A Simple Parallel Tree Contraction Algorithm†

K. Abrahamson, N. Dadoun*
D.G. Kirkpatrick, T. Przytycka

Technical Report 87-30
August 1987

Abstract

A simple reduction from the tree contraction problem to the list ranking problem is presented. The reduction takes O(log n) time for a tree with n nodes, using O(n/log n) EREW processors. Thus tree contraction can be done as efficiently as list ranking.

A broad class of parallel tree computations to which the tree contraction techniques apply is described. This subsumes earlier characterizations. Applications to the computation of certain properties of cographs are presented in some detail.

† This research was supported in part by the Natural Sciences and Engineering Research Council of Canada.

* Current Address: Computer Science Dept., Washington State University, Pullman, Washington 99164, USA

1. Introduction

The tree contraction problem is to reduce a rooted ordered tree[1] to its root by a sequence of independent vertex removals. Tree contraction can be viewed as a general technique for scheduling parallel computations on trees. We describe an efficient parallel algorithm for tree contraction based on a simple reduction to list ranking. The reduction for trees with n nodes can be implemented on an exclusive read - exclusive write (EREW) PRAM (see [V83] for a detailed comparison of parallel models of computation) in O(log n) time and using O(n/log n) processors. Combined with recent results on parallel list ranking, it leads to a simple optimal parallel tree contraction algorithm.

Parallel algorithms for tree contraction have been studied before in several papers. Most of the earlier results have been described for the stronger concurrent read/exclusive write (CREW) model though they do not appear to make any essential use of concurrent reads. Miller and Reif [MR85] describe a deterministic algorithm which runs in O(log n) time with O(n) processors. They present their algorithm in the context of dynamic expression evaluation for an expression presented in the form of a parse tree. A single step of their algorithm converts a current binary parse tree to a simpler one by removing in parallel all leaves (RAKE operation) and compressing maximal chains of nodes with only one child (COMPRESS operation). They show that after O(log n) such steps a given tree is reduced to its root.

Miller and Reif apply their method to construct parallel algorithms for problems which can be reduced to computation on trees. They give a randomized algorithm for testing isomorphism of trees, a deterministic algorithm for constructing a tree of 3-connected components of a planar graph, and other algorithms.

---

[1]Throughout this paper trees are all rooted and ordered. These adjectives will not be repeated hereafter.

He [H86] defines the binary tree algebraic computation (BTAC) problem and applies Miller and Reif's technique to obtain a parallel algorithm for this problem. Roughly speaking the BTAC problem is to compute the value of an algebraic expression given in the form of a parse tree under the assumption that the algebra in which the computation is performed has a finite carrier.

Another approach to dynamic expression evaluation is presented by Gibbons and Rytter [GR86]. Their algorithm runs in $O(\log n)$ time using $O(n/\log n)$ processors. They assume, however, that the input (the string representing the expression to be computed) is given in an array and is hence preordered. Similarly to He, they prove that the algorithm can be applied to compute an algebraic expression in any algebra with finite carrier.

Cole and Vishkin [CV86c] propose an alternative method for computation on trees. They solve the tree contraction problem by parallel reduction to the list ranking problem. In this respect their approach is similar to ours. Our reduction, an abstraction of the technique used for the parallel preprocessing of region trees for fast (sequential) subdivision search [DK87], is simpler and more explicit than that of Cole and Vishkin; in particular, it completely avoids the centroid decomposition techniques that lie at the heart of their reduction.

2. Reduction to list ranking

Within a linked list, the *rank* of a given element is defined as the number of elements following that element. Given a linked list, the *list ranking problem* is to determine in parallel the rank of each element in the list. Once each element has its rank, an ordered array of elements can be constructed simply by using the rank as an array index. The list ranking problem generalizes easily to the weighted list ranking problem, where each element has an associated weight and the goal is to determine the *weighted rank* — the sum of the weights of all subsequent elements — for each list element.

The list ranking problem has received much attention in recent research (*cf.* [ADKP87], [CV86a], [CV86b]) because of its fundamental role in many parallel algorithms. Of particular interest is its role in the so-called *Euler Tour Technique* [TV84] which can be used to compute various functions on trees including preorder number, postorder number, descendent numbering and others.

Among the important contributions of Miller and Reif [MR85] is their abstraction of the problem of tree contraction. This leads to a separation of the problem from its familiar applications (notably, dynamic expression evaluation) and places it, along with list ranking, among the fundamental problems of parallel computation. A natural side effect has been the identification of new and unforeseen applications [DK87, He86, GR86]. To further solidify this abstraction we present the following definitions.

We will assume that trees are presented as an (unordered) array of vertices each of which has associated with it a parent pointer and a doubly-linked list of children. (Of course, it is possible to efficiently convert to or emulate such a representation starting with other more primitive representations.) (Successive) vertices on any list of children are said to be (immediate) siblings.

