Dynamic Parallel Tree Contraction[*]
(Extended Abstract)

John H. Reif[†]

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213–2890

Stephen R. Tate

Department of Computer Science
University of North Texas
P.O. Box 13886
Denton, TX 76203–6886

Abstract

Parallel tree contraction has been found to be a useful and quite powerful tool for the design of a wide class of efficient graph algorithms. We propose a corresponding technique for the parallel solution of incremental problems. As our computational model, we assume a variant of the CRCW PRAM where we can dynamically activate processors by a forking operation.

We consider a dynamic binary tree T of $\leq n$ nodes and unbounded depth. We describe a procedure, which we call the *dynamic parallel tree contraction algorithm*, which incrementally processes various parallel modification requests and queries:

(1) parallel requests to add or delete leaves of T, or modify labels of internal nodes or leaves of T, and also

(2) parallel tree contraction queries which require recomputing values at specified nodes.

Each modification or query is with respect to a set of nodes U in T.

Our dynamic parallel tree contraction algorithm is a randomized algorithm that takes $O(\log(|U|\log n))$ expected parallel time using $O(\frac{|U|\log n}{\log(|U|\log n)})$ processors. We give a large number of applications (with the same bounds), including:

(a) maintaining the usual tree properties (such as number of ancestors, preorder, etc.),

(b) Eulerian tour,

(c) expression evaluation,

(d) least common ancestor, and

(e) canonical forms of trees.

Previously, there where no known parallel algorithms for incrementally maintaining and solving such problems in parallel time less than $\Theta(\log n)$.

In deriving our incremental algorithms, we solve a key subproblem, namely a *processor activation problem*, within the same asymptotic bounds, which may be useful in the design of other parallel incremental algorithms. This algorithm uses an interesting persistent parallel data structure involving a non-trivial construction.

In a subsequent paper, we apply our dynamic parallel tree contraction technique to various incremental graph problems: maintaining various properties, (such as coloring, minimum covering set, maximum matching, etc.) of parallel series graphs, outerplanar graphs, Helin networks, bandwidth-limited networks, and various other graphs with constant separator size.

1 Introduction

Parallel tree contraction is broadly applicable technique for the parallel solution of a large number of tree problems, and is used as an algorithm design technique for the design of a large number of parallel graph algorithms. Parallel tree contraction was introduced by Miller and Reif [12], and has subsequently been modified (to improve efficiency and/or simplify explanation) by He and Yesha [8], Gazit, Miller, and Teng [7], and Kosaraju and Delcher [11], among many others. A textbook description is given by JáJá [9], and an excellent survey presentation is given by Karp and Ramachandran [10].

Tree contraction has been used in designing many efficient parallel algorithms, including expression evaluation, finding least common ancestors, tree isomorphism, maximal subtree isomorphism, common subexpression elimination, computing the 3-connected components of a graph, and finding an explicit planar embedding of a planar graph.

In this paper, we are concerned with dynamic problems on trees. In dynamic problems, we are given an initial tree T, and then an on-line algorithm processes requests regarding T. Requests may be either incremental changes to T or requests for certain values computed using the tree. A simple example is maintaining the pre-order numbering on a tree. The on-line algorithm would then have to handle incremental changes to the tree, and would also have to quickly answer queries about the pre-order number of any tree node. Our dynamic algorithms are based on the parallel tree contraction process, and hence we call such algorithms *incremental tree contraction algorithms.* By maintaining the connection between our incremental algorithms and the parallel tree contraction algorithm, we use the vast amount of previous work in parallel algorithms that use tree contraction to create incremental algorithms for a wide range of problems.

[*]This research was supported by DARPA/ISTO Grant N00014-91-J-1985, Subcontract KI-92-01-0182 of DARPA/ISTO prime Contract N00014-92-C-0182, NSF Grant NSF-IRI-91-00681, and NASA subcontract 550-63 of prime Contract NAS5-30428.

[†]Permanent address: Department of Computer Science, Duke University, Box 90129, Durham, NC 27708–0129

1.1 Previous sequential incremental algorithms

Incremental algorithms respond to requests to either change a data structure slightly, or answer queries to the data structure. We make a distinction between quantities that we *exactly maintain*, and quantities that we *incrementally maintain*. A quantity is exactly maintained if after each incremental step of the algorithm there are variables containing *exactly* the value that we are interested in. In other words, to find the value, an algorithm only needs to read the appropriate variable — no computation is necessary. For example, in an algorithm in this paper we will exactly maintain the number of descendants in a certain tree; thus, after each step, each node has a variable that says exactly how many descendants are below that node.

