, \\ \gamma + 1, & \text{if } \alpha \leq x < \beta, \\ \gamma + 2 + x - \beta, & \text{if } x \geq \beta, \end{cases}$$

for all $x \in \mathbb{N}_0$, and take $\mathcal{F} = \{\phi_{\alpha,\beta,\gamma} \mid \alpha, \beta, \gamma \in \mathbb{N}_0 \text{ and } \alpha < \beta\}$. The function $\phi_{\alpha,\beta,\gamma} \in \mathcal{F}$ can be represented via the triple (α, β, γ). Condition (A) of Lemma 2.1 is clearly satisfied, and Conditions (B) and (C) are demonstrated in the following lemmas.

Lemma 3.1 *For any two functions f and g in $\mathcal{F}$, $g \circ f$ belongs to $\mathcal{F}$ and can be computed from f and g in constant time with a single processor.*

PROOF Given a function $f = \phi_{\alpha,\beta,\gamma} \in \mathcal{F}$, we define f^{-1} as the function from $\{y \in \mathbb{N} \mid y > \gamma\}$ to $\mathbb{N}_0$ with $f^{-1}(y) = \min\{x \in \mathbb{N}_0 \mid f(x) = y\}$, for all $y > \gamma$. f^{-1} can be evaluated in constant time and is strictly increasing, and $f^{-1}(y+1) = f^{-1}(y) + 1$ for all $y \geq \gamma + 2$.

Now let $f = \phi_{\alpha,\beta,\gamma}$ and g be functions in $\mathcal{F}$, take $h = g \circ f$ and define γ' as $g(\gamma)$ and h^* as $f^{-1} \circ g^{-1}$. Since $g^{-1}(\gamma' + 1) = g^{-1}(g(\gamma) + 1) > \gamma$, the domain of h^* includes $\{y \in \mathbb{N}_0 \mid y > \gamma'\}$.

We claim that $h = \phi_{\alpha',\beta',\gamma'}$, where $\alpha' = h^*(\gamma' + 1)$ and $\beta' = h^*(\gamma' + 2)$. First observe that for all $y > \gamma'$, $h^*(y) = \min\{x \in \mathbb{N}_0 \mid h(x) = y\}$. Thus $h(x) = \gamma'$ for all $x < \alpha'$, and $h(x) = \gamma' + 1$ for all x with $\alpha' \leq x < \beta'$. Since clearly $h(\beta) = \gamma' + 2$, what remains is to show that $h(x+1) = h(x) + 1$ for all $x \geq \beta$. But this follows from the fact that for all $y \geq \gamma' + 2$,

$$h^*(y+1) = f^{-1}(g^{-1}(y+1)) = f^{-1}(g^{-1}(y) + 1) = f^{-1}(g^{-1}(y)) + 1 = h^*(y) + 1.$$

□

Lemma 3.2 *For all $\mu \in \{0, 1\}$, all $k \in \mathbb{N}_0$ and all $b_1, \ldots, b_k \in \mathbb{N}_0$, the function f_μ from $\mathbb{N}_0$ to $\mathbb{N}_0$ mapping x to $\odot_\mu(b_1, \ldots, b_k, x)$, for all $x \in \mathbb{N}_0$, belongs to $\mathcal{F}$.*

PROOF For $k = 0$, $f_\mu = \phi_{1,2,1-\mu}$, so assume that $k \geq 1$. We first consider the case $\mu = 0$ and let $\odot = \odot_0$. Given a sequence $a_1, \ldots, a_k$ of arguments to $\odot$ with $a_1 \geq \cdots \geq a_k$, we consider a_i, for $i = 1, \ldots, k$, to have a *value*, a_i, an *index*, i, and a *contribution*, $a_i + i$. Define an index i and an argument a_i to be *critical* if $a_i + i = \odot(a_1, \ldots, a_k)$.

Assume without loss of generality that $b_1 \geq \cdots \geq b_k$. The insertion of x into the sorted sequence $b_1, \ldots, b_k$ increases the contribution of every element following x by 1 and leaves the contribution of every element preceding x unchanged. Let i be the maximal critical index of the sequence $b_1, \ldots, b_k$ and consider three cases:

Case 1: $x < b_i$. Then $\odot(b_1, \ldots, b_k, x) = \odot(b_1, \ldots, b_k) = b_i + i$. For no element following x in the sorted sequence was critical before the insertion of x, and the contribution of x itself exceeds $\odot(b_1, \ldots, b_k)$ only if x immediately follows an element of the same value that was critical before the insertion of x, which contradicts the assumption $x < b_i$.

Case 2: $b_i \leq x < b_i + i + 1$. Now $\odot(b_1, \ldots, b_k, x) = b_i + i + 1$. For certainly $\odot(b_1, \ldots, b_k, x) \geq b_i + i + 1$, since x can now be placed before b_i, which increases the contribution of b_i by 1. On the other hand, either the contribution of x is $x + 1 \leq b_i + i + 1$ (if x is inserted in the first position in the sorted sequence), or the contribution of x is at most one more than that of its predecessor, i.e., again bounded by $\odot(b_1, \ldots, b_k) + 1$.

Case 3: $x \geq b_i + i + 1$. Then $x > b_1$ and $\odot(b_1, \ldots, b_k, x) = x + 1$.

Putting together Cases 1–3, we see that $f_0 = \phi_{\alpha,\beta,\gamma} \in \mathcal{F}$, where $\alpha = b_i$, $\beta = b_i + i + 1$ and $\gamma = b_i + i \geq 1$. This also implies that $f_1 = \phi_{\alpha,\beta,\gamma-1}$. □

3.1 The EREW PRAM algorithm

For the EREW PRAM we take $\mathcal{T} = \mathcal{T}_{D,\Omega}$ and $\tau(T) = \lceil \log(d+1) \rceil$, where d is the maximum degree of a node in T, and $p(T) = 1$ for all $T \in \mathcal{T}$. Then Condition (D) of Lemma 2.1 is easily seen to be satisfied. What remains is to show how to satisfy Condition (E). But this is also easy: We simply wait until time a_s^*, i.e., until all of the keys $b_1^*, \ldots, b_s^*$ are available, and then process them as implicit in the proof of Lemma 3.2. For $\diamond = \odot_\mu$, this involves sorting $b_1^*, \ldots, b_s^*$ into nonincreasing order, adding $i - \mu$ to the ith element in the sorted sequence, for $i = 1, \ldots, s$ (we call this "adding appropriate offsets"), computing the maximum in the resulting sequence, and determining the largest position of an occurrence of the maximum. Since $s \leq d$, all of this can be done in $O(\tau^*)$ time on an EREW PRAM with s processors, the only nontrivial subroutine needed in fact being one for logarithmic-time sorting with optimal speedup [3, 8]. We have proved:

Lemma 3.3 *For all integers $n \geq 2$ and $d \geq 1$, $\odot$-problems can be solved on input trees with n nodes and maximum degree d in $O(\log n \log(d+1))$ time on an EREW PRAM with n processors.*

3.2 The CRCW PRAM algorithm

For the CRCW PRAM we take $\tau(T) = 1$ and $p(T) = \lceil \log n \rceil^2$ for all $T \in \mathcal{T}$ with $n \geq 2$ nodes; $\mathcal{T}$ will be specified below. Again Condition (D) of Lemma 2.1 is obviously satisfied. Consider Condition (E).

We cannot follow the strategy of the EREW PRAM algorithm of waiting until all keys $b_1^*, \ldots, b_s^*$ have become available before starting the raking, the reason being that a PRAM with a polynomial number of processors cannot sort in constant time. Instead we have to carry out the sorting incrementally, i.e., to sort keys as they arrive. More precisely, our raking algorithm proceeds as follows:

Let $\lambda \in \mathbb{N}$ be a constant to be fixed later. We divide the s keys into $q = \lceil (a_s^* - a_1^* + 1)/(2\lambda) \rceil$ *generations* $G_1, \ldots, G_q$. For $i = 1, \ldots, q$, a key belongs to the ith generation G_i if its arrival time lies in the set $\{a_s^* - i(2\lambda) + 1, \ldots, a_s^* - (i-1)(2\lambda)\}$, called the *span* of G_i. In other words, the generations are numbered "backwards", and except for G_q, which may be smaller, each generation spans exactly 2λ time units. We say that a family belongs to a generation G if its arrival time lies in the span of G. The raking algorithm sorts the keys within each generation; we will use G_i also to denote the sorted sequence obtained by sorting the keys in the ith generation, for $i = 1, \ldots, q$, relying on context to resolve any ambiguity. In addition, with $H_q = G_q$, G_i is merged with H_{i+1} to create the sorted sequence H_i, for $i = q-1, \ldots, 1$ (see Fig. 2). Each sorting and merging operation is started as soon as its input is available. Thus several generations may be sorted simultaneously, while the merges must happen in a strictly sequential fashion. When the final sorted sequence H_1 has been produced, we remember its maximum M and add appropriate offsets in the range $\{0, \ldots, s\}$. Since the maximum of the resulting sequence S obviously lies in the range $\{M, \ldots, M+s\}$, computing this quantity reduces to computing the maximum of the sequence S' obtained from S by subtracting M from each of its elements and replacing negative elements by zero. The elements of S' lie in the range $\{0, \ldots, s\}$, which means that their maximum can be computed in constant time with s processors [11, Theorem 1]. The final nontrivial step of the raking, the computation of the position in S of the last occurrence of the maximum, also reduces to computing the maximum of s integers in the range $\{0, \ldots, s\}$

