# Efficient parallel algorithms for tree accumulations

Jeremy Gibbons, Wentong Cai and David B. Skillicorn

Abstract. Accumulations are higher-order operations on structured objects; they leave the shape of an object unchanged, but replace elements of that object with accumulated information about other elements. Upwards and downwards accumulations on trees are two such operations; they form the basis of many tree algorithms. We present two Erew Pram algorithms for computing accumulations on trees taking $\mathsf{O}(\log \mathsf{n})$ time on $\mathsf{O}(\mathsf{n}/\log \mathsf{n})$ processors, which is optimal.

Keywords. Tree contraction, upwards accumulations, downwards accumulations, Erew Pram, data parallel programming.

## 1 Introduction

*Accumulations* are higher-order operations on structured objects that leave the shape of an object unchanged, but replace every element of that object with some accumulated information about other elements. For example, the *prefix sums* or *scan* operation (Blelloch, 1989) on lists that, for an associative operator $\odot$, maps the list $[\mathsf{a}_1, \ldots, \mathsf{a}_n]$ to the list of 'partial sums' $[\mathsf{a}_1,\ \mathsf{a}_1 \odot \mathsf{a}_2,\ \ldots,\ \mathsf{a}_1 \odot \mathsf{a}_2 \odot \cdots \odot \mathsf{a}_n]$ is an accumulation: it replaces each element of the list with the 'sum' of the elements to its left. Another way of saying this is that information is 'passed along the list', from left to right.

This paper concerns two kinds of accumulation on binary trees, *upwards* and *downwards* accumulation. Upwards accumulation passes information up through a tree, from the leaves towards the root; each element is replaced by some function of its *descendants*, that is, of the elements below it in the tree. Symmetrically, downwards accumulation passes information downwards, from the root towards the leaves; each element is replaced by some function of its *ancestors*, that is, of the elements above it in the tree.

Upwards and downwards accumulations form the basis of many important algorithms, and so are a useful idiom to add to the programmer's toolbox. For example, computing the sizes of subtrees and the depths of nodes are natural applications of

 Authors' addresses: Dept of Computer Science, University of Auckland, Private Bag 92019, Auckland, New Zealand, email `jeremy@cs.aukuni.ac.nz` (JG); School of Applied Science, Nanyang Technological University, Singapore 2263, email `aswtcai@ntu.ac.sg` (WC); Dept of Computing and Information Science, Queen's University, Kingston, Ontario K7L 3N6, Canada, email `skill@qucis.queensu.ca` (DBS). Accepted for publication in *Science of Computer Programming*.

upwards and downwards accumulation, respectively. The *parallel prefix algorithm* (Ladner and Fischer, 1980) for computing the prefix sums of a list in logarithmic time on linearly many processors involves building a tree with the list elements as leaves, then performing an upwards and downwards accumulation on the tree (Gibbons, 1993); the prefix sums problem in turn has applications in the evaluation of polynomials, compiler design, and numerous graph problems including minimum spanning tree and strongly connected components (Akl, 1989). Upwards accumulation can be used to solve some optimization problems on trees, such as minimum covering set and maximal independent set (He, 1986). Other algorithms such as Reingold and Tilford's algorithm (Reingold and Tilford, 1981) for drawing trees and a two-pass algorithm for completely labelling a tree according to an attribute grammar (Gibbons, 1991) also consist of an upwards followed by a downwards accumulation.

For a tree with $n$ elements, these accumulations can be computed naively on a sequential machine in time proportional to $n$, and on a parallel machine with $n$ processors in time proportional to the depth of the tree. We show here how to adapt Abrahamson *et al.*'s parallel tree contraction algorithm (1989) for computing tree *reductions* to allow the accumulations to be computed in logarithmic time on an $n$-processor EREW PRAM, even if the tree has greater than logarithmic depth. Straightforward application of Brent's Theorem (Brent, 1974) reduces the processor usage to $n/\log n$, which gives optimal algorithms.

The remainder of this paper is organized as follows. In Section 2 we give the definitions of tree reductions and of upwards and downwards accumulations. In Section 3 we review parallel tree contraction. In Sections 4 and 5 we adapt the contraction algorithm to computing upwards and downwards accumulations, respectively.

## 2 Upwards and Downwards Accumulations

Our binary trees have labels drawn from two sets: *leaf* labels are drawn from a set $A$ and *junction* labels from a set $B$. A binary tree is either a leaf, labelled with an element of $A$, or a junction with two children, labelled with an element of $B$. Thus, we have no empty tree and every parent has exactly two children.

The $\circledast$-*reduction* of a tree for a ternary operator $\circledast$ from $A \times B \times A$ to $A$ reduces a binary tree to a single value in $A$. For example, the $\circledast$-reduction of the tree in Figure 1, where $a, c \in B$ and $b, d, e \in A$, is the single value $b \circledast_a (d \circledast_c e)$ in $A$. Note that the ternary operator is written with its middle argument as a subscript. (A more general form of reduction—in fact, a *homomorphism*—can be obtained by first mapping a function $f$ of type $A \to C$ for some third type $C$ over the leaves, and returning a single value in $C$; the discussion in the rest of this paper can easily be generalized to such homomorphisms.)

Tree reductions can be computed naively on a sequential machine in time pro-

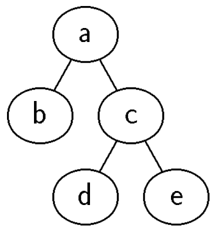


Figure 1: A binary tree

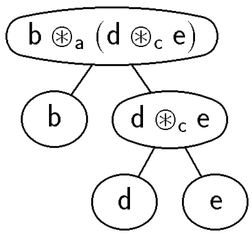


Figure 2: The $\circledast$-upwards accumulation of the tree in Figure 1

portional to the size of the tree and on a parallel machine with $n$ processors in time proportional to the depth of the tree, if the operator $\circledast$ takes constant sequential time. (For the rest of the paper, we assume that 'component' operators like $\circledast$ take constant time.) Under certain conditions on $\circledast$, the parallel contraction algorithm reviewed in the next section reduces this to logarithmic parallel time even if the tree has greater than logarithmic depth.

Upwards accumulations generalize reductions; instead of computing a single value, an upwards accumulation computes a tree of partial results with the same shape as its argument—each node is labelled with the reduction of the subtree originally rooted at that node. For example, the $\circledast$-upwards accumulation of the tree in Figure 1 is shown in Figure 2. Notice that an upwards accumulation returns a *homogeneous* tree, that is, one in which leaf and junction labels have the same type.

Upwards accumulations can be computed naively with the same amount of work as reductions, since the partial results must be computed anyway. It is significant, though, that an upwards accumulation consists of applying a *reduction*, as opposed to just any function, to all subtrees. Were this not the case, it would not be possible to compute the value for a parent from the values for its children—put another way, the required 'partial results' would not be byproducts of the naive computation of the reduction.