On the other hand, some quantities are *incrementally maintained.* These values are not stored explicitly, but can be quickly calculated when needed. For example, it is difficult to exactly maintain the pre-order numbering of a tree because one small change to the tree can affect the pre-order numbering of $\Omega(n)$ nodes in the tree. However, by exactly maintaining the number of descendants in a balanced tree, we can quickly (in $O(\log n)$ sequential time) compute the current pre-order number of any node in the tree.

Previous work on maintaining dynamic trees has been done by Sleator and Tarjan [16] and by Frederickson [5, 6]. In particular, Frederickson notes that his algorithm for dynamic tree maintenance clusters tree nodes in a manner very similar to that of tree contraction, and performs sequential updates in $O(\log n)$ time. The problem of maintaining dynamic expression trees was studied by Cohen and Tamassia [3], who gave an algorithm with $O(\log n)$ update and query time. Fredrickson applies his dynamic tree data structure to many interesting dynamic graph problems, including dynamic expression evaluation [6], giving $O(\log n)$ bounds for all incremental requests. In fact, his algorithm does cluster nodes in a similar manner to the original versions of tree contraction (those with both rake and compress operations), but in this paper we consider clustering nodes in a manner more similar to the more recent, and simpler version of tree contraction due to Kosaraju and Delcher [11]. Our data structure is in turn considerably simpler and easier to maintain than the dynamic tree structures of Frederickson. Furthermore, by by making the connection between tree contraction and our dynamic tree maintenance algorithms clear, we are able to easily apply our incremental tree contraction procedure to the wide variety of problems that have used standard parallel tree contraction.

1.2 The largely unexplored problem of parallel incremental algorithms

In this paper, we consider the largely unexplored problem of performing incremental updates on a dynamic tree using a parallel machine (a CRCW PRAM) for the updates. In fact, we consider the more general case where a set of updates is to be performed concurrently by a parallel machine. We give algorithms that can perform a set of updates in $O(\log(|U|\log n))$ expected time with $O(\frac{|U|\log n}{\log(|U|\log n)})$ processors, where $|U|$ denotes the size of the set of concurrent updates that have been requested. For a constant number of updates (i.e., when $|U| = O(1)$), notice that the updates are performed in $O(\log\log n)$ expected time using $O(\frac{\log n}{\log\log n})$ processors. Also notice that with the known sequential algorithms, a sequence of $|U|$ queries or update requests takes $O(|U|\log n)$ time, so our parallel algorithms are work-optimal with respect to these bounds.

1.3 Dynamic parallel tree contraction

We consider a dynamic binary tree T of $\leq n$ nodes and unbounded depth. We define a procedure, namely the dynamic parallel tree contraction algorithm, which incrementally processes parallel requests to add or delete leaves of T, modify labels of internal nodes or leaves of T, and also incrementally processes parallel tree contraction queries that recompute values at specified nodes. Each modification or query is with respect to a set of parallel update requests specified at a set of nodes U in T.

Our dynamic parallel tree contraction algorithm will initially need to solve the following incremental problem. Given a tree PT of size n, and a small set of leaves U, define the parse tree of U (denoted $PT(U)$) to be the set containing all of the leaves in U, and all of their ancestors. We need to maintain data structures so that, given any set U, we can quickly identify and activate processors for the nodes of $PT(U)$.

We will show how to solve this problem in Section 2, and how to solve the remaining problems required for dynamic parallel tree contraction in Section 4. The result is a randomized algorithm that runs in $O(\log(|U|\log n))$ expected time, using $O(\frac{|U|\log n}{\log(|U|\log n)})$ processors.

1.4 The Self-Healing Paradigm: Terminology for Dynamic Algorithms Based on Tree Contraction

For our dynamic algorithms, we use a scenario, concepts and terminology which we will we borrow from the movie *Terminator 2*. In that movie, there is a robot made of a liquid metal. The robot exhibits a very interesting self healing property, which can be adopted to dynamic algorithms. Projectiles which entering the robot may cause wounds running completely through the robot. Nevertheless, the robot rapidly restructures itself to adapt to the change, and heal the wound (the response to the attack is rapid as well).