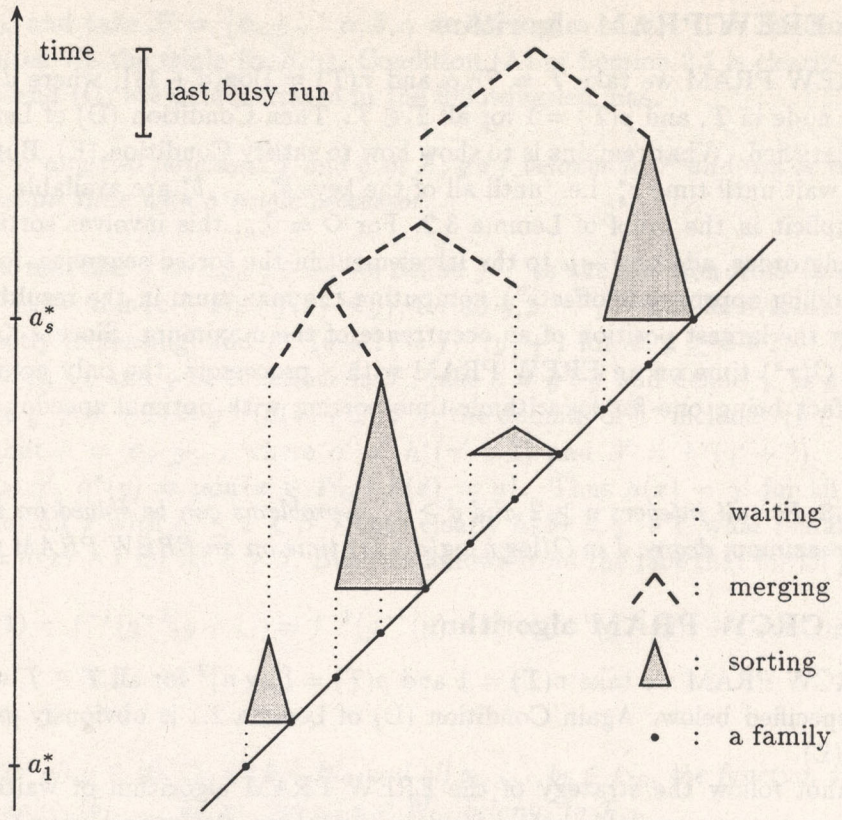

Fig. 2. Sorting and merging generations.

(replace every occurrence of the maximum by its position and every other number by zero). Thus the raking finishes within a constant delay after the end of the last merge.

We still have to specify the routines used for sorting and merging. We employ a standard sorting algorithm using logarithmic time and a linear number of processors [3, 8] and a constant-time merging algorithm characterized in the lemma below.

Lemma 3.4 *For all integers N and M with $2 \leq N \leq M$ and all fixed $\epsilon > 0$, two sorted sequences, each containing N integers in the range $\{0, \ldots, M\}$, can be merged on a CRCW PRAM using constant time and $O(N(M/N)^\epsilon \log N)$ processors.*

PROOF If $M > N^{1+1/\epsilon}$, at least N^2 processors are available, and it is very easy to carry out the merging of arbitrary values (not necessarily integers) in constant time, even on a CREW PRAM. We will therefore assume that $M \leq N^{1+1/\epsilon}$, so that $\log M = O(\log N)$.

The *segmented-broadcasting* problem of size N takes as input an array A of N cells, some of which contain *significant objects*, and the goal is to mark each cell in A with the position of the nearest significant object to its left, if any. It is known that for all integers $N \geq 2$, segmented-broadcasting problems of size N can be solved on a CRCW PRAM using constant time and $O(N \log N)$ processors [5, 7].

Returning to the merging problem, let X and Y be the two input sequences. Multiplying all elements in X and Y by 2 and subtracting 1 from those in X only, we ensure that no value

occurs in both X and Y. It now suffices to compute the rank of each element z of X and Y in the opposite sequence, i.e., the number of elements $\leq z$ in that sequence, since this number added to the position of the element in its own sequence yields its position in the output sequence. We describe how to compute the rank in Y of each element in X.

Let $Y = (y_1, \ldots, y_N)$. For $i = 1, \ldots, N$, store the rank in Y of y_i in $A[y_i]$, where $A[0\,..\,2M]$ is an array. This is easy to do in constant time with N processors: For $i = 1, \ldots, N$, if $i = N$ or $y_i \neq y_{i+1}$, then store i in $A[y_i]$. Considering the values thus stored in some of the cells of A as significant objects, the remaining task is, for each value x occurring in X, to compute the nearest significant object to the left of $A[x]$, since this is the desired rank of x in Y. We could carry out this task by solving the segmented-broadcasting problem defined by A. However, A is large, and we will instead compute only the N entries actually needed.

Let T be a tree of constant height with the cells of A in the natural order at its leaves and with the following degree restrictions: The root of T is of degree $O(N)$, and all other nodes in T are of degree $O((M/N)^\epsilon)$. Mark each node in T that is the root of a subtree containing a significant object, and subsequently solve a segmented-broadcasting problem for each marked internal node u in T. The problem associated with u is defined over the sequence of its children by taking a marked child to be significant object. In order to solve these segmented-broadcasting problems, we use $O(N \log N)$ processors at the root of T and $O((M/N)^\epsilon \log N)$ processors at each remaining marked internal node, a total of $O(N(M/N)^\epsilon \log N)$ processors.

Now a single processor associated with an element x in X can use the information stored in T to navigate to the nearest marked leaf to the left of $A[x]$. First it moves upwards, starting at $A[x]$, until it hits a marked node u with a marked sibling to its left. If this never happens, x is smaller than all elements in Y. Otherwise the processor descends from the nearest marked sibling of u to the left of u, always proceeding to the rightmost marked child of the current node. When it reaches a leaf, this will be the nearest marked leaf to the left of $A[x]$.

The description given in the previous paragraph implicitly assumed that an uninitialized node in T can be recognized as such. This assumption can be obviated simply by augmenting the marking of a node with a proof of its validity, namely a pointer to an element in Y corresponding to a marked leaf descendant. □

As can be seen, the algorithm of Lemma 3.4 applies only to input sequences consisting of small integers, for which reason we will place restrictions on the labels in the input tree. More precisely, we fix a constant $\nu \in \mathbb{N}$ and let $\mathcal{T}$ be the set of those trees T in $\mathcal{T}_{D,\Omega}$ whose leaf labels are all bounded by $m = (\log n)^\nu$, where $n \geq 2$ is the number of nodes in T (fix any convention for $n = 1$). It is easy to see by induction that the value of every tree in $\mathcal{T}_{D,\Omega}$ with N nodes and leaf labels of at most m is bounded by mN. Thus, whenever our raking algorithm merges two sequences involving $N \geq 2$ keys and a maximum key value of M, the number of processors available for the merge will be at least $p^* \max\{N, M/m\} \geq \max\{N(\log n)^2, M/m\}$. With $\epsilon = 1/(3\nu)$, this is at least $N(M/N)^\epsilon \log N$. For if $(M/N)^\epsilon \leq \log n$, the claim is obvious; and if not, $(M/N)^{1/3} \geq m \geq \log N$ and hence $M/m = N(M/N)^{1/3}(M/N)^{1/3}(M/N)^{1/3}/m \geq N(M/N)^\epsilon \log N$. It now follows from Lemma 3.4 that every merge carried out by the raking algorithm can be executed in constant time. We choose the constant $\lambda \in \mathbb{N}$ anticipated above as any constant upper bound on the number of time steps needed for a single merge.

Define the lead of a generation G as the number of time steps from the beginning of the sorting of G to the end of the raking. In order to show that the raking is sufficiently fast, we

must find a generation whose lead is within a constant factor of its log-size; to see this, we use the facts that the log-size of each generation is within a constant factor of the log-size of the largest family in the generation (since there are at most 2λ such families), and that the lead of every family in a generation G is within a constant factor (in fact, within an additive term of 2λ) of the lead of G. We can clearly assume that $q \geq 2$, i.e., that at least one merge takes place.