A downwards accumulation also computes a tree of values with the same shape

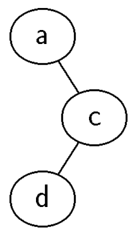


Figure 3: The path to the element $\mathsf{d}$ in the tree in Figure 1

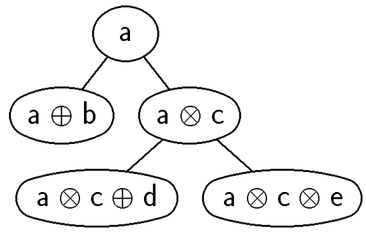


Figure 4: The $(\oplus, \otimes)$-downwards accumulation of the tree in Figure 1

as its argument; each node is labelled with a *path reduction* of the elements on the path from the root of the tree to that node. For two binary operators $\oplus$ and $\otimes$ which *cooperate* (Gibbons, 1993), that is, which satisfy the four laws

$$\begin{aligned} \mathsf{a} \oplus (\mathsf{b} \oplus \mathsf{c}) &= (\mathsf{a} \oplus \mathsf{b}) \oplus \mathsf{c} \\ \mathsf{a} \oplus (\mathsf{b} \otimes \mathsf{c}) &= (\mathsf{a} \oplus \mathsf{b}) \otimes \mathsf{c} \\ \mathsf{a} \otimes (\mathsf{b} \oplus \mathsf{c}) &= (\mathsf{a} \otimes \mathsf{b}) \oplus \mathsf{c} \\ \mathsf{a} \otimes (\mathsf{b} \otimes \mathsf{c}) &= (\mathsf{a} \otimes \mathsf{b}) \otimes \mathsf{c} \end{aligned}$$

the $(\oplus, \otimes)$-path reduction consists of replacing all 'left turns' with $\oplus$ and all 'right turns' with $\otimes$. For example, Figure 3 shows the path to the element labelled $\mathsf{d}$ in the tree in Figure 1; the reduction of this path is $\mathsf{a} \otimes \mathsf{c} \oplus \mathsf{d}$. Because of cooperativity, no parentheses are needed in this expression. Note also that a consequence of cooperativity is that $\oplus$ and $\otimes$ must each have type $\mathsf{A} \times \mathsf{A} \rightarrow \mathsf{A}$; downwards accumulations can only be performed on homogeneous trees.

The $(\oplus, \otimes)$-downwards accumulation of a tree consists of replacing every element with the $(\oplus, \otimes)$-path reduction of the path to that element. For example, Figure 4 shows the accumulation of the tree in Figure 1. Again, the result is a homogeneous tree.

## 3 Parallel Tree Contraction

Our algorithms for computing accumulations on trees are modifications of *parallel tree contraction* algorithms (Miller and Reif, 1985; Abrahamson *et al.*, 1989), which reduce a tree to a single value. We present here a description of parallel tree contraction which we generalize later to the algorithms for accumulation.

We suppose that a binary tree with $\mathsf{n}$ nodes is represented as a collection of $\mathsf{n}$ processors. Each processor $\mathsf{u}$ maintains a number of variables: $\mathsf{u.j}$ is a boolean, and is true iff $\mathsf{u}$ is a junction; $\mathsf{u.p}$ is a pointer to the parent of $\mathsf{u}$; $\mathsf{u.s}$ is a boolean, and is true iff $\mathsf{u}$ is the left child of its parent (the '$\mathsf{s}$' stands for 'side'). We suppose that $\mathsf{root}$ is a pointer to the root of the tree, and to avoid special cases we suppose that $\mathsf{root.p}$ points somewhere, and that $\mathsf{root.p.l} = \mathsf{root}$ and $\mathsf{root.s}$ is true, so that the root is 'the left child of its parent'. If $\mathsf{u.j}$ is false, $\mathsf{u}$ has the variable $\mathsf{u.a} \in \mathsf{A}$, a 'leaf label'; otherwise, $\mathsf{u}$ has the variable $\mathsf{u.b} \in \mathsf{B}$, a 'junction label'. If $\mathsf{u.j}$ is true, $\mathsf{u}$ also has the extra variables $\mathsf{u.l}$ and $\mathsf{u.r}$, being pointers to its left and right children. Finally, every processor $\mathsf{u}$ has a variable $\mathsf{u.g}$, an 'auxiliary function' from $\mathsf{A}$ to $\mathsf{A}$. This auxiliary function can be thought of as labelling the edge between $\mathsf{u}$ and $\mathsf{u.p}$ and affecting the computation of the tree reduction in the obvious way; it is initially the identity function, but we make use of it in the parallel tree contraction algorithm, to record the 'postprocessing' that has to be done at a node as the tree gets rearranged.

The purpose of *tree reduction* is to compute the value $\mathsf{naivered(root)}$, where

$$\begin{array}{rcll} \mathsf{naivered(u)} & = & \mathsf{u.g(naivered(u.l)} \circledast_{\mathsf{u.b}} \mathsf{naivered(u.r))}, & \text{if } \mathsf{u.j} \\ & = & \mathsf{u.g(u.a)}, & \text{otherwise} \end{array}$$

for some ternary operator $\circledast$. The problem is that although the reductions of the two children can be computed in parallel, this naive formulation still requires parallel time proportional to the depth of the tree—the length of the critical path—which is linear in the worst case. Parallel tree contraction reduces the complexity to logarithmic parallel time in the worst case by allowing processors to do useful work before they receive their inputs. The tree is gradually contracted, halving in size at each step, while maintaining the value of $\mathsf{naivered(root)}$; after $\lceil \log \mathsf{n} \rceil$ steps, the tree has collapsed to a single leaf, $\mathsf{root}$, and its reduction is simply $\mathsf{root.g(root.a)}$.

Contraction is performed locally by the operations $\mathsf{contractl}$ and $\mathsf{contractr}$, which 'bypass' nodes of the tree. These two operations are symmetric, so we discuss only $\mathsf{contractl}$ here; the duality is straightforward. The operation $\mathsf{contractl}$ is performed on a node $\mathsf{u}$ when $\mathsf{u}$ is a junction and $\mathsf{u.l}$ is a leaf; nodes $\mathsf{u}$ and $\mathsf{u.l}$ are removed from the tree. This is illustrated in Figure 5.

The operation $\mathsf{contractl(u)}$ that does this is:

$$\begin{array}{l} \mathsf{contractl(u)}: \\ \quad \mathsf{u.r.g} := \lambda \mathsf{x} \cdot \mathsf{u.g(u.l.g(u.l.a)} \circledast_{\mathsf{u.b}} \mathsf{u.r.g(x))}; \\ \quad \mathsf{u.r.p} := \mathsf{u.p}; \\ \quad \textbf{if } \mathsf{u.s} \textbf{ then } \mathsf{u.p.l} := \mathsf{u.r} \textbf{ else } \mathsf{u.p.r} := \mathsf{u.r}; \\ \quad \mathsf{u.r.s} := \mathsf{u.s}; \\ \quad \textbf{if } \mathsf{root} = \mathsf{u} \textbf{ then } \mathsf{root} := \mathsf{u.r} \end{array}$$

The first assignment records the deleted nodes in the auxiliary function for $\mathsf{u.r}$, so that $\mathsf{naivered(u.r)}$ has the value after the operation that $\mathsf{naivered(u)}$ did before-

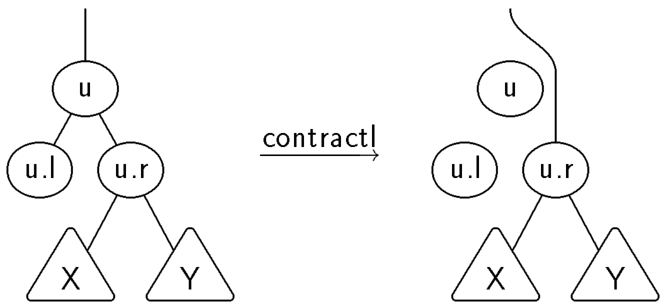


Figure 5: The operation $\mathsf{contractl(u)}$

hand; together with the pointer manipulations, this maintains $\mathsf{naivered(root)}$. The following three assignments update the pointers, effectively deleting $\mathsf{u}$ and $\mathsf{u.l}$ from the tree. The final assignment is to ensure that we still know where the root is: in our formulation of the algorithm, in contrast to others', the original root of the tree may be bypassed.