Let T be any binary tree with vertex set $V(T)$. A sequence of trees $T_1, T_2, \ldots T_k$ is said to be a *tree contraction sequence* of length k for T if,

(i) $T_1 = T$;

(ii) $V(T_i) \subseteq V(T_{i-1})$;

(iii) $|V(T_k)| \leq 3$; and

(iv) if $v \in V(T_{i-1}) - V(T_i)$ then either

(a) v is a leaf of T_{i-1}, or

(b) v has exactly one child, x, in T_{i-1}, $x \in V(T_i)$, and the parent of v in T_{i-1} is the parent of x in T_i.

It is clear from the definition that successive trees in a tree contraction sequence are formed by "independent" executions of the following two fundamental operations (see Figure 2.1): Prune(v) — leaf v is removed from the current tree; and Bypass(v) — non-root node v with exactly one child x is removed from the current tree, and the parent w of v becomes the new parent of x (with x replacing v as a child of w).

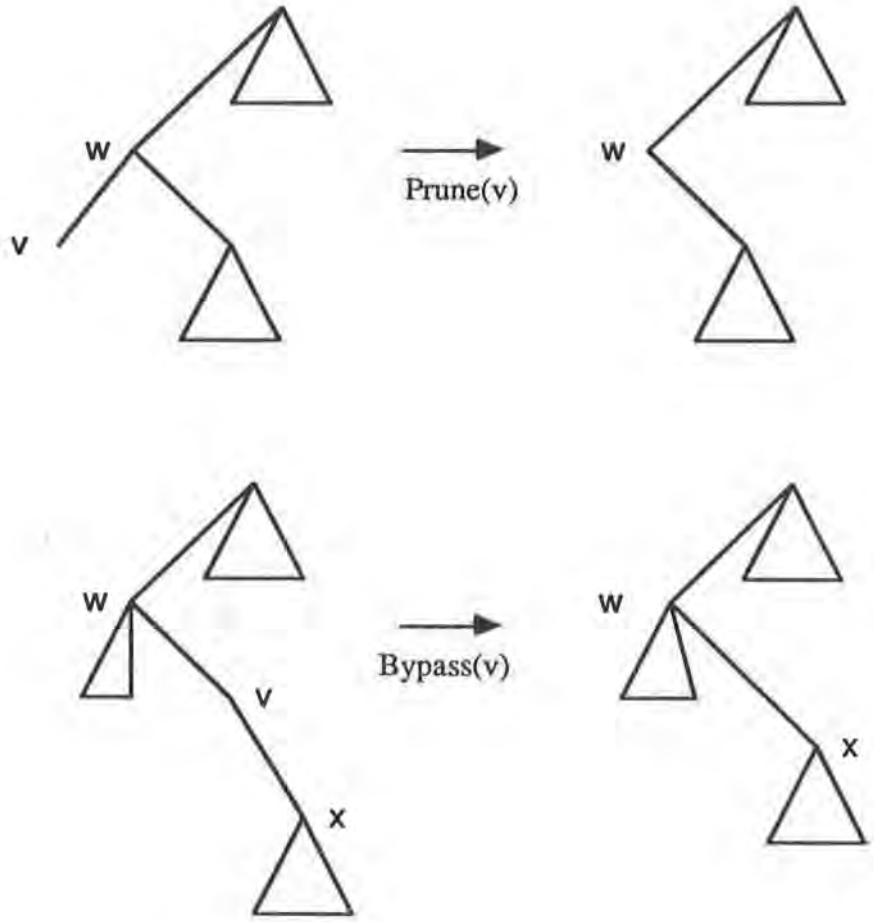

Figure 2.1. Prune and Bypass operations

By "independent" we mean that if v is pruned or bypassed then its parent is not bypassed. In this way tree modifications are ensured to be local and executable in parallel.

It is clear that every binary tree with n nodes admits a contraction sequence of length n; simply prune $n - 1$ times in succession until the tree has been reduced to its root. On the other hand, since just over half of the nodes can be removed in any one step, every contraction sequence must have length at least $\log(n) - 1$. We say that a contraction sequence is *optimal* if it has length $O(\log n)$. (Note, it may not be immediately obvious that every binary tree has an optimal contraction sequence, let alone that such a sequence can be constructed efficiently.)

The *tree contraction problem* is to construct, for an arbitrary binary tree T, an optimal tree contraction sequence for T. It is not necessary to construct the sequence explicitly; it suffices to associate with each node v the index i of the highest indexed tree containing v in the sequence, together with pointers to the parent and child (if any) of v in T_i.

It is meaningful to talk about tree contraction for non-binary trees as well. While a number of possible definitions suggest themselves the most natural is probably that which arises in conjunction with the familiar interpretation of general trees as regular binary trees (*cf.* Knuth[K68], pp. 332-345). Suppose T is an arbitrary ordered rooted tree. Consider the tree T' constructed from T as follows: If v is a vertex of T with d children then the vertex set of T' includes $v^1, v^2, \ldots, v^{d+1}$. Vertex v^{i+1} is the right child of vertex v^i in T', for $1 \le i \le d$. Furthermore, if vertex w is the i^{th} child of v in T then vertex w^1 is the left child of vertex v^i in T' (see Figure 2.2).