Say that a time step is *busy* if some merge is in progress during the step. Note that sorting may take place during a nonbusy step—we care only about merges. Define a *busy run* as a maximal sequence of busy time steps, let k be the number of merges carried out during the last busy run and consider two cases:

Case 1: $k \leq 2$. What triggers the last busy run is the completion of the sorting of some generation G. Since the last busy run takes only constant time, certainly the lead of G is within a constant factor of its log-size.

Case 2: $k \geq 3$. Let t be the time at which the last busy run begins and note that the last merge completes no later than at time $t + \lambda k$. The fact that G_k and H_{k+1} cannot start merging before time t implies that for some l with $k \leq l \leq q$, the sorting of G_l has not been completed by time $t - 1$. Since generations begin only every 2λ steps, the sorting of G_l starts no later than $2\lambda(l-1)$ steps before the beginning of the last merge, i.e., no later than at time $t + \lambda k - 2\lambda(l-1) \leq t + \lambda k - 2\lambda(k-1) = t - \lambda(k-2)$. Thus the time r needed for the sorting of G_l is at least $\lambda(k-2) \geq \lambda k/3$, and the lead of G_l is at most $r + \lambda k$. It follows that the lead of G_l is within a constant factor of its log-size, as desired.

Lemma 3.5 *For all integers $n \geq 2$ and all constants $\nu \in \mathbb{N}$, $\odot$-problems can be solved on input trees with n nodes and all leaf labels bounded by $(\log n)^\nu$ in $O(\log n)$ time on a CRCW PRAM with $O(n(\log n)^2)$ processors.*

3.3 The CREW PRAM algorithm

For the CREW PRAM we again allow all input trees ($\mathcal{T} = \mathcal{T}_{D,\Omega}$) and take $\tau(T) = \lceil \log \log n \rceil$ and $p(T) = \lceil \log n \rceil^2$ for all $T \in \mathcal{T}$ with $n \geq 4$ nodes. The raking now has to perform not only the sorting, but also the maximum-finding in an incremental fashion, since a CREW PRAM cannot carry out either task in sublogarithmic time (even for inputs consisting of small integers). Our approach to incremental maximum-finding is to extend the sorted sequences employed by the CRCW PRAM algorithm by additional information, as expressed in the definition of an R-structure below. A *range query* about a sequence $a_1, \ldots, a_N$ of integers specifies two integers k and l with $1 \leq k \leq l \leq N$ and asks for the position of the last occurrence of $\max_{k \leq i \leq l} a_i$ in the subsequence $a_k, \ldots, a_l$.

Definition Given a multiset S of N keys, an *R-structure* for S consists of

(1) The sorted sequence $b_1 \geq b_2 \geq \ldots \geq b_N$, where $S = \{b_1, \ldots, b_N\}$.

(2) For $N \geq 4$, a *range-query structure* Q that allows arbitrary range queries about the sequence $b_1 + 1, \ldots, b_N + N$ to be answered in $O(\log \log N)$ time with $O(\log N)$ processors.

It is easy to see that in the context of Condition (E) of Lemma 2.1, an R-structure for the full multiset $\{b_1^*, \ldots, b_s^*\}$ will allow us to finish the raking in $O(\log \log n)$ time. Except in the trivial case $s = 0$, we obtain such an R-structure by repeatedly combining R-structures for smaller multisets with the algorithm described in the following lemma.

Lemma 3.6 *Given R-structures for two multisets X and Y with $N = |X|$ and $M = |Y|$ elements, where $2 \leq M \leq N$, an R-structure for $X \cup Y$ can be constructed in $O(\log\log N + \log M)$ time on a CREW PRAM with $O(N(\log N)^2)$ processors.*

PROOF It is well-known that X and Y can be merged within the stated bounds to create item (1) of an R-structure for $X \cup Y$ [6]. Now each element in X or Y knows its position in the opposite sequence. We begin by creating new range-query structures Q'_X and Q'_Y for X and Y that are defined as the old structures Q_X and Q_Y, but with respect to sequences with the quantity $b_i + i$ replaced by $b_i + i'$, where i' is the position of the element in question in the sorted sequence $X \cup Y$. Q'_X and Q'_Y have a straightforward structure and simply consist of precomputed positions of the last range maxima within selected *standard ranges*, namely the whole range in question, its first and second halves, its four quarters, etc. It is easy to see that an arbitrary query interval can be decomposed into $O(\log N)$ disjoint standard ranges, so that the query can be answered in $O(\log\log N)$ time by combining the information associated with the relevant standard ranges. In the case of Q'_Y, the relevant information for the standard ranges can be computed from scratch in $O(\log M)$ time using M processors. For Q'_X, focus on one particular standard range and note that it decomposes into at most $M + 1$ *segments*, each consisting of consecutive elements of X with no intervening elements from Y. All elements within one segment increase their position by the same amount i in going from X to $X \cup Y$, namely by the number of smaller elements in Y. With $\Theta((\log N)^2)$ processors associated with each element of $X \cup Y$, we can devote $\Theta(\log N)$ processors to each segment, and we can easily arrange that these processors learn the value of i. The team of $\Theta(\log N)$ processors queries the old range-query structure Q_X about its associated segment, after which one processor in the team adds i and collaborates with one processor from every other segment within the standard range under consideration to compute the position of the last occurrence of the new overall maximum for that standard range. The queries need $O(\log\log N)$ time, and the subsequent combination of at most $M + 1$ answers takes $O(\log M)$ time.

Having Q'_X and Q'_Y, we can easily create a new structure of the same kind for $X \cup Y$. Each standard range of $X \cup Y$ is composed of the elements from one interval in X and the elements from one interval in Y, these intervals can be determined in constant time, and all that remains is to query about the intervals in Q'_X and Q'_Y and combine the two answers obtained. □

We will use the phrase "R-merging X and Y" to denote the procedure described in Lemma 3.6. The most noteworthy aspect of Lemma 3.6 is its time bound: When used to R-merge sets of comparable sizes, the algorithm of Lemma 3.6 offers no advantage whatsoever over sorting—the time bound is logarithmic. An edge over sorting is obtained only in heavily unbalanced R-merges, where one input set is much larger than the other one. It is interesting that this behavior can be put to good use.

Except for the fact that R-merges take the place of usual merges, our CREW PRAM raking algorithm is similar to the CRCW PRAM algorithm described in the previous subsection. The only other difference is a new definition of the generations $G_1, \ldots, G_q$. Let λ be a positive integer with $\lambda = \Theta(\log\log n)$ such that λ steps suffice to (1) sort any sequence of at most $K = \lceil \log n \rceil^3$ keys, and (2) R-merge any sequence of at most K keys with any sequence of at most n keys. Let g be the function from $\{a_1^*, \ldots, a_s^*\}$ to $\mathbb{N}$ given by

$$g(x) = \left\lfloor \log\left(\left\lfloor \frac{a_s^* - x}{3\lambda} \right\rfloor + 1 \right) \right\rfloor + 1,$$

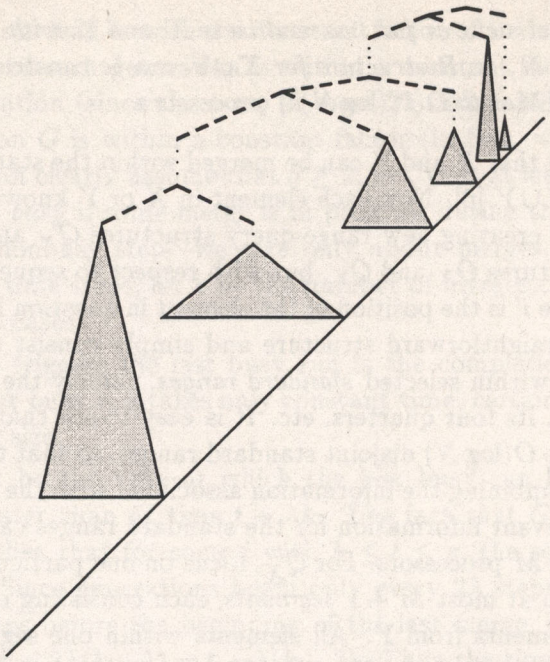

Fig. 3. Generations of geometrically increasing sizes.

for all $x \in \{a_1^*, \ldots, a_s^*\}$, and define G_i, for $i = 1, \ldots, q = g(a_1)$, as the set of those keys whose arrival time is mapped to i by g. Thus G_1 comprises the last 3λ families, G_2 spans the previous 6λ families, and so on. In general, for $i = 2, \ldots, q$, G_i spans twice as many families as G_{i-1}, except that G_q may be smaller (see Fig. 3).