These contraction operations each remove two nodes, at least one of which is a leaf. Abrahamson *et al.* (1989) present a simple scheme by which many such contraction operations—in fact, half as many as there are leaves—can be performed in just two steps, without mutual interference. Their scheme is as follows:

(i) Assume all leaves are numbered from left to right, starting with zero. This numbering is easily computed in $\mathsf{O}(\log \mathsf{n})$ time on $\mathsf{O}(\mathsf{n}/\log \mathsf{n})$ processors (Cole and Vishkin, 1986).
(ii) Mark all even-numbered leaves.
(iii) For every junction $\mathsf{u}$ such that $\mathsf{u.l}$ is a marked leaf, perform $\mathsf{contractl(u)}$.
(iv) For every junction $\mathsf{u}$ not involved in the previous step such that $\mathsf{u.r}$ is a marked leaf, perform $\mathsf{contractr(u)}$.
(v) Renumber the leaves by halving their numbers.

Actually, Abrahamson *et al.* mark the odd-numbered leaves; marking the even-numbered leaves instead sometimes reduces the size of the tree by an extra element. Note also that the operations $\mathsf{contractl}$ and $\mathsf{contractr}$ are never performed concurrently; the reason for this is explained later.

Clearly, this scheme deletes at least half of the leaves, but we must show that no concurrent contraction operations interfere.

Note that the operation $\mathsf{contractl(u)}$ involves nodes on three consecutive levels, $\mathsf{u.p}$, $\mathsf{u}$ and $\mathsf{u.r}$; however, the fields of $\mathsf{u.p}$ and $\mathsf{u.r}$ that are involved are disjoint, so contractions involving nodes two (or more) levels apart do not interfere. Contractions involving two nodes on the same level do not interfere: the children of the two nodes are disjoint, and the nodes can at worst be left and right children of the same parent. Only nodes on adjacent levels remain to be considered. If neither of the two nodes is the parent of the other, they do not interfere, so suppose that one is the parent of the other. The two will only be contracted simultaneously if they both have marked leaves as (without loss of generality) left children—but such leaves are adjacent in left-to-right order, and so cannot both be even-numbered. Hence, a node and its parent will not be simultaneously contracted, and consequently no concurrent contraction operations interfere.

Only one aspect is left to consider: the contraction steps must take constant time, but the contraction operations assign lambda expressions of increasing size to the auxiliary functions. We must ensure that the time taken to compute new lambda expressions from old remains constant no matter what lambda expressions are generated at intermediate steps, and despite the fact that they appear to grow in size (and might therefore require longer times to access and manipulate). One way to ensure this is to use indices into a set of lambda expressions in place of the lambda expressions themselves. Then the requirement that each contraction step takes constant time is met if the indexed sets meet the following conditions:

- for every $\mathsf{u}$, the 'sectioned' binary operator $\circledast_{\mathsf{u.b}}$ from $\mathsf{A} \times \mathsf{A}$ to $\mathsf{A}$ and the function $\mathsf{u.g}$ are drawn from indexed sets of functions $\mathsf{F}$ and $\mathsf{G}$ (which contains the identity function) respectively;
- functions in $\mathsf{F}$ and $\mathsf{G}$ can be applied in constant time;
- for all $\odot \in \mathsf{F}$, $\mathsf{g} \in \mathsf{G}$ and $\mathsf{a} \in \mathsf{A}$, the functions $\lambda \mathsf{x} \cdot \mathsf{g}(\mathsf{x}) \odot \mathsf{a}$ and $\lambda \mathsf{x} \cdot \mathsf{a} \odot \mathsf{g}(\mathsf{x})$ are both in $\mathsf{G}$, and their indices can be computed from $\mathsf{a}$ and from the indices of $\odot$ and $\mathsf{g}$ in constant time;
- for all $\mathsf{g_i}, \mathsf{g_j} \in \mathsf{G}$, the composition $\mathsf{g_i} \circ \mathsf{g_j}$ is in $\mathsf{G}$, and its index can be computed from the indices $\mathsf{i}$ and $\mathsf{j}$ in constant time.

These conditions are stronger than required here, but we need the extra strength for the upwards accumulation algorithm. Note that these conditions are satisfied if $\mathsf{A}$ is finite: the functions can be modelled by matrices. Alternatively, these conditions are satisfied if there exist associative commutative operators $\oplus$ and $\otimes$ such that $\otimes$ distributes over $\oplus$, such that each $\odot$ in $\mathsf{F}$ can be written $\mathsf{x} \odot \mathsf{y} = \mathsf{a} \otimes (\mathsf{x} \oplus \mathsf{y})$ for some $\mathsf{a}$, and each $\mathsf{g}$ in $\mathsf{G}$ can be written $\mathsf{g}(\mathsf{x}) = \mathsf{b} \oplus (\mathsf{c} \otimes \mathsf{x})$ for some $\mathsf{b}$ and $\mathsf{c}$, and such that $\mathsf{G}$ contains the identity; this allows us to compute, among other things, the heights and sizes of all subtrees in logarithmic time. Either way, these conditions ensure that functions in $\mathsf{F}$ and $\mathsf{G}$ can be represented by variables of constant size, and can be combined and applied in constant time.

An example of the tree contraction procedure is given in Figure 6. For simplicity, we set $\mathsf{a} \circledast_{\mathsf{b}} \mathsf{c} = \mathsf{a} + \mathsf{c}$, ignoring the junction labels; the contraction computes the sum of the leaves. We set $\mathsf{u_i.a} = \mathsf{i}$ for every leaf $\mathsf{u_i}$. Every node $\mathsf{u}$ has auxiliary function $\mathsf{u.g} = \mathsf{id}$ unless otherwise noted, and we write '$\mathsf{c}+$' for the function $\lambda \mathsf{x} \cdot \mathsf{c} + \mathsf{x}$; nodes are 'marked' for deletion by underlining. The result is $29$, being $\mathsf{i_8.g(u_8.a)}$; indeed, $2 + 4 + 6 + 8 + 9 = 29$.