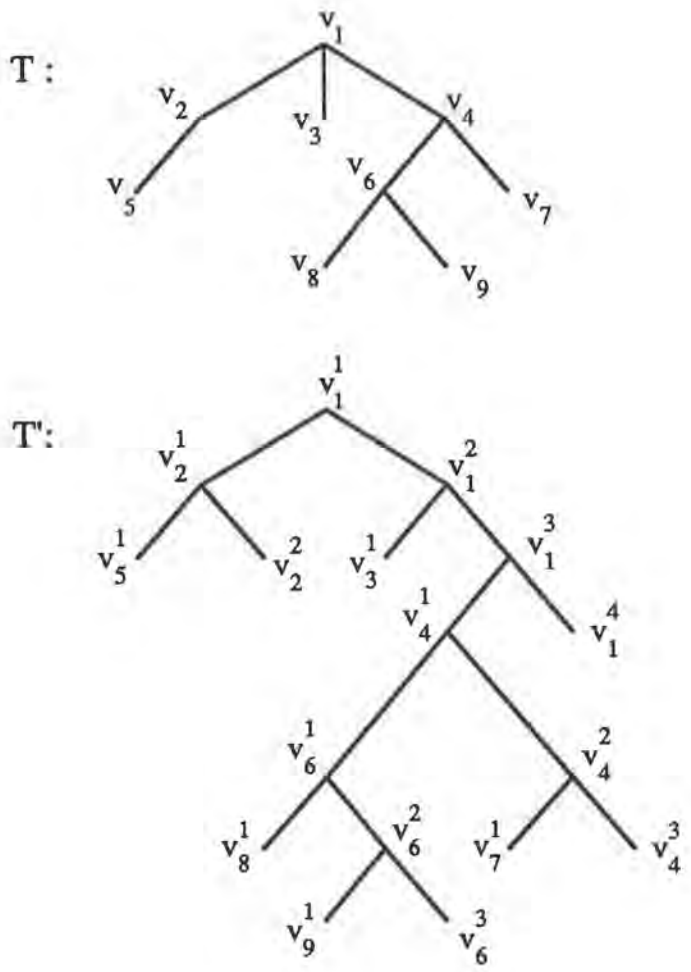

Figure 2.2 Interpretation of Trees as Binary Trees

Note that T' is a regular binary tree (all of its internal nodes have exactly two children). Furthermore, there is no cost associated with the construction of T' from T since it involves only a reinterpretation of pointers. Any tree contraction sequence for T' yields — what we might define to be — a tree contraction sequence for T. A vertex v of T is pruned or bypassed at step i if the last of its associated vertices $v^1, \ldots, v^{d+1}$ in T' is pruned or bypassed at step i. A straightforward case analysis shows that the resulting sequence satisfies the additional "independence" condition that no two adjacent siblings are removed simultaneously. Since T' has fewer than twice as

many vertices as T, it follows that any optimal tree contraction sequence for T' yields an optimal tree contraction sequence for T. (This reduction of general tree contraction to binary tree contraction is similar to the "binarization" step of Cole and Vishkin [CV86c].)

Suppose once again that T is a binary tree. It is clear that if T has height h then T admits a contraction sequence of length h consisting entirely of prunes but when T is unbalanced such a sequence is far from optimal. Miller and Reif [MR85] introduce the idea of path compression — a familiar operation in the context of list ranking — to cope with trees, or parts thereof, that have become so unbalanced that they have degenerated into paths. Miller and Reif show that by interleaving global pruning (that is, the removal of *all* leaves) with global path compression (the compression of *all* paths of length greater than one) it is possible to construct an $O(\log n)$ contraction sequence in $O(\log n)$ time using $O(n)$ processors. The complications of Miller and Reif's approach, especially with regard to processor reduction, arise in the compression step. Our approach differs in that we define the primitive contraction operations at a more elementary level. By scheduling the order in which leaves are eliminated it is possible to completely eliminate the difficulties associated with path compression. Indeed, paths of length greater than two never arise as our algorithm contracts an (initially regular) binary tree.

The pair of operations prune(v) followed by bypass(parent(v)) (where v is any leaf) form a conceptual unit in our algorithm. The algorithm proceeds in phases, each of which consists of a batch of these basic contractions performed in parallel. The independence of the underlying operations is guaranteed by a simple global schedule for leaf removal. Let the leaves be numbered in left to right order. A leaf is removed in phase t if the rightmost 1 in its leaf index is in position t.

Our binary tree contraction algorithm has the following simple description:

```
procedure contract (T)
(* Assign leaf indices from 0 to n - 1 *)
 for each leaf v in parallel
      index(v) ← left_to_right leaf index of v
(* Contraction iterations. *)
 repeat ⌈log n⌉ - 1 times
      for each leaf v in parallel
           w ← parent (v)
           if index(v) is odd and w ≠ root
                then if v is a left child
                          then prune (v)
                               bypass (w)
                     if v is a right child
                          then prune (v)
                               bypass (w)
                else index(v) ← index (v) /2
```

Note that the innermost **if** statements, though they have opposite conditions, are intended to be executed in sequence, with appropriate synchronization in between. Thus, each iteration of the **repeat** loop has four slots in which prune or bypass operations may be executed. Accordingly, we associate four elements of the tree contraction sequence with each iteration of the **repeat** loop, describing the tree after each of the four slots. It is also helpful to view the behaviour of the algorithm at two other levels. It is immediate from the description that each prune operation is immediately followed by a bypass. Hence in each successive pair of slots a number of composite prune-bypass operations are executed in parallel. Each pair of these composite slots (making up an entire iteration of the **repeat** loop) serves to eliminate all of the leaves with odd parity in the current tree together with their parents.