We must demonstrate the existence of a family whose lead is within $O(\log \log n)$ of its log-size. Recall that a busy run is a maximal sequence of steps in which (R-)merging takes place. Take $k = 0$ if $q = 1$, and otherwise let k be the number of R-merges in the last busy run. We define a *final-merger* as follows: The generations $G_1, \ldots, G_k$ are final-mergers. Additionally, if $k = 0$ or the sorting of G_{k+1} finishes after that of G_k, G_{k+1} is a final-merger. Let G_r be a largest final-merger and consider two cases:

Case 1: $|G_r| \leq K$. In this case each final-merger can be sorted in at most λ steps, and every R-merge in the last busy run, if any, takes at most λ steps. We can conclude that $k \leq 1$, since otherwise, when the R-merging of G_k with H_{k+1} completes, the sorting of G_{k-1} has not yet begun, contradicting the fact that the last busy run is in progress. But then any family in G_1 has a lead of $O(\log \log n)$.

Case 2: $|G_r| > K$. Let L be the log-size of G_r. It suffices to show that the lead of G_r is $O(L \log \log n)$. To see this, note the following: First, since the total number of families is $O(\log n \log \log n)$, the log-size of G_r is within a constant factor of the log-size of a largest family in G_r. And second, the lead of G_r is at most three times that of any of its families if $r > 1$, while it is at most an additive $3\lambda = O(\log \log n)$ larger if $r = 1$. In particular, this allows us to assume that $q \geq 2$ and hence $k \geq 1$.

Since the size of G_r dominates that of any final-merger, every R-merge that takes place during the last busy run can be executed in $O(L)$ time. Because the total number of R-merges

is $O(\log\log n)$, the length of the final busy run is $O(L\log\log n)$. The last busy run is triggered by the completion of the sorting of a generation G_l whose size is dominated by that of G_r. Moreover, $l \geq r$, i.e., the sorting of G_r starts no earlier than that of G_l. Thus the sorting of G_r starts no earlier than $O(L)$ steps before the beginning of the last busy run, which implies that the lead of G_r is $O(L\log\log n)$.

Lemma 3.7 *For all integers $n \geq 4$, $\odot$-problems can be solved on n-node input trees in $O(\log n \log\log n)$ time on a CREW PRAM with $O(n(\log n)^2)$ processors.*

4 Achieving optimal speedup

In this section we show how the algorithms of the previous section can be modified to achieve optimal speedup. To a first approximation, we describe preprocessing that reduces the size of the input tree by any desired polylogarithmic factor. Part of the preprocessing evaluates small expression trees sequentially, for which reason we first consider this problem. Let $\mathcal{T}_0$ be the set of rooted trees in which each leaf is labeled with 0 or 1, and each internal node is labeled with one of the functions $\odot_0$ and $\odot_1$.

Lemma 4.1 *Every n-node tree in $\mathcal{T}_0$ can be evaluated sequentially in $O(n)$ time.*

PROOF The claim was proved in [21] for the case in which each leaf is labeled with 0 and each internal node is labeled with $\odot_0$. For the general case, choose for each internal node u of degree ≥ 2 in the given tree T a child v of u whose value is second-largest (i.e., if the values of the children of u are arranged into nonincreasing order, the integer occurring in the second position will be the value of v), and call v the *second-largest child* of u. We now process the nodes of T in postorder, at each internal node u computing the value of u from those of its children. By treating the maximum separately and using bucket sorting, we can carry out the processing at a node u of degree ≥ 2 in time proportional to the value of its second-largest child. It therefore suffices to show that the sum B of the values of all second-largest children in T is $O(n)$. To this end consider the forest F obtained from T by removing the edge between each second-largest child and its parent. As is easily seen by induction, the value of each node (with respect to T) is bounded by twice the number of its descendants in F. Thus $B \leq 2n$. □

Lemma 4.2 *Suppose that the value of a single leaf v in an n-node tree $T \in \mathcal{T}_0$ is considered as an indeterminate x (the values of all other leaves being integer constants). Then the value of T, as a function of x, can be computed in $O(n)$ time.*

PROOF We begin by evaluating all trees in the forest obtained by removing all nodes and edges on the simple path π in T from its root to v. Using the algorithm of Lemma 4.1, this can be done in $O(n)$ time. We then sort the values obtained lexicographically, with the parent on π of the corresponding root as the primary key and the value itself as the secondary key; this can be done in $O(n)$ time using radix sorting. It produces sorted arrays, one for each node on π other than v, with the aid of which we can shrink T to the path π, without changing the value of the root, by associating appropriate functions in $\mathcal{F}$ with the nodes on π according to Lemma 3.2. The time to do this is proportional to the total size of all sorted arrays, which is certainly $O(n)$.

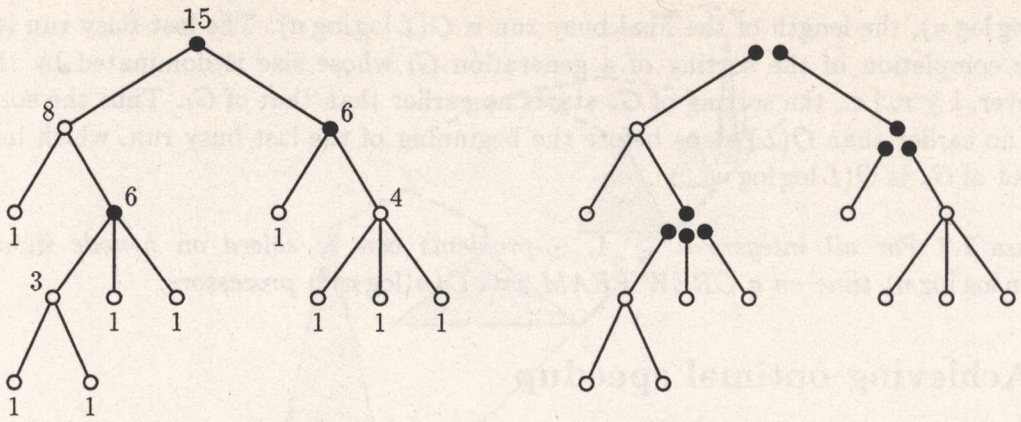

Fig. 4. A tree, its 5-critical nodes, and its partition into two edge bridges and five leaf bridges.

Finally, in time proportional to the length of π, we can compose the functions associated with its nodes according to Lemma 3.1 to obtain the final result as a function in $\mathcal{F}$. □

Another concept central to the preprocessing is that of m-bridges, as introduced by Gazit *et al.* [12]. Given a positive integer m and a rooted tree T, define the *m-rank* of a node u in T as $\lceil w/m \rceil$, where w is the number of descendants of u, and say that an internal node u is *m-critical* if its m-rank differs from that of each of its children. An *m-bridge* (for brevity: a *bridge*) in T is a subtree of T spanned by a maximal set of edges, any two endpoints of which can be joined by a path in T without m-critical nodes in its interior (see Fig. 4). An *attachment* of a bridge W is a node shared between W and another bridge; it is necessarily m-critical. Gazit *et al.* prove that a bridge can have at most one leaf attachment and that, except possibly for its root, it has no other attachments. An bridge is called an *edge bridge* if it has a leaf attachment, and a *leaf bridge* otherwise. Gazit *et al.* also show that if T has n nodes and $1 \leq m \leq n$, then the number of m-critical nodes and, hence, the number of edge bridges is bounded by $2n/m$, and that the *size* of every m-bridge W, defined as the number of nodes in W, is bounded by $m + 1$. Finally, they argue that the set of m-critical nodes in an n-node tree can be computed in $O(\log n)$ time using $O(n)$ operations on an EREW PRAM by means of the Euler-tour technique [22] and optimal list ranking [10, 4, 18]. The same method allows us to select a unique *leading edge* from each m-bridge and to mark it with the size of the bridge.