Our deterministic dynamic algorithms based on tree contraction have the property that a set of parallel requests (insertions or deletions) may require the tree to be restructured. The parts to be restructured form a subtree of the parse tree. We will call the parts to be restructured the *wound.*

(Step 1) *Wound Location and Process Activation:*

The first step will be to identify the location of the wound, which (in dynamic tree contraction in processing the query) can run from a query node up to the root. Also in this step we must activate processes which will be used in subsequent steps (this step is harder than might be expected, and is solved in Section 2).

(Step 2) *Wound Healing*

The next step in processing the query will be to heal the wound using the processes activated in step 1. In our case, this is the process of restructuring the parallel tree contraction parse tree. This will be done by a call to a our dynamic parallel tree contraction algorithm, as described in Section 4, using the processes activated in step 1.

(Step 3) *Answering the Attack*

The final step in processing will be to respond to the query. In our case we will need simply to re-evaluate the fragment of the parallel tree contraction parse tree healed (restructured) in step 2. This can be done in our case by the usual (non-dynamic) parallel tree contraction using the processes activated in step 1.

The above concept, scenario, and terminology are quite useful in the case where we have a dynamic processes which persists indefinitely, and must heal itself in a dynamic manner. Note: We feel it is important that a commonly used, widely applicable terminology be adopted for processes such as described above.

Note that the recent papers of Armon and Reif [2], and Reif, Spirakis, and Yung [15] use the replicant terminology of the movie *Blade Runner* for description of a (rather different) class of randomized dynamic algorithms where processes cycle through periods of reincarnation: activation, death, total rebuilding, and reactivations.

2 The Random Splitting Tree

A *binary splitting tree* (BST) is a binary tree in which each node has either zero or two children (also called a full binary tree). Many $O(\log n)$ time parallel algorithms have an underlying computational structure that resembles such a tree, and either proceeds from the root of the tree to the leaves (as in quicksort or flashsort), or from the leaves to the root (as in an n element summation), or perhaps even both (the contraction and expansion phases of tree contraction). By considering how to maintain such a splitting tree under incremental changes, we can hope to derive parallel incremental algorithms for a large class of problems.

Given a BST PT with n leaves (which we will call a *parse tree* for reasons that become apparent later), and a subset U (called the *update set*) of the leaves, define the *parse tree of the set* U (denoted $PT(U)$) to be the subtree of PT that is made up of the leaves in U and all of the ancestors of nodes in U. It is fairly easy to see that for balanced trees, $PT(U)$ cannot be too large — in fact, if the depth of PT is $O(\log n)$, then we can bound the size by $|PT(U)| = O(|U| \log n)$.

We would like to perform operations on $PT(U)$ in parallel, using only $O(\frac{|U|\log n}{\log(|U|\log n)})$ processors, which raises the *processor allocation problem*: namely, in a tree of size $O(n)$ how can we quickly activate a small set (size $O(\frac{|U|\log n}{\log(|U|\log n)})$) of processors, one for each node in the parse tree (by quickly, we mean $O(\log(|U|\log n))$ time). For example, consider the case in which U contains a single leaf of a balanced tree, and so $PT(U)$ consists of the $O(\log n)$ nodes on the path from this leaf to the root. If we have no supplemental information about our tree T, then the best we can do is follow the parent links, giving an $\Theta(\log n)$ time, or $\Theta(|U|\log n)$ time algorithm (in particular, we cannot do the standard "pointer jumping" because we don't know which set of $O(\log n)$ nodes are involved in the operations — in fact, identifying these nodes is our goal!).

To solve the processor activation problem, we supplement our tree with the following information: at each node v, store a flag ACTIVE_v which is initially 0 for all nodes (the purpose of this flag will be explained later), the depth of the node d_v (the root has depth 0), the number n_v of nodes in the subtree rooted at v, and for every node whose height is greater than $\log\log n$ we store an array with $m_v = \lfloor \log d_v \rfloor$ entries $s_{v,1}, s_{v,2}, \cdots, s_{v,m_v}$ (called shortcuts), where $s_{v,i}$ is the unique ancestor of v such that

$$d_{s_{v,i}} = \left\lceil d_v \left(1 - \left(\frac{1}{2}\right)^i\right)\right\rceil .$$

For uniformity of later arguments, we assume that $s_{v,0}$ is the root node of the tree T. Note that if only $O(\frac{n}{\log\log n})$ nodes of a depth $O(\log n)$ tree T have height greater than $\log\log n$ (as is true in most trees), then all of this information can be stored in $O(n)$ space. We call this data structure a *binary splitting tree with shortcuts* (BSTS), and it can be used to solve the processor activation problem as described in the following theorem.