Lemma 2.1. Procedure **contract** constructs an optimal tree contraction sequence.

Proof: It suffices to demonstrate that the prunes and bypasses are performed independently. Since prunes and bypasses are never executed simultaneously, it need only be demonstrated that no vertex v and its parent w are ever bypassed simultaneously. Suppose this is not the case. Without loss of generality, v is the right child of w. Since they are bypassed simultaneously, they must both have leaves as left children. But since these leaves are adjacent in the left to right order and since the index array maintains each leaf's left-to-right rank in the current contracted tree, v and w must have indices of opposite parity, a contradiction. •

Theorem 2.1. Procedure **contract** provides an $O(\log n)$ time and $O(n / \log n)$ processor deterministic reduction of tree contraction to list ranking.

Proof: First note that there is a straightforward reduction of the leaf ranking problem, which arises in the first step of procedure **contract**, to the list ranking problem. The method is called the Euler Tour technique [TV84]. Each node v of T is split into three nodes v_T, v_L and v_R. For each of the resulting nodes we define a *next* field as follows. If v is a leaf then $v_T.\text{next} = v_L$ and $v_L.\text{next} = v_R$. If w is the right child of v then $v_L.\text{next} = w_T$ and $w_R.\text{next} = v_R$. If w is the left child of v then $v_T.\text{next} = w_T$ and $w_R.\text{next} = v_L$. What results is a list that starts at root_T and ends at root_R and traverses each edge of T once in each direction. If we assign $\text{weight}(z) = 1$ if $z = v_T$ and v is a leaf and $\text{weight}(z) = 0$ otherwise, then the weighted rank of each node v_T, where v is a leaf, gives the leaf index of v in T.

As a consequence of the above, it suffices to prove that the iterated contraction step of procedure **contract** can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors. An $O(\log n)$ time, $O(n)$ processor implementation is immediate; if one processor is devoted to each leaf then each phase can be carried out in $O(1)$ time. On the other hand if each of $n / \log n$ processors is assigned to a block of $\log n$

successive leaves, the obvious simulation of the $O(n)$ processor implementation incurs an additional additive overhead of only $O(\log n)$ time (*cf.* [B74], Lemma 2). •

It follows from Theorem 2.1 that results for list ranking carry over directly to tree contraction. We summarize the most important implication in the following:

Corollary 2.1. The tree contraction problem can be solved deterministically in $O(\log n)$ time using $O(n / \log n)$ processors.

Proof: This is an immediate consequence of Cole and Vishkin's $O(\log n)$ time $O(n / \log n)$ processor deterministic list ranking algorithm [CV86b]. •

The deterministic list ranking algorithm of [CV86b], though asymptotically optimal, does not provide a practical solution to the problem for lists of realistic size. A more practical deterministic solution — which uses $O(\log n \log^* n)$ time with $O(n / (\log n \log^* n))$ processors — is described in [CV86a]. Alternatively, there exist practical randomized parallel list ranking algorithms that achieve the asymptotically optimal bounds [MR85, ADKP87].

3. Algebraic tree computations

The applications presented in this and the following section serve to illustrate our tree contraction algorithm. In particular, we will use the explicit sequence of trees created by the algorithm in section 2 and treat each prune and its subsequent bypass as an indivisible operation. Though the descriptions benefit from the simplicity of our particular approach to tree contraction, it is clear that with suitable modifications other tree contraction schemes may be used for these applications.

A tree contraction algorithm gives a method for solving a large class of parallel tree computation problems. This class includes, for example, dynamic expression evaluation ([MR85], [GR86]). He [H86] and Gibbons and Rytter [GR86] noted that any algebraic expression with operands from an algebra with carrier of fixed finite size can be computed in the cost of tree contraction. This result provides efficient parallel algorithms for several optimization problems, for example minimum covering set, maximum independent set and maximum matching, when the underlying graph is a tree [H86].

In fact, we can relax the assumption that the carrier of the algebra is of fixed finite size and put some restrictions on the operations only. We can generalize He's binary tree algebraic computation problem in the following way: Let S be a set and $F \subseteq \{f \mid f: S\times S\rightarrow S\}$ a set of two-variable functions over S. The objective of bottom-up algebraic tree computations is to take any regular binary tree T whose leaves are labelled by elements of S and whose internal nodes are labelled by elements of F and to evaluate the algebraic expression associated with T (where functions at internal nodes denote operators and elements labelling leaves are operands).