The first iteration illustrates the need to contract left and right children sepa-

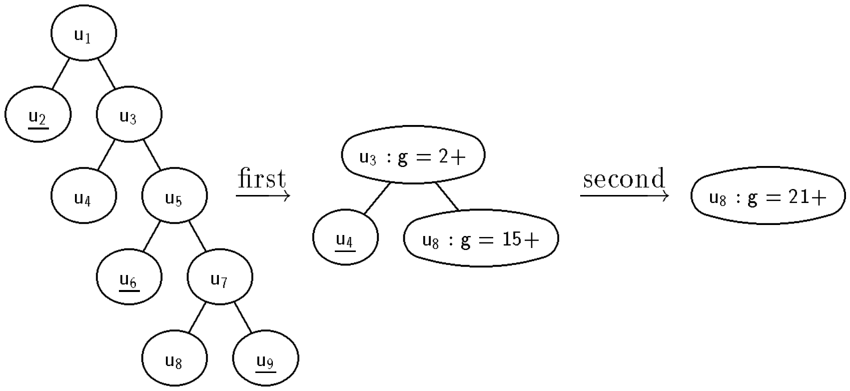


Figure 6: An illustration of tree contraction

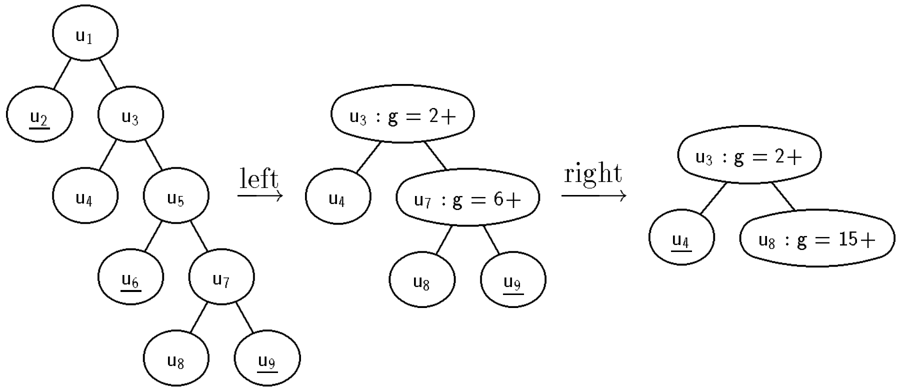


Figure 7: The first step in Figure 6, split in two

rately. In Figure 7, we split this step in two, contracting first $u_1$ and $u_5$, whose left children are marked, and then $u_7$, whose right child is marked. The function $15+$ assigned to $u_8.g$ in the second half is $(6+9)+$, and depends on the function $6+$ assigned to $u_7.g$ in the first half.

Thus, in constant time on an $n$ processor EREW PRAM we can delete half of the leaves while maintaining $\mathsf{naivered}(\mathsf{root})$, so in $O(\log n)$ time we can reduce the tree to a single leaf and thereby compute the tree reduction. A straightforward application of Brent's Theorem reduces the number of processors to $O(n/\log n)$, which gives optimal efficiency.

## 4 Parallel Upwards Accumulation

The parallel tree contraction algorithm can be used to compute the upwards accumulation, in which not only the final result but also all intermediate results are computed. In essence, whenever a node $u$ is deleted from the tree, it is placed on a stack maintained by its remaining child; after this child receives its final value, it unstacks u and computes the final value of u too.

We assume that every node u has an extra variable, u.val; the purpose of the upwards accumulation is to compute u.val for every u, the required values being those given by the sequence

```
initialize; naiveua(root)
```

where

```
initialize:
    for each node u do in parallel u.g := id;
    for each node u do in parallel if ¬u.j then u.val := u.a
```

and

```
naiveua(u):
    if u.j then begin
        do in parallel begin
            naiveua(u.l) □ naiveua(u.r)
        end;
        u.val := u.l.g(u.l.val) ⊛_u.b u.r.g(u.r.val)
    end
```

(The ' □ ' is parallel, as opposed to sequential, composition.) The problem is that, as for reduction, naiveua has a critical path as long as the tree is deep.

To get around this problem, we use tree contraction as before. Each contraction bypasses two nodes in the tree, one a leaf and one a junction. The final value to be assigned to the junction is not yet known, but it is some known function of the final value to be assigned to its remaining child. The bypassed junction, together with this function, is put aside on a stack belonging to this child; when the final value to be assigned to this child is computed, the value for the bypassed junction can in turn be computed. Every node u has yet another variable, u.st, which is a stack to contain nodes awaiting their final values.

Thus, the upwards accumulation algorithm operates in two phases, a 'collection' phase in which the tree is reduced to a single leaf and some nodes are put aside on stacks, and a 'distribution' phase in which the stacked nodes receive their final values.

### *4.1 The collection phase*

The invariant for the collection phase consists of two parts. Firstly, every node u satisfies exactly one of the following three conditions:

- node u is *awake*: u is still in the tree (reachable from the root), and if it is a leaf then u.val has been computed, or
- node u is *asleep*: for some function h, the pair (u, h) is in v.st for exactly one node v (a descendant of u in the original tree); the correct final value for u is h(x) where x is the final value assigned to v, or

- node $u$ is *dead*: $u$ is in neither the tree nor any stack, and $u.val$ is computed.

Secondly, performing a naive accumulation on the remaining tree assigns the correct values to nodes that are awake.

Initially, all stacks are empty and all nodes are awake; by definition, the naive accumulation completes the computation correctly. On completion of the collection phase, exactly one node is awake: it is a leaf, and so its final value is already computed. Moreover, the naive accumulation does nothing on a single leaf, and the distribution phase has simply to assign the correct values to all sleeping nodes.

The same tree contraction scheme is used for the collection phase as for computing reductions, although the individual contractions are different. As they are independent we again need only show that each in isolation maintains the invariant.

The operation $contractl(u)$ is only called when $u$ is a junction and $u.l$ a leaf, as before. Both $u$ and $u.l$ are bypassed; $u$ is put to sleep and $u.l$ is killed. The operation is:

$contractl(u)$:
  $push(u.r.st, (u, \lambda x \cdot u.l.g(u.l.val) \oplus_{u.b} u.r.g(x)));$
  $u.r.g := \lambda x \cdot u.g(u.l.g(u.l.val) \oplus_{u.b} u.r.g(x));$
  $u.r.p := u.p;$
  **if** $u.s$ **then** $u.p.l := u.r$ **else** $u.p.r := u.r;$
  $u.r.s := u.s;$
  **if** $root = u$ **then** $root := u.r$

Apart from the addition of the first line, the only change to the contraction operation for tree reduction is that $u.l.val$, rather than $u.l.a$, is used in the second line.

We must now show that $contractl(u)$ maintains the invariant. Node $u.l$ is killed; it is a leaf, so its value is already computed. Node $u$ is put to sleep, on $u.r.st$, and the final value it will receive is $u.l.g(u.l.val) \oplus_{u.b} u.r.g(x)$, where $x$ is the final value received by $u.r$; clearly, this value is what the naive accumulation would have given it. The assignment to $u.r.g$ thus ensures that the values given by the naive accumulation to the ancestors of $u$ remain unchanged, as are the values given to descendants of $u.r$ and all unrelated nodes. Hence the invariant is maintained.

Because the same contraction scheme is used as for tree reduction, the collection phase takes $O(\log n)$ steps.