Theorem 2.1 *Given a $O(\log n)$ depth BSTS and a set of nodes U, we can identify the nodes of $PT(U)$ in parallel time $O(\log(|U|\log n))$ using $O(\frac{|U|\log n}{\log(|U|\log n)})$ processors.*

Proof: Initially, we start out with $|U|$ processors active, each processor associated with an element of U. As a first stage, every processor follows parent pointers up the tree, setting the ACTIVE flag for each node visited, until it finds a node that contains a shortcut list. This uses $O(|U|)$ processors and $O(\log\log n)$ time.

The next stage of the activation process uses the shortcut information to quickly identify the remaining nodes of $PT(U)$. Processors will be activated by a forking procedure — for example, a processor may discover (by a method to be described later) a node of $PT(U)$ that has not been previously identified, and so will start a new processor that will be associated with that node. A processor can be activated for a tree node v only if the corresponding ACTIVE_v flag is 0, and when the processor is started ACTIVE_v is set to 1. After an update set U is fully processed, and the processors of $PT(U)$ are being deactivated, each processor resets it's ACTIVE flag to 0 for the next round.

At time t of the startup procedure, each processor manipulates a range of depth values denoted $\ell_{v,t} \cdots u_{v,t}$, where each processor initially starts off with range $\ell_{v,0} = 0 \cdots d_v = u_{v,0}$. Each node also maintains a position in its shortcut list $p_{v,t}$, initialized to $p_{v,0} = 0$. At every step, we will maintain the property that $\ell_{v,t} = d_{s_{v,p_{v,t}}}$. To advance the startup procedure by one step, we set $p_{v,t+1} = p_{v,t} + 1$ and set $\ell_{v,t+1}$ to the corresponding value. Next, we activate a processor for node $w = s_{v,p_{v,t+1}}$ (if necessary). For the new node and processor, initialize $u_{w,t+1} = \ell_{v,t+1}$, and initialize $\ell_{w,t+1}$ by setting $p_{w,t+1}$ to the unique value k such that

$$d_{s_{w,k}} \leq \ell_{v,t} \text{ and } d_{s_{w,k+1}} > \ell_{v,t}.$$

In essence, we are taking a range of depths, and starting a new processor to activate nodes in the smallest half of the depths.

It can easily be shown that $p_{w,t+1}$ can only be $p_{v,t}$ or $p_{v,t+1}$, so $p_{w,t+1}$ can be found in constant time by checking both of the possible values. At time $t+1$, the range of depths for both v and w are at most $2/3$ of the range of v at time t, so consequently the largest range present in the tree goes down by a constant factor at each step. We repeat this basic step until every range contains at most $\log(|U|\log n)$ values. Now each processor can sequentially traverse up the tree to the next higher activated location, marking nodes as being in $PT(U)$ as it goes, which takes at most $\log(|U|\log n)$ steps. Thus, it follows that for any leaf $v \in U$, all of the $d_v + 1$

nodes on the path from v to the root get identified within time

$$O\left(\log\left(\max_{v\in U} d_v\right) + \log(|U|\log n)\right) = O(\log(|U|\log n)).$$

Furthermore, leaf $v \in U$ starts at most $O(\frac{d_v}{\log(|U|\log n)})$ processors, so the total number of processors used by this procedure is $O(\frac{|U|\log n}{\log(|U|\log n)})$. ▮

We can define a probability distribution on binary splitting trees by the following construction procedure: For the n leaves $v_1, v_2, \cdots, v_n$, pick a random integer k in the range $1..n-1$. Create a node w (this will be the root of the BST), and split the leaves into two sets $(v_1, \cdots, v_k)$ and $(v_{k+1}, \cdots, v_n)$. Recursively construct trees for these two sets, and the roots of the constructed trees will be the children of node w. If, at some point in time, a BST is really a random tree with exactly this distribution, then we call the tree a *random binary splitting tree* (RBST); furthermore, if the tree is a random variable with this distribution and the shortcut information is present in the tree, we call the tree a *random binary splitting tree with shortcuts* (RBSTS). We relax the condition on where shortcut information must be in a RBSTS — shortcut information is only required in nodes with subtrees of depth at least $2\log\log n$, and should *not* be in nodes with depth less than $\frac{1}{2}\log\log n$. Note that a RBST or a RBSTS with n leaves has expected depth $O(\log n)$. We show below that we can efficiently construct a RBSTS from a list of leaves, including computing all of the shortcuts.