It is natural (and helpful) to generalize the above notion to include a set of functions $G \subseteq \{g \mid g : S\rightarrow S\}$, including the identity function, which serve as edge labels and influence the computation in the obvious way. The triple (S, F, G) defines a *bottom-up algebraic tree computation* (B-ATC) *problem.* Such an problem is said to be *decomposable* if

(i) the sets F and G are indexed sets and their elements can be evaluated in O(1) sequential time; and

(ii) for all $g_i, g_j, \in G, f_m \in F$ and $a \in S$, the functions g_s and g_t given by $g_s(x) = g_i(f_m(g_j(x), a))$ and $g_t(x) = g_i(f_m(a, g_j(x)))$ both belong to G and their indices s and t can be computed in O(1) sequential time from i, j, m and a.

Condition (ii) defines a kind of closure operation on the sets F and G. Its significance is perhaps most easily understood by referring to the transformations of Figure 3.1.

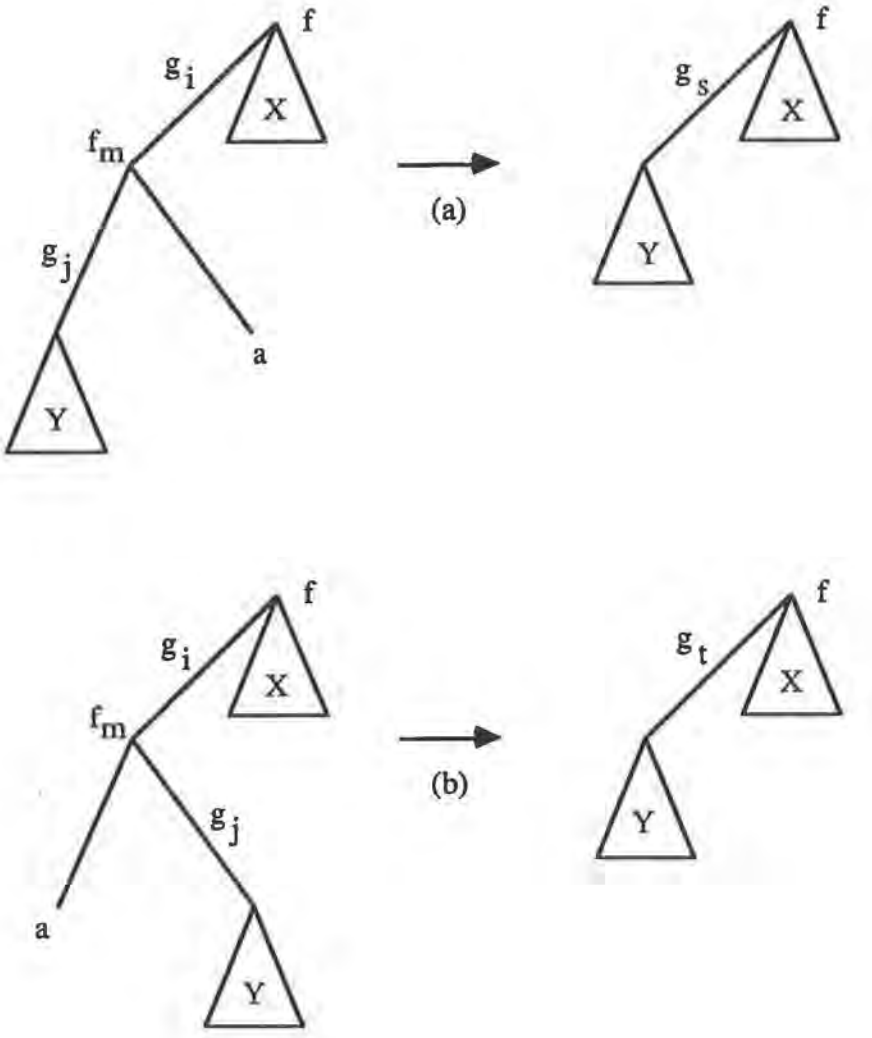

Figure 3.1

Note that any algebra with a finite carrier has associated with it a decomposable B-ATC problem. It is also easy to see that general {+, -, *, /} arithmetic computations can be described as a decomposable B-ATC problem [B74], provided arithmetic operations are assumed to take constant time.

Theorem 3.1: If (S, F, G) is a decomposable B-ATC problem then for any input tree the associated algebraic expression can be evaluated in time proportional to the cost of tree contraction using the same number of processors.

Proof: Condition (ii) of decomposability guarantees that each of the trees in the tree contraction sequence of T can be labelled in such a way that their associated algebraic expression is equivalent to that of T. Since this labelling can be computed in O(1) time at each step the entire computation runs in time proportional to that of tree contraction.•

Remark 3.1: We should note that we can easily modify the general computation scheme in such a way that we will compute also all the functions in internal nodes. In order to do so each bypassed node should maintain a pointer to the lower level endpoint of the bypassing edge (i.e. the node whose computation has to be finished in order to finish computation in the given node). For example, in Figure 3.1 the bypassed node labelled f_m retains a pointer to the leaf labelled a and the root of the subtree Y. After finishing the computation of the function in the root we add a new phase to the algorithm. In this phase we allow all the internal nodes to finish computation of the values of the function assigned to these nodes. The computation is completed at vertices in the reverse of their order of elimination in the contraction sequence.