Assume that we are given an input tree $T \in \mathcal{T}_0$ with $n \geq 4$ nodes and maximum node degree d. In order to obtain an EREW PRAM algorithm with optimal speedup, we choose m as a positive integer with $m = \Theta(\log n \log(d + 1))$ and carry out the computation of m-bridges described above. Using the information associated with the leading edges to distribute the work among the processors, we then process each bridge W sequentially, contracting W while preserving the attachments of W and their labels as well as the value of the full tree. This is done using the algorithm of Lemma 4.1 if W is a leaf bridge, and using that of Lemma 4.2 if W is an edge bridge. Let r be the root of W. If r is not an attachment, we apply Lemma 4.1 or 4.2 to all of W, labeling r with the resulting value or function in $\mathcal{F}$; besides r, the only node in the resulting tree is the leaf attachment of W, if any. If r is an attachment, it has a single child u in W. Unless u is an attachment (in which case W contains only the two nodes r and u), we apply Lemma 4.1 or 4.2 to the maximal subtree of W rooted at u, labeling u with the

resulting value or function in $\mathcal{F}$; besides r and u, the only node in the resulting tree is the leaf attachment of W, if any. This uses $O(m)$ time and $O(n)$ operations and yields a tree T' with at most n nodes and the same value as T. Moreover, since every nonroot internal node in T' is m-critical in T or has an m-critical child, the number of internal nodes in T' is $O(n/m)$. Define a *phantom* as a leaf of T'. For each internal node in T', we sort the values of its phantom children into nondecreasing order. Since all of these values are bounded by $m+1$, as is easily seen by induction, this can be done in $O(m)$ time using $O(n)$ operations [9, 23]. For each internal node u in T', we then conceptually remove all of its phantom children, while replacing the function labeling u by its corresponding projection. In terms of the representation, what we actually do to the label of u is nothing if u is left with two or more (nonleaf) children (i.e., in that case the projection of its associated function is represented in a "raw" form by the sorted sequence of the values of its phantom children). If u is left with exactly one child, we label it with the appropriate function in $\mathcal{F}$, and if it becomes a leaf, we label it with its value. All of this can be done in $O(\log n)$ time using $O(n)$ operations. The resulting tree T'' has internal nodes labeled not just with $\odot_0$ and $\odot_1$, but also with projections of these functions; still, it is easy to see that Conditions (A)–(C) of Lemma 2.1 continue to hold for this more general set Ω of operators. T'' has $O(n/m)$ nodes, and we evaluate it using the algorithm of Lemma 3.3, with one difference: Suppose that the raking at an internal node u is about to finish, so that a sorted sequence of the values of all except one of the children of u in T'' is available. We need to merge this list with the sorted sequence of the values of the phantom children of u, after which we can proceed to add appropriate offsets, compute the position of the last occurrence of the maximum and finish the raking, now with the phantom children of u taken appropriately into account. The computation just described can be carried out in $O(\log(d+1))$ time, as required, but not necessarily using only the processors "provided" by Condition (E) of Lemma 2.1 (since their number is independent of the number of phantom children of u). We thus have to allocate additional processors to u to help with the raking. The computational effort is $O(1)$ per phantom involved, so that the total number of operations needed to process all phantoms is $O(n)$. The allocation of the additional processors essentially reduces to sorting the phantoms by their raking times, while keeping sibling phantoms with the same raking time together and in the same order, and can be done "off-line" in $O(m)$ time using $O(n)$ operations before the processing of T''. Because of the use of additional processors, we are not in full compliance with Condition (E) of Lemma 2.1. It is easy to see, however, that the lemma continues to hold even if such additional processors are employed, provided that they are accounted for in the total resource requirements. Thus we have proved:

Theorem 4.1 *For all integers $n \geq 2$ and $d \geq 1$, $\odot$-problems can be solved on input trees with n nodes, maximum degree d and all leaf labels bounded by 1 on an EREW PRAM using $O(\log n \log(d+1))$ time and $O(n)$ operations.*

For the CREW PRAM, we choose $m = \Theta((\log n)^6)$, compute m-bridges as in the EREW PRAM algorithm, and subsequently process each bridge using the EREW PRAM algorithm. This again means contracting each bridge W, while preserving the attachments of W and their labels as well as the value of the full tree. In the case of leaf bridges this can be done using the algorithm of Theorem 4.1. We have not proved a lemma corresponding to Lemma 4.2 to be used for the edge bridges. It is not difficult to see, however, that the algorithm of Theorem 4.1 performs as desired if the leaf attachment of an edge bridge under consideration is treated as an

internal node with several children; alternatively, one can proceed as in the proof of Lemma 4.2, applying the algorithm of Theorem 4.1 separately to each subtree hanging from the path π and subsequently contracting π by means of pointer doubling. The processing of all m-bridges uses $O((\log\log n)^2)$ time and $O(n)$ operations and produces a tree T' with at most n nodes, $O(n/m)$ internal nodes and the same value as T. Moreover, since the leaf values in T' are the values of subtrees of disjoint m-bridges, each such leaf value is bounded by $m+1$, and the sum of all leaf values is at most n. Let U be the set of internal nodes in T'. As in the EREW PRAM algorithm, we define a phantom as a leaf in T' and sort the phantom children of each node in U by their values (now using radix sorting in 6 passes). For each $u \in U$, we then replace all phantom children of u with a common value by a *packed phantom.* If a packed phantom v is formed out of r phantoms with a common value of b, we call b and r the *value* and the *multiplicity* of v, respectively, and represent v via the triple (b, r, R), where R is the sum of the multiplicities of all packed phantoms with the same parent as v and values of at least b; in other words, in the sorted sequence of packed phantom siblings, the third components are the prefix sums of the second components. The conversion of (simple) phantoms to packed phantoms can be done in $O(\log n)$ time using $O(n)$ operations.

Let $\mathcal{P}$ be the set of nonempty sequences $(b_1, r_1, R_1), \ldots, (b_k, r_k, R_k)$ of triples of nonnegative integers with $b_1 \geq b_2 \geq \cdots \geq b_k$ and $R_i = \sum_{j=1}^{i} r_j$, for $i = 1, \ldots, k$. We view a sequence in $\mathcal{P}$ derived from a nonincreasing sequence S of integers, as described above for the case of a sequence of phantom values, as just a convenient way of representing S. This allows us, e.g., to apply the operators $\odot_0$ and $\odot_1$, originally defined on $\mathbb{N}_0^+$, also to sequences in $\mathcal{P}$; it is easy to see that for all sequences $(b_1, r_1, R_1), \ldots, (b_k, r_k, R_k)$ in $\mathcal{P}$ and for $\mu \in \{0, 1\}$,

$$\odot_\mu((b_1, r_1, R_1), \ldots, (b_k, r_k, R_k)) = \max_{1 \leq i \leq k} (b_i + R_i - \mu).$$

One can also observe that for all $N \in \mathbb{N}$ and all fixed $\epsilon > 0$, two sequences in $\mathcal{P}$ of length N each can be merged (to create another sequence in $\mathcal{P}$) in constant time on a CREW PRAM with $O(N^{1+\epsilon})$ processors: It suffices to let each triple carry out a $\Theta(N^\epsilon)$-way search for the value of its first component in the opposite sequence, after which it can deduce its position in the output sequence as well as its new third component in constant time; note that we make no attempt to coalesce adjacent triples with identical first components.

The final step is to apply the algorithm of Lemma 3.7 to T', for which we have more than enough processors. As in the case of the EREW PRAM algorithm, we must describe how to incorporate phantom leaves at the time of a raking. Note first in general that if k, l and h are positive integers with $l \leq k$ and $a_1, \ldots, a_k, b_1, \ldots, b_h$ are nonnegative integers with $a_1 \geq a_2 \geq \cdots \geq a_k$ and $a_l \geq b_1 \geq b_2 \geq \cdots \geq b_h$, then for $\mu \in \{0, 1\}$,

$$\odot_\mu(a_1, \ldots, a_k, b_1, \ldots, b_h) = \max\{\odot_\mu(a_1, \ldots, a_k), l + \odot_\mu(a_{l+1}, \ldots, a_k, b_1, \ldots, b_h)\}.$$

We will use this in a situation where $a_1, \ldots, a_k$ is the sorted sequence of values of the nonphantom children of a node $u \in U$, while $b_1, \ldots, b_h$ is the sorted sequence of values of the phantom children of u. Since $b_1 \leq m+1$, the observation shows that it suffices to combine the phantom leaves with the subsequence $a_{l+1}, \ldots, a_k$ of nonphantom leaves with values bounded by m; we can also assume that $k > l$. We begin by converting the sequence $a_{l+1}, \ldots, a_k$ to a sequence in $\mathcal{P}$ of length $q = \min\{k - l, m + 1\}$. If $k - l \leq m + 1$, the sequence $a_{l+1}, \ldots, a_k$ is augmented with second and third components and used as is; otherwise we compute the rank in the sequence

$a_{l+1}, \ldots, a_k$ of each integer in $\{0, \ldots, m\}$ that occurs in the sequence and extend the information to the remaining values by means of segmented broadcasting. Since $m = (\log n)^{O(1)}$, this can be done in $O(\log \log n)$ time with $k - l$ processors, which are certainly available.