Lemma 2.1 *Given a list of n values, we can construct a RBSTS in $O(\log n)$ expected time using $O(\frac{n}{\log n})$ processors.*

Proof: The construction proceeds in two stages: building the tree, and making the shortcut lists. Building the tree is a straightforward procedure in which new processors are started for each subtree, until at most $\frac{n}{\log n}$ processors have been started. At this point, each remaining subtree has expected size $O(\log n)$, so the remaining splittings are done sequentially within each of these subtrees. Once the tree is constructed, tree contraction can be performed on the tree to optimally compute the depth of the subtree rooted at each node.

Next, stage 2 will construct a shortcut list for every node that is the root of a subtree of depth greater than $\log\log n$. To see how to construct a shortcut list efficiently, consider the shortcut lists for two nodes v and w, where v is a parent of w. Since $d_w = d_v + 1$, it must be true that for all shortcut entries j, $s_{w,j}$ can only be either $s_{v,j}$ or $s_{v,j}+1$. Thus, one time step after v has computed $s_{v,j}$, node w can, in constant time, compute $s_{w,j}$. Computation of the shortcut list for nodes at depth d is started at time-step d of stage 2, and a node at depth d can immediately start computing its shortcut values from the values already computed at its parent. The total time for a node on level d to initialize its shortcut list is $d + \log d$ — since the expected depth of the tree is $O(\log n)$, the total expected time for stage 2 is $O(\log n + \log\log n) = O(\log n)$. The total number of processors required is the total expected number of nodes with subtrees of depth greater than $\log\log n$, which is $O(\frac{n}{\log n})$. ▮

Note that Theorem 2.1 applies to a RBSTS, so we may quickly identify and activate parse trees in a RBSTS. The benefit of the randomized structure is that a RBSTS is easy to dynamically maintain, as we will explain next.

First, we consider adding new leaves to a RBSTS, and maintaining all of the shortcut and supplemental information, in addition to maintaining the correct distribution on the set of possible BST's. The idea is essentially this: if a new leaf z is inserted between leaves v_k and v_{k+1}, then with probability $\frac{n-1}{n}$ simply recursively insert z in the subtree that contains v_k. However, with probability $\frac{1}{n}$ throw away the old tree structure and build a new tree RBSTS with root w and subtrees containing the leaves $(v_1, \cdots, v_k)$ and $(z, v_{k+1}, \cdots, v_n)$. In the worst case, this procedure can be very expensive, but the worst case only occurs with probability $\frac{1}{n}$; the following theorem generalizes the addition procedure to the insertion of a set of nodes, and shows that the expected running time is small.

Theorem 2.2 *A set U of new leaves can be inserted into an existing RBSTS in $O(\log(|U|\log n))$ expected time using $O(\frac{|U|\log n}{\log(|U|\log n)})$ processors, resulting in a grown, valid RBSTS with high probability.*

Proof: (Sketch) Let S a random variable representing the size of the subtree to be rebuilt. By Lemma 2.1, we can perform the tree restructuring in expected time $O(E[\log S])$ using an expected number of processors of $O(E[\frac{S}{\log S}])$. Since the both of these functions are concave in S, we know that $E[\log S] \le \log(E[S])$ and $E[\frac{S}{\log S}] \le \frac{E[S]}{\log(E[S])}$, so we will concentrate on finding $E[S]$.

For any node $v \in PT(U)$, recall that n_v is the number of descendants of v in T (the full tree, not just $PT(U)$). From the description of the tree restructuring procedure, the probability of restructuring the subtree rooted at v is at most $1/n_v$, and the number of nodes involved in such a restructuring would be n_v. Therefore,

$$E[S] \le \sum_{v\in PT(U)} \frac{1}{n_v}\cdot n_v = |PT(U)| \le |U|\log n,$$

which is exactly the bound we need to prove our theorem.