To illustrate the process, consider the example of tree contraction given earlier. The corresponding accumulation computes for each node $u$ the sum of the leaves of the subtree rooted at $u$. At each contraction operation, one leaf is killed—being a leaf, its final value is already computed—and one junction is put to sleep. After the first iteration, $u_3.st$ contains $(u_1, 2+)$, $u_7.st$ contains $(u_5, 6+)$ and $u_8.st$ contains $(u_7, 9+)$; after the second iteration, $u_8.st$ grows to contain $(u_3, 4+ \circ 15+ = 19+)$. Thus, on completion of the collection phase, the status of the stacks is

| $u_3.st$ | $u_7.st$ | $u_8.st$ |
|---|---|---|
| $(u_1, 2+)$ | $(u_5, 6+)$ | $(u_3, 19+)$ |
| | | $(u_7, 9+)$ |

so $u_1$, $u_3$, $u_5$ and $u_7$ (the junctions) are sleeping, $u_8$ is still awake, and the remaining leaves are dead; all the leaves, of course, already have their final values.

### *4.2 The distribution phase*

Every node $u$ has a stack $u.st$ of (node, function) pairs; if $(v, h)$ is in $u.st$ then $v.val$ should be set to $h(u.val)$, once $u.val$ has been computed. The distribution phase is simply:

```
for each node u do in parallel begin
    wait until u.val is computed;
    while u.st not empty do begin
        (v, h) := pop(u.st);
        v.val := h(u.val)
    end
end
```

Clearly this terminates: there are no circular dependencies, because all dependencies run from parents to children. Clearly, also, when it terminates every node has the correct value assigned to it, by virtue of the invariant for the collection phase. We show now that the distribution phase also terminates in $O(\log n)$ steps.

Define $dep(v)$ for a node $v$ that has been put to sleep to be the node whose stack contains $v$—that is, $v$ is in $dep(v).st$. During the collection phase, $dep(v)$ may itself be put to sleep, as may $dep(dep(v))$, and so on. Define the *dependency chain* of $v$ at a particular point during the collection phase to be the sequence $v_0, v_1, \ldots, v_k$ such that $v_0 = v$, $v_i = dep(v_{i-1})$ for $1 \leq i \leq k$, and $dep(v_k)$ is not asleep. Write $ld(v)$ for $v_k$, the last sleeping node on whose final value $v$ depends, and $src(v)$ for $dep(ld(v))$, the non-sleeping node on whose final value $v$ depends. Write $sd(v)$ for the depth of $v$ in $dep(v).st$, counting the top of the stack as depth $1$.

For every node $v$, define the *dependency depth* $dd(v)$ of $v$ at a particular point during collection as $sd(v) + sd(dep(v)) + \cdots + sd(ld(v))$, the sum of the stack depths of all nodes in the dependency chain of $v$. (If $v$ is not asleep, $sd(v) = 0$.) For example, after the first iteration of the collection phase in the example, $u_7$ is at the top of $u_8.st$ and $u_5$ is at the top of $u_7.st$, so $sd(u_7) = 1$ and $sd(u_5) = 1$, and $dd(u_5) = 2$. We claim that at all points during the collection phase, $dd(v)$ is bounded above by twice the number of iterations that have been made, and after $dd(v)$ steps of the distribution phase $v$ will receive its final value. Hence, $dd(v)$ is bounded above overall by $2 \log n$, and the distribution phase takes at most $2 \log n$ steps.

Consider a node $\mathsf{v}$ and the contraction of left children during one iteration of the collection phase. The node $\mathsf{src(v)}$ is either awake or dead. If $\mathsf{src(v)}$ is dead, the dependency depth of $\mathsf{v}$ doesn't change. If $\mathsf{src(v)}$ is awake, it may be put to sleep, in which case the dependency chain of $\mathsf{v}$ grows by one element, and $\mathsf{dd(v)}$ increases by one. Alternatively, the parent of $\mathsf{src(v)}$ may be put to sleep, in which case $\mathsf{src(v).st}$ grows by one element, and $\mathsf{sd(ld(v))}$ and hence $\mathsf{dd(v)}$ increase by one. Otherwise, $\mathsf{dd(v)}$ does not change. Thus, $\mathsf{dd(v)}$ increases by at most one during the contraction of left children in one iteration of the collection phase. Similarly, $\mathsf{dd(v)}$ increases by at most one during the contraction of right children. Thus, $\mathsf{dd(v)}$ increases by at most two on each iteration of the collection phase.

During the distribution phase, $\mathsf{src(v).val}$ has been computed for every sleeping node $\mathsf{v}$. On each iteration of the distribution phase, the top node in $\mathsf{src(v).st}$ is popped and its final value computed, and so $\mathsf{sd(ld(v))}$ and hence $\mathsf{dd(v)}$ decrease by one. If $\mathsf{ld(v)}$ was at the top of $\mathsf{src(v).st}$, that is, $\mathsf{sd(ld(v))} = 1$, then this computes $\mathsf{ld(v).val}$, at which point the dependency chain for $\mathsf{v}$ shortens by one element; $\mathsf{src(v)}$ becomes what $\mathsf{ld(v)}$ used to be. (No assignments are involved; the names $\mathsf{ld(v)}$ and $\mathsf{src(v)}$ are purely for expository purposes.) When $\mathsf{dd(v)}$ reaches one, $\mathsf{ld(v)} = \mathsf{v}$ and $\mathsf{v}$ is the top element in $\mathsf{src(v).st}$, and $\mathsf{v.val}$ can be computed. Hence, $\mathsf{v.val}$ is computed after exactly $\mathsf{dd(v)}$ iterations of the distribution phase; final values of nodes are 'filled in' in the reverse of the order in which those nodes were stacked.

Returning to our example, $\mathsf{u_8.val} = 8$ is known immediately on completion of the collection phase, so $\mathsf{u_8.st}$ is popped and $\mathsf{u_3.val} = 19 + 8 = 27$ is computed. Next, $\mathsf{u_3.st}$ and again $\mathsf{u_8.st}$ are popped, and $\mathsf{u_1.val} = 2 + 27 = 29$ and $\mathsf{u_7.val} = 9 + 8 = 17$ are computed. Finally, $\mathsf{u_7.st}$ is popped and $\mathsf{u_5.val} = 6 + 17 = 23$ computed.

## 5 Parallel Downwards Accumulation

The downwards accumulation likewise operates in two phases, collection and distribution. However, it is more natural to express the contraction operations for downwards accumulation in terms of parents bypassing children, rather than children bypassing parents. To this end we reformulate the parallel tree contraction algorithm to use this method, before adapting the algorithm to downwards accumulation. Then the major change is that the two children of a junction, at least one of which is a leaf, are bypassed simultaneously, and *both* must be placed on the parent's stack, because the final values of both depend on the final value of the parent.