Suppose that the sequence $b_1, \ldots, b_h$ is represented by a sequence in $\mathcal{P}$ of length r. We will show that $\Omega((q+r)^{4/3})$ processors can be devoted to the raking. Then the packed representations of the sequences $a_{l+1}, \ldots, a_k$ and $b_1, \ldots, b_h$, of lengths q and r, can be merged in constant time, as described above, after which the raking can be finished in $O(\log \log n)$ time by computing maxima in sequences of length $O(q + r) = O(m) = (\log n)^{O(1)}$. To see that the required number of processors is available, first recall that $|U| = O(n/(\log n)^6)$, so that we can associate $\Theta((\log n)^4)$ processors with each edge in T' without using more than $O(n/(\log n)^2)$ processors. In particular, this provides $\Omega(q(\log n)^4)$ processors for the raking under consideration. Since $q \leq m + 1 = O((\log n)^6)$, this number is always $\Omega(q^{4/3})$, and it is $\Omega(r^{4/3})$ unless $r \geq (\log n)^3$, which we can therefore assume. For each $u \in U$, let r_u be the number of packed phantom children of u. The values of the packed phantom children of u, being distinct, must sum to at least $0+1+\cdots+(r_u-1) = r_u(r_u-1)/2$. Since the sum of the values of all phantoms is bounded by n, it follows that $\sum_{u \in U} r_u^2 = |U| + \sum_{u \in U} r_u(r_u - 1) \leq n + 2n = O(n)$. Thus, for all $u \in U$ with $r_u \geq (\log n)^3$, we can allocate $\Omega(r_u^2/(\log n)^2) = \Omega(r^{4/3})$ processors to u without using more than $O(n/(\log n)^2)$ processors altogether, which is what was to be shown.

This completes the description of an algorithm for the CREW PRAM with optimal speedup and a running time of $O(\log n \log \log n)$. In order to lower the running time to $O(\log n)$ for the CRCW PRAM, it suffices to provide constant-time algorithms for the two tasks that need more time without concurrent writing, namely segmented broadcasting and computing the maximum. In both cases, the number of processors available to solve problems of size N is $\Omega(N^{4/3})$, so that very simple solutions are possible. For segmented broadcasting, a stronger result was already used in the proof of Lemma 3.4, while computing the maximum was described in [20]. Note that when the algorithm of Lemma 3.5, applied to the contracted tree T', merges two sequences containing a total of N keys, then the maximum key value involved is at most $(m + 1)N = N(\log n)^{O(1)}$, so that the algorithm is indeed applicable; this is because, for each node u in T', the maximal subtree of T rooted at u contains at most $m + 1$ times as many nodes as the maximal subtree of T' rooted at u.

Theorem 4.2 *For all integers $n \geq 4$, $\odot$-problems can be solved on n-node input trees with all leaf labels bounded by 1 using $O(n)$ operations and either $O(\log n \log \log n)$ time on a CREW PRAM or $O(\log n)$ time on a CRCW PRAM.*

Appendix: Proof of the tree-contraction lemma

This appendix provides a proof of Lemma 2.1, reproduced below for the reader's convenience. The lemma was essentially shown by Miller *et al.* [15, 16, 12, 17], in a somewhat less general form, but we believe an explicit statement and proof of the lemma to be useful.

Lemma 2.1 *Let D be a set, Ω a set of commutative functions from D^+ to D, and $\mathcal{T}$ a subset of $\mathcal{T}_{D,\Omega}$. If there are τ, p and $\mathcal{F}$ satisfying Conditions (A)-(E) below, then, for all integers $n \geq 2$, every n-node tree $T \in \mathcal{T}$ can be evaluated in $O(\tau(T) \log n + \log(p(T)))$ time on a PRAM with $np(T)$ processors.*

(A) $\mathcal{F}$ *is a set of functions from* D *to* D*, and for all* $f \in \mathcal{F}$ *and all* $x \in D$*,* $f(x)$ *can be computed from* f *and* x *in constant time with one processor.*

(B) For any two functions f *and* g *in* $\mathcal{F}$*,* $g \circ f$ *belongs to* $\mathcal{F}$ *and can be computed from* f *and* g *in constant time with one processor.*

(C) Every 1-dimensional projection of a function in Ω *belongs to* $\mathcal{F}$*.*

(D) τ *and* p *are functions from* $\mathcal{T}$ *to* $\mathbb{N}$ *such that for all integers* $n \geq 2$ *and all* n*-node trees* $T \in \mathcal{T}$*,* $\tau(T)$ *and* $p(T)$ *can be computed from* T *in* $O(\tau(T) \log n)$ *time with* $np(T)$ *processors.*

(E) Fix an input tree $T^* \in \mathcal{T}$ *with* $n \geq 2$ *nodes, let* $\tau^* = \tau(T^*)$ *and* $p^* = p(T^*)$ *and consider the following setting: A sequence* $a_1^*, \ldots, a_s^*$ *of* s *integers, where* s *is a nonnegative integer bounded by the maximum degree of a node in* T^* *and where* $1 \leq a_1^* \leq a_2^* \leq \cdots \leq a_s^* = O(\tau^* \log n)$*, is available for preprocessing in* $O(\tau^* \log n)$ *time with* s *processors. Assume that the preprocessing finishes at time 0 (this is just a convention for fixing the origin of the time axis). Subsequently, at time* a_i^**, for* $i = 1, \ldots, s$*, the value* b_i^* *of a node* r_i *in* T^* *becomes known, and* $p^* n_i$ *processors numbered* $p^* \sum_{j=1}^{i-1} n_j + 1, \ldots, p^* \sum_{j=1}^{i} n_j$ *become available, where* n_j *is the number of nodes in the maximal subtree* T_j^* *of* T^* *rooted at* r_j*, for* $j = 1, \ldots, s$*. Thus every value that becomes known contributes new processors, and the available processors at all times are consecutively numbered. Moreover, the trees* $T_1^*, \ldots, T_s^*$ *are disjoint.*

In these circumstances, for some constant $C > 0$ *and for all* $\diamond \in \Omega$*, the function* $f : D \to D$ *mapping* x *to* $\diamond(b_1^*, \ldots, b_s^*, x)$*, for all* $x \in D$*, must be computable with the available processors to be ready by time*

$$\max_{a_1^* \leq j \leq a_s^*} (j + C\tau^* \log(|F_j| + 2)),$$

where $F_j = \{i : 1 \leq i \leq s \text{ and } a_i^* = j\}$*, for* $j = a_1^*, a_1^* + 1, \ldots, a_s^*$*; for* $s = 0$ *we take this condition to mean that* f *must be computable in constant time with one processor.*

PROOF We assume that the input tree is given according to a standard adjacency-list representation, where each node has an adjacency list with an entry for each of its children, and each nonroot node has a pointer to its entry in the adjacency list of its parent. When merging a node v with its parent u, we will identify the resulting node with the parent u—to emphasize this, we also denote the operation as "merging v into u". In terms of the representation, we simply remove v after appending its adjacency list to that of u.

Let $T^* = (V^*, E^*)$ be the input tree and take $n = |V^*|$, $\tau^* = \tau(T^*)$ and $p^* = p(T^*)$. We construct a new tree $T_0 = (V_0, E_0)$ as follows (see Fig. 5): For each node $v \in V^*$ with k children $v_1, \ldots, v_k$, we introduce k new nodes $u_1, \ldots, u_k$, make u_i the parent of v_i, for $i = 1, \ldots, k$, make u_{i-1} the parent of u_i, for $i = 2, \ldots, k$, and make v the parent of u_1. This is very easy to do; in fact, it can be viewed largely as adopting a new interpretation of the adjacency list of v. We will call the original nodes in V^* "black", while the new nodes in $V_0 \backslash V^*$ are "white".

We now associate a processor with each node in T_0 and *contract* T_0 in a sequence of *stages*, ending when only one node remains. We will use "T" to denote the evolving tree, reserving "T_0"

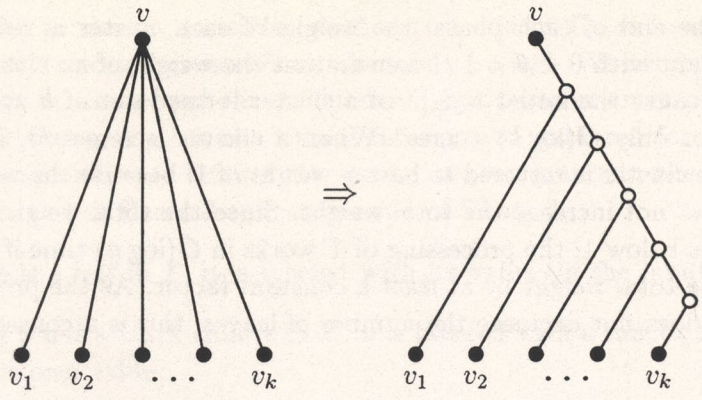

Fig. 5. The transformation from T^* to T_0.

for its original value. A stage ends, in general, with a number of nodes grouped into disjoint *clusters*; initially, there are no clusters. We will call a node *free* if it does not belong to a cluster. Each stage takes constant time and comprises the following four *phases*:

Phase 1: Cluster formation;
Phase 2: Cluster processing;
Phase 3: Cluster removal;
Phase 4: Leaves-cutting.