There is one problem with this construction, as exemplified by the following case: consider many additions to the right side of a tree. When the left side of the tree is initially created, the tree has a certain number of nodes, say N, and the set of nodes containing shortcut information is determined from this value (i.e., the nodes with shortcut information are those whose subtrees have depth at least $\log\log N$). Now after additions to the right side, there are $n > N$ nodes in the tree, and if the left side of the tree has not been rebuilt, there are shortcuts on nodes with subtrees of less than $\log\log n$ depth. However, for shortcuts to exist on nodes with subtrees of less than $\frac{1}{2}\log\log n$ depth, the tree must have grown so that $n > N^{\log N}$ — if the tree grows this much, it will be entirely rebuilt (including the left side) with high probability. ▮

Deletions can be handled similarly, and will be completely described in the full paper. This section can be summarized by the following theorem.

Theorem 2.3 *Given a RBSTS and a set U of leaves, we can (a) identify the parse tree for U and activate $\frac{|U|\log n}{\log(|U|\log n)}$ processors, (b) add new leaves at the positions of U, or (c)* *delete the leaves in U, all in expected time $O(\log(|U|\log n))$, using $O(\frac{|U|\log n}{\log(|U|\log n)})$ processors. In all cases, a valid RBSTS is output with high probability.*

3 Incremental List Prefix

Before considering the more difficult problem of incremental tree contraction, we introduce the ideas that we will be using by considering the easier problem of incremental list prefix sum. The incremental list prefix problem is the prefix sum problem, where the input is a linked list of nodes containing the input values (we use this terminology rather than "list ranking" to avoid confusion with the fact that list ranking usually refers to computing *suffix* sums). We will see that this problem is easy, given the dynamic random splitting tree methods of the previous section.

To solve the incremental list prefix problem, we maintain a RBSTS in which the linked list nodes are the leaves of the RBSTS, and we maintain additional information at each node of the RBSTS. In particular, any internal node v of the RBSTS corresponds to a sub-list of values from the maintained linked list (i.e., the leaves of the subtree rooted at that node), and we store the sum of all the values in that sub-list at the internal node (call this SUM_v). Given this information, it is easy to answer in parallel requests for the prefix sum at a set of nodes U.

In particular, we first identify the parse tree $PT(U)$ for the set U, as described in the preceding section. Next we will build a secondary parse tree, $\hat{PT}(U)$, that is an extension of $PT(U)$ (this tree is really a conceptual help, and doesn't have to actually be constructed). Every node $v \in PT(U)$ has two children in the full tree T, call them w_1 and w_2, at least one of which must be in $PT(U)$. If one of the children, say w_1, is not in $PT(U)$ we add w_1 as a child of v in $\hat{PT}(U)$ — w_1 is a leaf node of $\hat{PT}(U)$, and we give it the value SUM_{w_1}. In effect, this one leaf node replaces the entire subtree rooted at w_1 as far as this parse tree is concerned. The extended tree $\hat{PT}(U)$ has at most twice as many nodes as $PT(U)$, so $|\hat{PT}(U)| = O(|PT(U)|)$.

To compute all the requested prefix sum values, first use the Euler tour technique to obtain an ordered list of the leaves of $\hat{PT}(U)$. This requires $O(\log|\hat{PT}(U)|)$ time and $O(|\hat{PT}(U)|)$ work. Due to the way that $\hat{PT}(U)$ was constructed, performing the standard prefix sum algorithm on this list will give the desired prefix sums for the nodes in U. The complexity of this step is the same as the complexity of computing the list of leaves (and in fact could be done as a side-effect of the Euler tour computation), so once the parse tree is identified, the prefix sums can be computed in deterministic time $O(\log|PT(U)|)$ with $O(|PT(U)|)$ work.

It remains to be seen how to exactly maintain the SUM_v values, which we now describe. When a sub-tree is rebuilt, computation of the SUM_v values can be done by performing a tree contraction and expansion on the re-built tree. For a size S tree, this can be done in $O(\log S)$ time using $O(S)$ work. In the RBSBS maintenance routines, we saw that $E[S] \leq |U|\log n$, so this step can be done in $O(\log(|U|\log n))$ expected time, with $O(\frac{|U|\log n}{\log(|U|\log n)})$ processors. In addition, whenever leaf values are changed we need to recompute all the SUM_v values on the path from the updated leaves to the root. However, this can easily be done by performing a tree contraction and expansion on the extended parse tree $\hat{PT}(U)$.