Remark 3.2: Let us assume that we have computed the value of the expression defined by a given tree T. Assume now that we would like to compute an expression defined by the same expression tree except that one leaf has a new value. To compute this modified expression we can use all the partial results from the previous computation with the exception of the values on the path from the leaf to the root. We can do it in O(log n) *sequential* time using information stored in the sequence of trees constructed by the contraction algorithm. We need to make only one change in each of

the trees. By conditions (i) and (ii) we need constant time to correct each of them. There are O(log n) trees so the algorithm runs in O(log n) time. As an easy generalization, we note that in order to recompute the value of the expression when values in l leaves have been changed we need O(log n) time with O(min(l, n/log n)) processors.

The B-ATC problem provides a useful abstraction of many bottom-up tree-based computations. In a number of applications it is necessary to consider top-down (or perhaps a combination of bottom-up and top-down) computations based on trees. Let S be a set as before and let $H \subseteq \{h \mid h : S \rightarrow S\}$ be a set of one-variable functions over S. The objective of a top-down algebraic tree computation is to take any regular binary tree T whose root is labelled by an element of S and whose non-root nodes are labelled by elements of H and to evaluate the function associated with each node in the natural way (the root takes its given value and the value of a non-root node is its associated function applied to its parent's value).

As with the B-ATC problem, it is natural to generalize this notion by associating elements of H (now assumed to include the identity function) with the edges of T as well; the resulting computation are modified in the obvious way. A pair (S, H) defines a *top-down algebraic tree computation* (T-ATC) *problem*. Such a problem is said to be *decomposable* if

(i) the set H is indexed and its elements can be evaluated from their index and argument in O(1) sequential time; and

(ii) H is closed under composition and for each h_i, $h_j \in H$ the index of $h_i \circ h_j$ can be computed in O(1) sequential time from i and j.

Theorem 3.2 If (S, H) is a decomposable T-ATC problem then for any input tree T the value associated with each vertex can be computed in time proportional to the cost of tree contraction, using the same number of processors.

Proof: By the decomposability condition, each edge e introduced by a bypass operation can, as part of the tree contraction process, be labelled by the composition of the functions associated with the nodes and edges on the path (in T) joining the endpoints of e. The vertices of T are considered in the opposite order of their elimination in the tree contraction sequence. It is straightforward to confirm that the value of all vertices in tree T_i can be computed in O(1) time knowing the values of all vertices in tree T_{i-1}. •

4. An application of the tree contraction method: Parallel algorithms for some problems on cographs

In this section, we illustrate the application of the tree contraction method to the solution of some problems for complement reducible graphs.

A *complement reducible graph,* also called *cograph,* is defined recursively in the following way:

(i) A graph on a single vertex is a cograph.

(ii) If G and H are cographs, then so is their union.

(iii) If G is a cograph, then so is its complement.

A cograph can be represented by its parse tree. Let us assume that this parse tree has a form of a full binary tree with two kinds of internal nodes: union nodes and complement-union nodes (that is nodes representing complement of the union of the graphs described by the descendent nodes). An example of a cograph and its parse tree (not unique) is given in Figure 4.1.

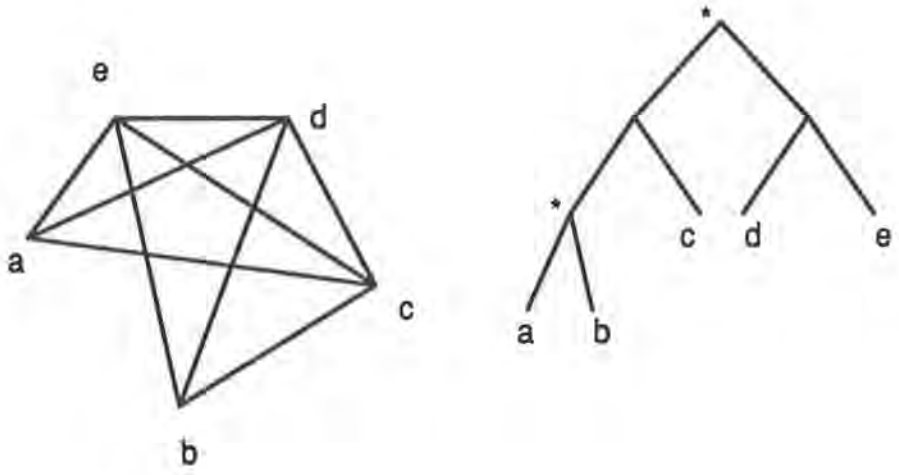

Figure 4.1 A cograph and its parse tree (complement union nodes are denoted by *)

Suppose that G is a cograph and T is a parse tree representation of G. For each node x of T, we denote by G_x the cograph represented by the subtree of T rooted at x. It is helpful to label each internal node x of T by its *complement parity* in T, i.e. by 0 if the number of complement-union nodes between the node x and the root (inclusive) is even and 1 otherwise. This labelling makes it possible to decide if two given nodes (leaves in the parse tree) are connected, simply by checking the label of their lowest common ancestor in the parse tree. It also simplifies descriptions of algorithms and has been used in conjunction with a normal form for cograph parse trees, called *cotrees* [CLS81].