The purpose of downwards accumulation is to compute $\mathsf{u.val}$ for every $\mathsf{u}$, the required values being those given by

$$\mathsf{initialize;\ naiveda(id, root)}$$

where

$\mathsf{initialize:}$

**for** each node $\mathsf{u}$ **do in parallel** $\mathsf{u.g := id}$

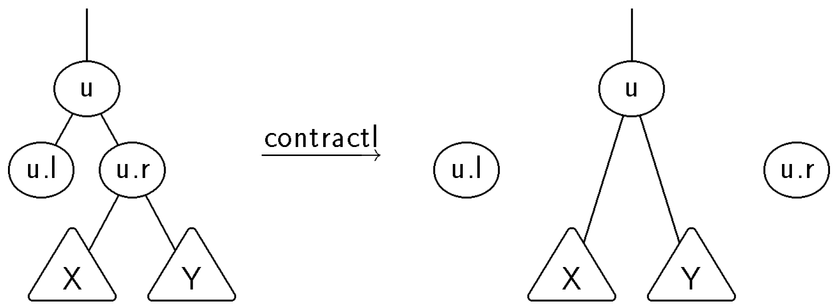


Figure 8: The operation contractl(u), with parents bypassing children

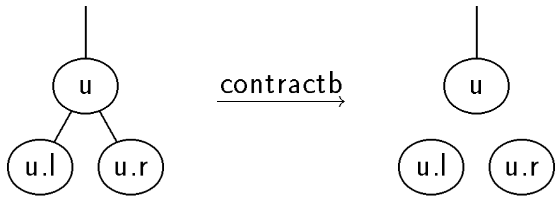


Figure 9: The operation contractb(u)

and

$$
\begin{array}{l}
\mathsf{naiveda}(\mathsf{h},\mathsf{u}):\\
\quad \mathsf{u.val} := \mathsf{h}(\mathsf{u.a});\\
\quad \textbf{if } \mathsf{u.j} \textbf{ then do in parallel begin}\\
\qquad \mathsf{naiveda}(\lambda \mathsf{x} \cdot \mathsf{u.g}(\mathsf{u.val}) \oplus \mathsf{x}, \mathsf{u.l}) \ \Box\\
\qquad \mathsf{naiveda}(\lambda \mathsf{x} \cdot \mathsf{u.g}(\mathsf{u.val}) \otimes \mathsf{x}, \mathsf{u.r})\\
\quad \textbf{end}
\end{array}
$$

As with tree reduction and upwards accumulation, the naive downwards accumulation algorithm has a critical path as long as the tree is deep, and the point of the exercise is to compute the accumulation in logarithmic time regardless of the depth.

### *5.1 Tree contraction, revisited*

Reexpressed so that parents bypass children, tree contraction involves three kinds of local operation. The operation contractl(u) is performed on a node u when u is a junction, u.l a leaf and u.r a junction; its effect is to remove both u.l and u.r from the tree. This is illustrated in Figure 8.

Symmetrically, contractr(u) is performed when u is a junction, u.l a junction and u.r a leaf, and again removes both u.l and u.r from the tree. Finally, the operation contractb(u) is performed when u is a junction and both u.l and u.r, which are again both removed, are leaves. This is illustrated in Figure 9. A consequence of this reexpression is that we have special cases at the leaves of the tree, rather than

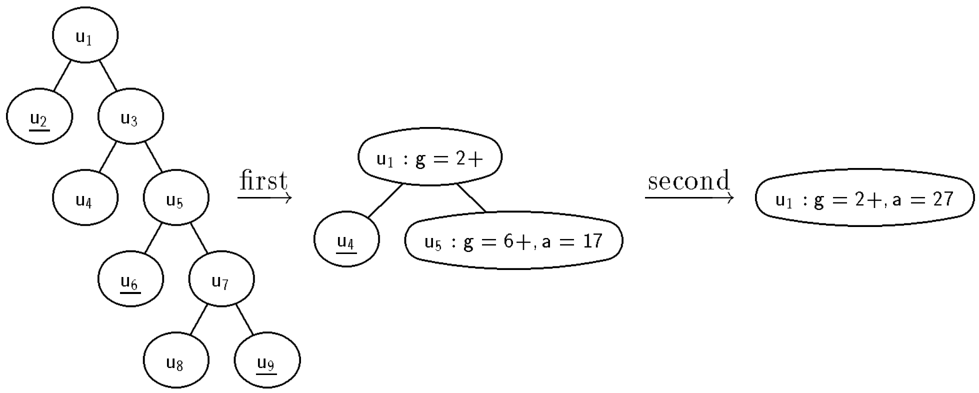


Figure 10: An illustration of revised tree contraction

at the root.

The operation $\mathsf{contractl(u)}$ is simply

$\mathsf{contractl(u)}$:

$$\mathsf{u.g} := \lambda \mathsf{x} \cdot \mathsf{u.g(u.l.g(u.l.a)} \circledast_{\mathsf{u.b}} \mathsf{u.r.g(x))};$$
$$\mathsf{u.b} := \mathsf{u.r.b};$$
$$\mathsf{u.l, u.r} := \mathsf{u.r.l, u.r.r}$$

(Note that the parent pointers are no longer needed.) This dualizes in the obvious way to $\mathsf{contractr}$.

The operation $\mathsf{contractb}$ operates similarly; node $\mathsf{u}$ is changed from a junction to a leaf, so we must construct a field $\mathsf{u.a}$ for it.

$\mathsf{contractb(u)}$:

$$\mathsf{u.a} := \mathsf{u.l.g(u.l.a)} \circledast_{\mathsf{u.b}} \mathsf{u.r.g(u.r.a)};$$
$$\mathsf{u.j} := \mathsf{false}$$

As before, both contraction operations maintain $\mathsf{naivered(root)}$. Moreover the same scheduling of local contractions ensures that conflicting contractions are not concurrent.

On our example tree, this modified contraction procedure acts as in Figure 10. The result is again 29, being $\mathsf{u_1.g(u_1.a)}$. In Figure 11, the first iteration of this process is split in two. Note that with this reformulation, junctions may become leaves and so gain $\mathsf{a}$ values—for example, $\mathsf{u_5}$ gains $\mathsf{u_5.a} = 17$ on the first iteration, when its children $\mathsf{u_8}$ and $\mathsf{u_9}$ are removed.

We show next how to adapt this tree contraction algorithm to compute downwards accumulation efficiently.

### *5.2 The collection phase*

As we observed above, under the contraction regime by which a parent bypasses its children, two children—at least one of which is a leaf—are removed from the

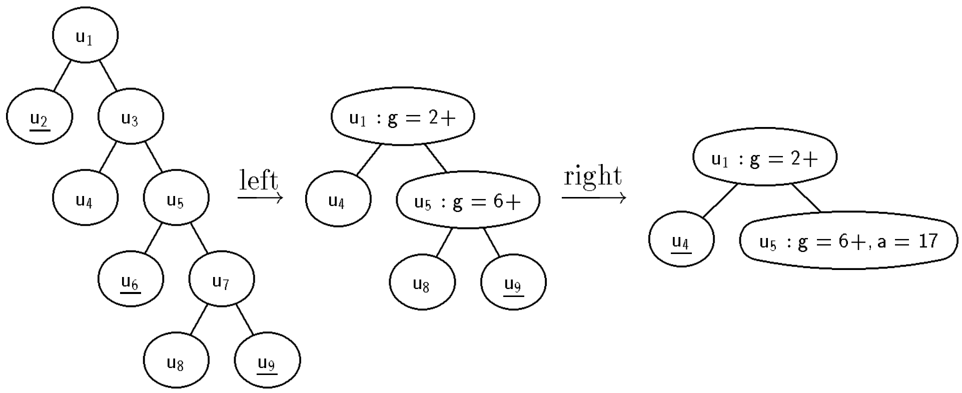


Figure 11: The first step in Figure 10, split in two

tree simultaneously. When junction $\mathsf{u}$ is contracted, that is, its children $\mathsf{u.l}$ and $\mathsf{u.r}$ are removed, the final values to be assigned to $\mathsf{u.l}$ and $\mathsf{u.r}$ are not yet known, but they are known functions of the final value to be assigned to their parent $\mathsf{u}$. Hence, the two children should be put to sleep on the parent's stack.