In Phase 1 (Cluster formation), certain free nodes are grouped into new clusters. Say that a *bond* exists between a node v and its parent u if they are both of degree ≤ 1 and both free and if either they are both white, or u is black and v does not have a white child (i.e., either it is a leaf, or it has a black child). Each node incident on one or two bonds becomes part of a new cluster, the other members of the cluster being those nodes that it can reach via one or more bonds. Note that no global information about a cluster is computed; each node merely determines whether it belongs to a cluster and, if so, which of its neighbors belong to the same cluster.

In Phase 2 (Cluster processing), each cluster carries out the task of repeatedly merging pairs of adjacent nodes in the cluster until just a single node remains, at which point the cluster (but not the remaining node) is removed in Phase 3 (Cluster removal). In each stage, however, only a constant number of steps of the task are executed, which is the reason why clusters may survive from one stage to the next. It is well-known and easy to see that the task of a cluster with k nodes can be carried out in $O(\log k)$ time, viewing the cluster as a linked list and performing repeated pointer doubling; as observed in [9, 12], concurrent reading during the pointer doubling can be avoided by letting each node maintain an indication of whether it points to the last element of the list. In Phase 4 (Leaves-cutting), every free nonroot leaf is merged into its parent.

Since only nodes of degree ≤ 1 are merged into their parents, it is easy to see that no node degree ever increases. Thus the degree of every white node remains bounded by 2, and the degree of every black node remains bounded by 1. We already used this implicitly above by assuming that all free leaves can be merged into their parents in constant time.

As an aid in analyzing the number of steps needed by the algorithm described above to contract T to a single node, we attach conceptual weights to nodes and clusters. Every free node is of weight 1. When k nodes form a cluster, their combined weight of k is transferred to

the cluster. At the end of each phase, the weight of each cluster is reduced by a factor of θ, where θ is a constant with $0 < \theta < 1$ chosen so that the weight of no cluster ever drops below 1; this is possible because the initial weight of a cluster formed out of k nodes is $k \geq 2$, while the cluster survives for only $O(\log k)$ stages. When a cluster is removed, finally, the single node emerging from the cluster is restored to have a weight of 1; because the weight of the cluster was at least 1, this does not increase the total weight. Since the total weight is $2n - 1$ initially and clearly never drops below 1, the processing of T works in $O(\log n)$ time if we can show that each stage decreases the total weight by at least a constant factor. As the processing and removal of clusters certainly does not decrease the number of leaves, this is a consequence of the following claim.

Claim *At the beginning of Phase 2 (Cluster processing) in every stage, at least* $1/20$ *of the total weight is contributed by leaves and clusters.*

PROOF Define a *bad chain* to be a maximal sequence $v_1, \ldots, v_k$ of free degree-1 nodes such that v_{i-1} is the parent of v_i, for $i = 2, \ldots, k$. At the beginning of Phase 2, a bad chain can contain at most 4 nodes, since any sequence of 5 or more nodes must contain two consecutive white nodes, three consecutive black nodes, or a white node between two black nodes; in each case, a bond would exist in Phase 1.

Consider a tree T' derived from the current tree T by merging the nodes in each cluster to form a single *cluster node* (i.e., finishing the task of the cluster), merging the nodes in each bad chain into a *bad node*, and finally merging each node of degree ≥ 2 with a bad node as its parent into its parent. The weight of a cluster node is the weight of the corresponding cluster, and the weight of a bad node is the total weight of the nodes from which it was formed.

Let U be the set of leaves and cluster nodes in T'. We must show that at least $1/20$ of the total weight (in T', and therefore also in T) is contributed by nodes in U. Since the average degree in T' is < 1, as in every tree, more than half of the nodes in T' belong to the set U_1 of nodes of degree ≤ 1. $U \subseteq U_1$, and every node in $U_1 \backslash U$ is the parent of a node in U. Thus $|U| \geq |U_1 \backslash U|$, which implies that U contains at least one quarter of the nodes in T'. Since every node in U is of weight at least 1, while the weight of no node exceeds 5, the claim follows. □

The procedure above, just shown to work in $O(\log n)$ time, serves exclusively to gather timing information for a second run, still to be described, which is augmented with computation that actually evaluates the input tree T^*. During the first run, whenever a black node is merged into its parent, it is marked with the time at which this happens, called its *completion time*, where we assume the origin of the time axis placed at the start of the procedure. We will actually slow the procedure down by a factor of $c\tau^*$, where $c \in \mathbb{N}$ is a suitably large constant to be chosen later; we here use Condition (D) of Lemma 2.1. This has the effect of multiplying all completion times by $c\tau^*$. After the first run, for each node u in T^*, we sort the children of u by their completion times [9, 23], after which we consider T^* to be an ordered tree. We use the Euler-tour technique as described in [22] to determine, for each node $u \in V^*$, the smallest and the largest preorder number in T^* of a descendant of u. For each node u in T^* with $k \geq 1$ children, the sorting provides us with an array $A_u[1..k]$ of the sorted completion times of the children of u. Reinterpreting these as arrival times, in the sense of Condition (E) of Lemma 2.1, and ignoring the last entry, we have one half of the input to a raking problem of size $s = k - 1$ associated with u. We describe below how to obtain the other half of the input.

In addition to the processor associated with each node in T_0, the second run of the tree contraction employs np^* *special processors* numbered $1, \ldots, np^*$. We assume that the additional processing carried out in the second run is "hidden", by means of extra processors and/or the slowdown mentioned above, so that the timing information gathered for the first run will be accurate also for the second run. During the second run we maintain the following invariants: In every stage, at the end of Phase 2 and at the end of Phase 4,

(1) If a black node is a leaf in T, it is labeled with its value (in the input tree T^*);

(2) If a black node u has a black child v in T, u is labeled with a function in $\mathcal{F}$ that maps the value of v to its own value.

If we copy leaf labels from T^* to T_0, Invariant (1) is satisfied initially, and Invariant (2) holds vacuously. If Invariant (1) holds at the end of the second run, the root of T (and therefore of T^*) will be labeled with its value, and we are done. In order to maintain the invariants, we augment the first run with the following steps:

(a) Whenever a black leaf v merges into a black parent u, the function f labeling u (Invariant (2)) is applied to the value x labeling v (Invariant (1)), and $f(x)$ is attached as the new label of u, thus maintaining Invariant (1). By Condition (A) of Lemma 2.1, this can be done in constant time.

(b) Whenever a black nonleaf v merges into a black parent u, the rules for the formation of clusters imply that v has a black child. Thus u and v are labeled with functions f_u and f_v, respectively (Invariant (2)). We replace the label of u by $f_u \circ f_v$, maintaining Invariant (2). By Condition (B), this can be done in constant time.

(c) Whenever a black node v merges into a white parent, the rules for the formation of clusters imply that v is necessarily a leaf. Thus the value of v is known (Invariant (1)). We supply the value of v as an input key to an ongoing raking computation associated with the parent u of v in T^*. Moreover, if the smallest and largest preorder numbers of a descendant of v in T^* are l_1 and l_2, respectively, we dedicate the special processors numbered $(l_1 - 1)p^* + 1, \ldots, l_2 p^*$ to this task. When the raking finishes, we label u with the resulting function which, by Condition (C), belongs to $\mathcal{F}$.

In Step (c), suppose that v is the ith child of u in T^*. It is easy to see that if we take T_i^* to be the maximal subtree of T^* rooted at v, then the number of processors dedicated to the raking in Step (c) above is precisely as required by Condition (E). Furthermore, the trees $T_1^*, \ldots, T_s^*$ indeed are disjoint, and since they receive preorder numbers in the order in which their roots complete, the special processors allocated to the raking at all times are consecutively numbered, as required; they are not numbered starting at 1, but this is easy to take care of.

Steps (a)–(c) above specify no action in the case of a white node merging into another node. The operation may create a black leaf or a give a black node a black child, however, so that we must show the invariants to be satisfied. Say that a node u in T *contains* a node $v \in V_0$ if either $u = v$, or (recursively) a node containing v at some point merged into u. A simple yet useful observation, easy to prove by induction on the number of merges involved in deriving T from T_0, is that if u and v are nodes in T, then u is an ancestor of v in T if and only if it is an

ancestor of v in T_0. Another observation of this kind is that every node u in T_0 is contained in the closest ancestor of u in T_0 that still belongs to T.