Note that the problem of computing the complement parity of all nodes is a simple instance of a decomposable T-ATC problem. The set S is just $\{0, 1\}$, the root is labelled by 1 if it is a complement-union node (0 otherwise), and internal nodes have associated the function $f(x) = 1 - x$ if they are complement-union nodes (and $f(x) = x$ otherwise). Thus by Theorem 3.2, the complement parity of all nodes can be computed in parallel in the time of tree contraction.

The representation of a cograph as a parse tree with complement parity labelling leads to polynomial algorithms for many problems which are very difficult for general graphs ([CLS81], [L71], [S78]). With the help of the parallel tree contraction algorithm, some of those functions can be computed in logarithmic parallel time. The algorithms presented below can be also applied to the cotree representation of a cograph if we assume that they are preceded by a binarization step.

Consider the problem of computing the size of the largest clique in a given cograph G (*cf.* [S78]). (In the case of cographs this value is equal to the chromatic number of the graph.) We assign the value 1 to each of the leaves of G's parse tree T. To internal nodes with complement parity 0 (0-nodes) we associate the function $f_0(x,y) = \max(x,y)$ and to those with complement parity 1 (1-nodes) we associate the function $f_1(x,y) = x + y$. It is easy to confirm that the value computed at the root of T gives the size of a largest clique in G. More generally, the value computed at node x of T is the size of the largest clique in G_x, if x has the same complement parity as the root, and the largest independent set in G_x, otherwise. Call this value the *clique size value* associated with x.

Notice that it is possible to avoid the preprocessing of the parse tree needed to obtain the complement parity. Having an unlabelled parse tree we can compute both maximal clique and maximal independent set together and switch those values when the complement operation is performed.

The functions associated with edges all have the form: $g(x) = \max(a, x+b)$. Initially we set for each edge $g(x) = x = \max(0, x+0)$. To prove that the problem can be solved in the cost of tree contraction it suffices to check condition (ii) in Section 3. But that follows easily from the identity $\max(u, v) + \max(w, x) = \max(u+w, u+x, v+w, v+x)$.

More generally we can find the size of a largest clique of an arbitrary chosen induced subgraph of G. To do so simply assign 1 to the leaves of T representing nodes

of the chosen subgraph and 0 to all other nodes. With this modification the above algorithm finds the size of a largest clique in the subgraph induced on the chosen nodes. Note that if the chosen subgraph is updated (by adding or/and removing some nodes) the result can be recomputed in the way described in Remark 3.2.

By exchanging functions in 0-nodes with functions in 1-nodes we obtain an algorithm for computing the size of a largest independent set in the given cograph. Using the tree contraction method we can also solve the following problems for cographs : number of maximal cliques , number of maximal independent sets, number of cliques of largest size and number of independent sets of largest size.

Finally, consider the problem of *identifying* a clique of maximum size in G. Specifically, we would like to label the leaves of T by 0 or 1 according to whether or not the corresponding node belongs to the chosen largest clique. More generally, we would like to compute at each non-root node x a binary value indicating whether or not any node of G_x belongs to the chosen largest clique. Call this the *clique choice value* associated with x. Assume that we have evaluated the clique size value associated with all of the internal nodes in T (*cf.* remark 3.1). We now associate new functions with each non-root node of T as follows. Suppose that x is a node with children y and z in T and suppose, without loss of generality, that the clique size value of y is at least the clique size value of z. If x has complement parity 1 then both y and z are assigned the identity function. Otherwise, y is assigned the identity function and z is assigned the zero function.

If the root is assigned the value 1 then this local assignment of functions creates an instance of another decomposable T-ATC problem. A leaf x gets the final value 1 if and only if all of the vertices on the path from x to the root are assigned the identity function. By construction, the lowest common ancestor of any two leaves with final value 1 must have complement parity 1 (i.e. they must be connected in G). Furthermore, if the maximum clique has size t, then exactly t leaves get final value 1

(i.e. these leaves form a maximum clique). By Theorem 3.2, it follows that a clique of maximum size in a cograph (represented by its parse tree) can be identified in parallel in the time of tree contraction.

5. Discussion

The optimal reduction of tree contraction to list ranking allows us to construct a simple tree contraction algorithm which runs in the cost of the chosen list ranking algorithm. As a consequence, we have an optimal solution to parallel tree contraction using the list ranking results of Cole and Vishkin [CV86b]

Gibbons and Rytter [GR86] have presented an optimal algorithm for dynamic evaluation of algebraic expressions assuming that the input is given in an array. They apply the algorithm of Bar-On and Vishkin [BV85] to obtain a parse tree. The leaves of the parse tree are numbered from left to right on the basis of their index in the input array. This numbering makes it possible to group leaves in blocks of $O(\log n)$ size and then to perform computation in each block in parallel. As a result the parse tree is contracted to a tree of size $O(n/\log n)$, at which point the algorithm of Reif and Miller computes the tree contraction using an optimal number of processors.