Because information flows down through the tree rather than up, during the downwards accumulation the auxiliary function $\mathsf{u.g}$ represents an edge label for the edges between $\mathsf{u}$ and its *children*, rather than its parent; it turns out that, because siblings are always bypassed together, the same edge label always applies to both children.

The invariant for the collection phase of the downwards accumulation algorithm is similar to that for upwards accumulation. Firstly, every node satisfies exactly one of the following two conditions:

- node $\mathsf{u}$ is *awake*: $\mathsf{u}$ is still in the tree (reachable from the root), or
- node $\mathsf{u}$ is *asleep*: assuming without loss of generality that $\mathsf{u}$ is a left child with right sibling $\mathsf{v}$, then for some function $\mathsf{h}$, the triple $(\mathsf{u}, \mathsf{v}, \mathsf{h})$ is in $\mathsf{w.st}$ for exactly one node $\mathsf{w}$ (an ancestor of $\mathsf{u}$ and $\mathsf{v}$ in the original tree); the correct final values for $\mathsf{u}$ and $\mathsf{v}$ are $\mathsf{h(x)} \oplus \mathsf{u.a}$ and $\mathsf{h(x)} \otimes \mathsf{v.a}$, respectively, where $\mathsf{x}$ is the final value assigned to $\mathsf{w}$.

(There are no 'dead' nodes; no node's final value is known until the distribution phase. Secondly, performing a naive downwards accumulation on the remaining tree assigns the correct values to nodes that are awake.

Initially, all stacks are empty and all nodes awake; by definition, the naive accumulation computes the correct values. On completion of the collection phase, exactly one node—the root—is awake, and its final value is just its label; the distribution phase need only assign the correct values to the sleeping nodes.

The junction contraction operations take the form

$\mathsf{contractl}(u)$:
  $\mathsf{push}(u.st, (u.l, u.r, u.g))$;
  $u.g := \lambda x \cdot u.r.g(u.g(x) \otimes u.r.a)$;
  $u.l, u.r := u.r.l, u.r.r$

To show that this maintains the invariant, we must show that the triple pushed onto $u.st$ provides the correct final values for $u.l$ and $u.r$, the two nodes put to sleep, and that the contraction does not change the values assigned by the naive accumulation to the nodes remaining awake. The first requirement is met by virtue of the invariant for the collection phase. For the second, we note that only descendants of $u$ may be affected by the contraction, because the naive accumulation passes information downwards. The value given to $u$ itself is unaffected. The assignment to $u.g$ ensures that the values given to the (new) children of $u$ are unaffected, and hence so are the values given to all further descendants of $u$.

Leaf contraction is simpler:

$\mathsf{contractb}(u)$:
  $\mathsf{push}(u.st, (u.l, u.r, u.g))$;
  $u.j := \mathsf{false}$

The argument for the maintenance of the invariant is correspondingly simpler. The two children of $u$ are put to sleep with the correct function, for the same reason as for junction contraction, and the values given to the remaining awake nodes are all unaffected, because none are descendants of $u$.

Using the same global scheduling scheme, the collection phase takes $O(\log n)$ steps.

Again, we have to show that as the auxiliary functions 'grow' to record more deleted nodes, they do not take longer to apply. This is the case if the following conditions are satisfied:

- the auxiliary functions are drawn from an indexed set $G$ of functions, containing the identity function;
- functions in $G$ can be applied in constant time;
- for all functions $g_i, g_j \in G$ and labels $a \in A$, the two functions $g_i \circ (\lambda x \cdot x \oplus a) \circ g_j$ and $g_i \circ (\lambda x \cdot x \otimes a) \circ g_j$ are in $G$, with indices that can be computed from $i$, $j$ and $a$ in constant time.

This in turn is satisfied if the set $G$ consists of the identity function and all functions of the form $(\lambda x \cdot x \oplus a)$ and $(\lambda x \cdot x \otimes a)$, and $A$ is finite. Alternatively, the finiteness of $A$ can be replaced by the condition that the operators $\oplus$ and $\otimes$ cooperate; this allows us to compute the depth of every node, among other things, in logarithmic time.

To illustrate this, consider the downwards accumulation in which $\oplus = \otimes = +$ on a tree of the same shape as before, but in which both leaves and junctions have $a$ values; again, $u_i.a = i$ for each $i$. The accumulation computes for each

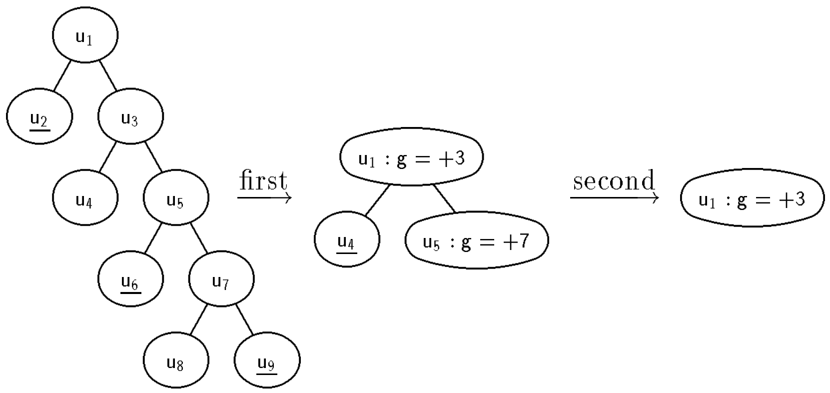


Figure 12: An illustration of computing a downwards accumulation

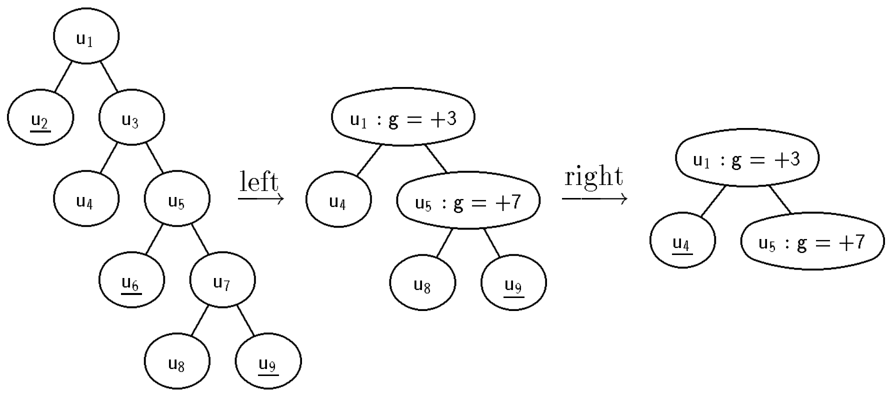


Figure 13: The first step in Figure 12, split in two

node $\mathsf{u}$ the sum of the values on the path from the root of the tree to $\mathsf{u}$. The two iterations are illustrated in Figure 12. Figure 13 dissects the first iteration to reveal the intermediate step. After the first half of the first iteration, $\mathsf{u}_1.\mathsf{st}$ contains $(\mathsf{u}_2, \mathsf{u}_3, \mathsf{id})$ and $\mathsf{u}_5.\mathsf{st}$ contains $(\mathsf{u}_6, \mathsf{u}_7, \mathsf{id})$; after the second half, the extra triple $(\mathsf{u}_8, \mathsf{u}_9, +7)$ is pushed onto $\mathsf{u}_5.\mathsf{st}$. After the second and final iteration, $\mathsf{u}_1.\mathsf{st}$ gains $(\mathsf{u}_4, \mathsf{u}_5, +3)$. Thus, the status of the stacks on completion of the collection phase is