Suppose that at some time t, a white node merging into a black node u causes u to acquire a black child v. Let U be the set of proper descendants of u in T_0 that are not descendants of v, and let Z be the set of (black) children of u in T^* that are not ancestors of v. By the first observation made above, no node w in U can still be present in T at time t, since it could not be a descendant of u without being a descendant of v. Since this holds for all nodes w in U, the second observation implies that at time t, all nodes in U are contained in u. A node in Z has no black proper ancestor that is also a proper descendant of u. Since it cannot merge into u (then u could never again acquire a white child, contrary to the fact that it has one at time t), it must merge into a white node in U, at which point its value is supplied to the raking problem associated with u. Suppose that this happens simultaneously in the step immediately before time j for a nonempty group F_j of nodes in Z. Since nodes merge only in disjoint pairs (or triples, if we allow two leaves to merge simultaneously with a common parent) and at least $|F_j|/2$ white nodes in U are still present in T at time j, we must have $t \geq j + c'c\tau^* \log(|F_j|+2)$, for some constant $c' > 0$ (recall that we slow down the contraction of T by a factor of $c\tau^*$). Since this holds for each integer j in the range $a_1^*, \ldots, a_s^*$, where a_1^* and a_s^* are the first and last completion times of a node in Z, Condition (E) guarantees that with the constant c chosen suitably (in dependence of C), the raking associated with u will finish before time t. The function computed by the raking is precisely as required by Invariant (2), which can therefore be satisfied.

Suppose now instead that at time t, a white node merging into a black node u makes u a leaf. An argument very similar to the one above, taking U as the set of proper descendants of u in T_0 and Z as the set of (black) children of u in T^*, shows that before time t, there will have been enough time to compute a function in $\mathcal{F}$ mapping the value of the last child v of u in T^* to the value of u. Invariant (1) and Condition (A) imply that when v becomes a leaf, constant time suffices to label u with its value, maintaining Invariant (1).

If a special processor is dedicated to the raking at two distinct (black) nodes u and w, then one of the two nodes is an ancestor in T^* of the other one. Suppose that u is an ancestor of w in T^* and let v be the child of u in T^* that is also an ancestor of w. As argued above, the raking at w finishes before w can acquire a black leaf or become a leaf itself. In particular, when the raking at w finishes, w still exists as a node in T. On the other hand, a special processor participating in the raking at w is not needed in the raking at u before v merges into a white parent, at which point v must be a leaf and w, if different from v, must have merged into a parent. Thus no special processor is simultaneously dedicated to different raking operations.

An issue that was ignored above is how to inform a special processor of the raking problem on which it is supposed to work at any given time. If concurrent reading is allowed, we can simply associate the special processors numbered $(i-1)p^*+1, \ldots, ip^*$ with the node v_i of preorder i in T^*, for $i = 1, \ldots, n$, and let each of these processors keep track of the node in T containing v_i. For the EREW PRAM, we offer the following more complicated solution:

Divide the np^* special processors into n *teams* of p^* consecutively numbered processors each. All members of a team are assigned to the same $O(\log n)$ raking problems. Thus if one processor in the team is informed of the identities of these problems, it can communicate this information to the other members of the team in $O(\log n + \log p^*)$ time by distributing it in a pipelined fashion via a binary tree—this is where the EREW PRAM algorithm may need more than $O(\tau^* \log n)$ time. We can therefore assume, in the interest of simplicity, that $p^* = 1$, so that each special

processor forms a team by itself. Divide the n special processors into $O(n/\log n)$ *groups* of $O(\log n)$ consecutively numbered processors each and choose a *leader* from each group. Each black node v that merges into a white parent generates the triple (l_1, l_2, t), where l_1 and l_2 are the smallest and largest preorder numbers of a descendant of v, and t is the completion time of v. The remaining task is to distribute each such triple, called a *message*, to the special processors numbered $l_1, \ldots, l_2$. The first step is to send each message (l_1, l_2, t) to all leaders of groups containing at least one processor whose processor number lies in the range $l_1, \ldots, l_2$. To this end, each message is replicated into as many copies as it has (leader) recipients. Subsequently the message copies are sorted by their destinations. We here use the fact that there are n black nodes, and hence $O(n)$ copies, to conclude that the sorting can be carried out in $O(\log n)$ time. Subsequently each group leader is informed of the positions in the sorted sequence of the first and last messages destined for it. Possibly aided by auxiliary processors, each group leader proceeds to "expand" each message that it received into individual messages for all concerned processors in its group, after which the individual messages within each group are sorted by their destinations. Since there are $O(n \log n)$ individual messages and $O(\log n)$ destinations within each group, this can be done in $O(\log n)$ time with the available processors [9, 23]. Finally each special processor is informed of the positions of the $O(\log n)$ messages destined for it. □

Acknowledgment. We thank Wojtek Plandowski for pointing out the similarity between register allocation and broadcasting.

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka, A simple parallel tree contraction algorithm, *J. Algorithms* **10** (1989), pp. 287–302.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[3] M. Ajtai, J. Komlós, and E. Szemerédi, An $O(n \log n)$ sorting network, *In* Proc. 15th Annual ACM Symposium on Theory of Computing (STOC 1983), pp. 1–9.

[4] R. J. Anderson and G. L. Miller, Deterministic parallel list ranking, *In* Proc. 3rd Aegean Workshop on Computing (AWOC 1988), Springer Lecture Notes in Computer Science, Vol. 319, pp. 81–90.

[5] O. Berkman and U. Vishkin, On parallel integer merging, *Inform. and Comput.* **106** (1993), pp. 266–285.

[6] A. Borodin and J. E. Hopcroft, Routing, merging, and sorting on parallel models of computation, *J. Comput. System Sci.* **30** (1985), pp. 130–145.

[7] S. Chaudhuri and T. Hagerup, Prefix graphs and their applications, *In* Proc. 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994), Springer Lecture Notes in Computer Science, Vol. 903, pp. 206–218.

[8] R. Cole, Parallel merge sort, *SIAM J. Comput.* **17** (1988), pp. 770–785.

[9] R. Cole and U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control* **70** (1986), pp. 32–53.

[10] R. Cole and U. Vishkin, Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM J. Comput.* **17** (1988), pp. 128–142.

[11] F. E. Fich, P. Ragde, and A. Wigderson, Relations between concurrent-write models of parallel computation, *SIAM J. Comput.* **17** (1988), pp. 606–627.

[12] H. Gazit, G. L. Miller, and S.-H. Teng, Optimal tree contraction in the EREW model, In *Concurrent Computations: Algorithms, Architecture, and Technology*, S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz (eds.), Chap. 9, pp. 139-156, Plenum Press, New York, 1988.

[13] S. R. Kosaraju and A. L. Delcher, Optimal parallel evaluation of tree-structured computations by raking, *In* Proc. 3rd Aegean Workshop on Computing (AWOC 1988), Springer Lecture Notes in Computer Science, Vol. 319, pp. 101–110.

[14] G. L. Miller and J. H. Reif, Parallel tree contraction and its application, *In* Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS 1985), pp. 478–489.

[15] G. L. Miller and J. H. Reif, Parallel tree contraction, Part 1: Fundamentals, preprint, 1987. The final version (not available to us) appeared in *Randomness and Computation*, Advances in Computing Research, Vol. 5, S. Micali (ed.), pp. 47–72, JAI Press, Greenwich, CT, 1989.

[16] G. L. Miller and J. H. Reif, Parallel tree contraction, Part 2: Further applications. *SIAM J. Comput.* **20** (1991), pp. 1128-1147.

[17] G. L. Miller and S.-H. Teng, Tree-based parallel algorithm design, manuscript. A preliminary version appeared in Proc. 2nd International Conference on Supercomputing (1987), pp. 392–403.

[18] M. Reid-Miller, G. L. Miller, and F. Modugno, List ranking and parallel tree contraction. In *Synthesis of Parallel Algorithms*, J. H. Reif (ed.), Chap. 3, pp. 115–194, Morgan Kaufmann Publ., San Mateo, CA, 1993.

[19] R. Sethi, Complete register allocation problems, *SIAM J. Comput.* **4** (1975), pp. 226–248.

[20] Y. Shiloach and U. Vishkin, Finding the maximum, merging, and sorting in a parallel computation model, *J. Algorithms* **2** (1981), pp. 88–102.

[21] P. J. Slater, E. J. Cockayne, and S. T. Hedetniemi, Information dissemination in trees. *SIAM J. Comput.* **10** (1981), pp. 692–701.

[22] R. E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14** (1985), pp. 862–874.

[23] R. A. Wagner and Y. Han, Parallel algorithms for bucket sorting and the data dependent prefix problem, *In* Proc. International Conference on Parallel Processing (ICPP 1986), pp. 924–930.