If we assume that the input is given in the form of a binary tree instead of an array the algorithm of Gibbons and Rytter can still be applied by numbering the leaves of the tree in a preprocessing step. As in our algorithm, this can be done in the cost of list ranking using the Euler tour technique [TV84]. An advantage of our approach is the clear identification of the role played by list ranking in tree contraction. In our approach the method in which a tree is contracted does not depend on the actual size of the tree. In this respect, our approach is more similar to that of Cole and Vishkin [CV86c].

Cole and Vishkin's tree contraction algorithm uses the same two basic parallel operations as ours. The first stage of Cole and Vishkin's algorithm is to compute for each tree vertex v the function SIZE(v), which is equal to the number of nodes in the subtree rooted at v. This step is done using the Euler tour technique. Then, using function SIZE, the tree is partitioned into the so-called centroid paths. The bypass operation can be applied only to a node which does not have a non centroid child. The next step of the algorithm (called the scheduling stage) is to compute the order in which the prune and bypass operations should be performed. In the last stage, called the evaluation stage, the prune and bypass operations are performed according to the order computed in the previous stage. In the evaluation stage some work has to be done in order to assign processors to the jobs.

In our algorithm the reduction to list ranking is done in a simpler way. The only information which is used to schedule the order of performing bypass operations is the initial numbering of the leaves. Consequently we do not need a separate scheduling phase, which in Cole and Vishkin's algorithm is quite complicated. Also the assignment of processors to the jobs is simpler in our approach.

Acknowledgment. We wish to express our thanks to Nicholas Pippenger who first pointed out the connection between our work on parallel processing of region search trees [DK87] and Miller and Reif's work on parallel tree contraction [MR85].

References

[ADKP87] K. Abrahamson, N. Dadoun, D. Kirkpatrick and T. Przytycka. "A simple optimal randomized parallel list ranking algorithm". Computer Science Department Technical Report 87-14, University of British Columbia, Vancouver, May 1987.

[B74] R. Brent. "The parallel evaluation of general arithmetic expressions". *Journal of the ACM* 21, 2, April 1974, pp. 201-206.

[BV85] I. Bar-On and U. Vishkin. "Optimal parallel generation of a computation tree form". *ACM Transactions on Programming Languages and Systems* 7,2, 1985, pp. 348-357.

[CLS81] D.G. Corneil, H. Lerchs and L. Stewart. "Complement reducible graphs. " *Journal of Discrete and Applied Mathematics* 3, 1981, pp. 163-175.

[CV86a] R. Cole and U. Vishkin. "Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. " In *18th Annual Symposium on Theory of Computing*, 1986, pp. 206-219.

[CV86b] R. Cole and U. Vishkin. "Approximate and exact parallel scheduling with applications to list, tree and graph problems. " In *27th Annual Symposium on Foundations of Computer Science*, 1986, pp. 478-491.

[CV86c] R. Cole and U. Vishkin. "The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time". Ultracomputer Note #108, TR-242, Dept. of Computer Science, Courant Institute NYU, 1986.

[DK87] N. Dadoun and D. Kirkpatrick. "Parallel processing for efficient subdivision search". In *3rd ACM Symposium on Computational Geometry* , Waterloo, Ontario, 1987, pp. 205-214.

[GR86] A. Gibbons and W. Rytter. "An optimal parallel algorithm for dynamic tree expression evaluation and its applications". In *Symp. on Foundations of Software Technology and Theoretical Comp. Sci.*, 1986, pp. 453-469.

[H86] X. He. "Efficient parallel algorithms for solving some tree problems. " In *24th Allerton Conference on Communication, Control and Computing*, 1986, pp. 777-786.

[K68] D. Knuth, *The Art of Computer Programming Volume 1: Fundamental Algorithms*, Addison-Wesley, 1968.

[L71] H. Lerchs. "On cliques and kernels. " Dept. of Comp. Science , University of Toronto, March 1971.

[Me83] N. Megiddo. "Applying parallel computation algorithms in the design of serial algorithms. " *Journal of the ACM*, 30, 4, Oct. 1983, pp. 852-865.

[MR85] G. L. Miller and J. Reif. "Parallel tree contraction and its application. " In *26th IEEE Symp. on Foundations of Computer Science*, 1985, pp. 478-89.

[S78] L. Stewart. *Cographs, A Class of Tree Representable Graphs.* M. Sc. Thesis, Dept. of Computer Science, Univ. of Toronto, TR 126/78, 1978.

[TV84] R. E. Tarjan and U. Vishkin. "Finding biconnected components and computing tree functions in logarithmic parallel time. " In *25th Annual Symp. on Foundations of Comp. Science*, 1984, pp. 12-22.

[V83] U. Vishkin. "Synchronous parallel computation — a survey. " TR-71, Dept. of Computer Science, Courant Institute, NYU, 1983.