| $\mathsf{u}_1.\mathsf{st}$ | $\mathsf{u}_5.\mathsf{st}$ |
|---|---|
| $(\mathsf{u}_4, \mathsf{u}_5, +3)$ | $(\mathsf{u}_8, \mathsf{u}_9, +7)$ |
| $(\mathsf{u}_2, \mathsf{u}_3, \mathsf{id})$ | $(\mathsf{u}_6, \mathsf{u}_7, \mathsf{id})$ |

### *5.3 The distribution phase*

Every junction $\mathsf{u}$ has a stack $\mathsf{u}.\mathsf{st}$ of (node, node, function) triples; if $(\mathsf{v}, \mathsf{w}, \mathsf{h})$ is in $\mathsf{u}.\mathsf{st}$ then $\mathsf{v}.\mathsf{val}$ should be set to $\mathsf{h}(\mathsf{u}.\mathsf{val}) \oplus \mathsf{v}.\mathsf{a}$ and $\mathsf{w}.\mathsf{val}$ to $\mathsf{h}(\mathsf{u}.\mathsf{val}) \otimes \mathsf{w}.\mathsf{a}$, once

$\mathsf{u.val}$ has been computed. The distribution phase is simply

**for** each node $\mathsf{u}$ **do in parallel begin**
  wait until $\mathsf{u.val}$ is computed;
  **while** $\mathsf{u.st}$ not empty **do begin**
    $(\mathsf{v}, \mathsf{w}, \mathsf{h}) := \mathsf{pop}(\mathsf{u.st})$;
    $\mathsf{v.val}, \mathsf{w.val} := \mathsf{h}(\mathsf{u.val}) \oplus \mathsf{v.a}, \mathsf{h}(\mathsf{u.val}) \otimes \mathsf{w.a}$
  **end**
**end**

The argument about dependency depths remains the same; the dependency depth $\mathsf{dd}(\mathsf{v})$ for each node $\mathsf{v}$ is bounded above by twice the number of iterations of the collection phase, and hence by $2\log \mathsf{n}$, and each sleeping node $\mathsf{v}$ receives its final value after $\mathsf{dd}(\mathsf{v})$ steps of the distribution phase.

In our example, initially $\mathsf{u}_1.\mathsf{val} = \mathsf{u}_1.\mathsf{a} = 1$ is computed. Then $\mathsf{u}_1.\mathsf{st}$ is popped and $\mathsf{u}_4.\mathsf{val} = 1 + 3 + 4 = 8$ and $\mathsf{u}_5.\mathsf{val} = 1 + 3 + 5 = 9$ are computed. Then $\mathsf{u}_5.\mathsf{st}$ and again $\mathsf{u}_1.\mathsf{st}$ are popped, and $\mathsf{u}_8.\mathsf{val} = 9 + 7 + 8 = 24$, $\mathsf{u}_9.\mathsf{val} = 9 + 7 + 9 = 25$, $\mathsf{u}_2.\mathsf{val} = 1 + 2 = 3$ and $\mathsf{u}_3.\mathsf{val} = 1 + 3 = 4$ computed. Finally, $\mathsf{u}_5.\mathsf{st}$ is popped again and $\mathsf{u}_6.\mathsf{val} = 9 + 6 = 15$ and $\mathsf{u}_7.\mathsf{val} = 9 + 7 = 16$ are computed.

## 6 Discussion

We have presented algorithms for the EREW PRAM for computing upwards and downwards accumulations that take $\mathsf{O}(\log \mathsf{n})$ time on $\mathsf{O}(\mathsf{n}/\log \mathsf{n})$ processors, which is optimal. This answers positively one of the questions posed in the conclusions of (Gibbons, 1991). These algorithms are adaptations of Abrahamson *et al.*'s (1989) parallel tree contraction algorithm for computing tree reductions, which is in turn a simplification of Miller and Reif's (1985) algorithm. In essence, our adaptations consist of stacking nodes that are deleted during tree contraction, and using a second 'distribution' phase to unstack the deleted nodes and compute their final values. Abrahamson *et al.* hint at this adaptation to their algorithm.

Previous tree contraction algorithms have all involved children bypassing parents; we found that downwards accumulations are more naturally computed by a contraction algorithm in which parents bypass children, presumably because information is flowing in the opposite direction.

We also give invariants and informal proofs of correctness for our algorithms.

Gibbons (1993) has presented a different algorithm for computing downwards accumulations, based on pointer doubling rather than on tree contraction. It runs in time proportional to the logarithm of the *depth*—as opposed to the size—of the tree, so is faster, but it requires the more powerful CREW PRAM model.

## References

K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka (1989). *A simple parallel tree contraction algorithm.* Journal of Algorithms, 10:287–302.

Selim G. Akl (1989). *Design and Analysis of Parallel Algorithms.* Prentice-Hall.

Guy E. Blelloch (1989). *Scans as primitive parallel operations.* IEEE Transactions on Computers, 38(11):1526–1538.

R. P. Brent (1974). *The parallel evaluation of general arithmetic expressions.* Communications of the ACM, 21(2):201–206.

R. Cole and U. Vishkin (1986). *Approximate and exact parallel scheduling with applications to list, tree and graph problems.* In *27th IEEE Symposium on Foundations of Computer Science*, pages 478–491.

Jeremy Gibbons (1991). *Algebras for Tree Algorithms.* D. Phil. thesis, Programming Research Group, Oxford University. Available as Technical Monograph PRG-94.

Jeremy Gibbons (1993). *Upwards and downwards accumulations on trees.* In *LNCS 669: Mathematics of Program Construction.* Springer-Verlag. A revised version appears in the Proceedings of the Massey Functional Programming Workshop, 1992.

Jeremy Gibbons (1993). *Computing downwards accumulations on trees quickly.* In *Proceedings of the 16th Australian Computer Science Conference*, pages 685–691.

X. He (1986). *Efficient parallel algorithms for solving some tree problems.* In *24th Allerton Conference on Communication, Control and Computing*, pages 777–786.

Richard E. Ladner and Michael J. Fischer (1980). *Parallel prefix computation.* Journal of the ACM, 27(4):831–838.

Gary L. Miller and John H. Reif (1985). *Parallel tree contraction and its application.* In *26th IEEE Symposium on the Foundations of Computer Science*, pages 478–489.

Edward M. Reingold and John S. Tilford (1981). *Tidier drawings of trees.* IEEE Transactions on Software Engineering, 7(2):223–228.