The results of this section can be summed up in the following theorem.

Theorem 3.1 *We can perform a set of concurrent queries or updates on a set of nodes U of our incremental list prefix data structure in $O(\log(|U|\log n))$ expected time using $O(\frac{|U|\log n}{\log(|U|\log n)})$ processors.*

Note: Notice that if $|U| = O(1)$ then the update is performed in $O(\log\log n)$ expected time using $O(\frac{\log n}{\log\log n})$ processors.

4 Dynamic parallel tree contraction

The incremental tree contraction algorithms that we design are based on the parallel tree contraction algorithm of Kosaraju and Delcher [11] (also see [9] for a textbook description of tree contraction). This algorithm operates by finding an Euler tour of the expression tree, performing a list ranking to order the leaves of the tree from left to right, and then repeatedly performs a rake on the leaves in odd numbered positions of this ordering. To rake a leaf, both the leaf and it's parent are removed from the tree, and the value of the leaf's grandparent is updated to reflect the removal of the these two nodes. Since only leaves in odd numbered positions are raked by this algorithm, it is guaranteed that no two sibling with both try to simultaneously rake and remove their common parent. Notice that in this step, both a leaf and an internal node are removed from the tree, which is why the tree size is reduced at such a rapid rate. After this step, only half of the original leaves remain, and the process is repeated using the leaves in odd numbered positions on this new, smaller set of leaves. Repeating this $O(\log n)$ times reduces the tree to a single node. Kosaraju and Delcher show how to update the sibling label during a rake so that after the entire procedure the single remaining node is labeled with the value of the entire expression tree. For a tree with n nodes, the contraction takes a total of $O(\log n)$ time with $O(n/\log n)$ processors. This process is known as the CONTRACTION phase of the parallel tree contraction process.

Most tree contraction algorithms require a second phase, the EXPANSION phase. The EXPANSION phase can be viewed as the logical reversal of the CONTRACTION phase. In particular, the rake operations are done in reverse order, with values propagating down the tree from the root. After the EXPANSION phase, the original tree is entirely reconstructed, with each internal node labeled with the value of the sub-expression rooted at that node. For complete details, see [11].

4.1 Overview of dynamic parallel tree contraction

We consider a dynamic binary tree T of $\leq n$ nodes and unbounded depth. We define a procedure, namely the incremental parallel tree contraction algorithm, which incrementally processes requests to modify or query T, where the requests can be any of the following.

- Add two new children below a current leaf.
- Delete two leaf children of a node.
- Modify labels of internal nodes or leaves of T.

- Processes parallel tree contraction queries that recompute values at specified nodes.

Each modification or query is with respect to a set of parallel update requests specified at a set of nodes U in T.

The dynamic parallel tree contraction algorithm maintains a *contraction parse tree* PT with leaves that correspond to the nodes of the expression tree T. The updates will be done on the subtree $PT(U)$ induced from PT consisting of the paths from the root to each node in U. Again, note that $PT(U)$ denotes exactly those nodes that may be wounded by an update request. These updates can cause the incremental tree contraction algorithm to add or delete at most U nodes. Our final result is stated below, which we will prove in the next subsection.

Theorem 4.1 *The dynamic parallel tree contraction algorithm takes $O(\log(|U| \log n))$ expected parallel time, using $O(\frac{|U| \log n}{\log(|U| \log n)})$ processors.*

4.2 Details of Dynamic Parallel Tree Contraction

In this section we extend the ideas of a RBSTS to the more complex problem of incremental tree contraction. First, consider a randomized version of the tree contraction algorithm of Kosaraju and Delcher [11] that works as follows. First, create a list of the leaves of the tree T in left to right order. Create a RBSTS (call it PT) for the set of leaves, and we will use the RBSTS to guide the sequence of rakes on T. In particular, we consider the set S of nodes of the RBSTS that have two leaf nodes (in PT) as children. Let L be the set of left children from S, and we will rake the corresponding nodes in the original tree T. Now we remove all nodes in S from PT, making each exposed parent node correspond to the unraked right child. Similar to Kosaraju and Delcher's algorithm, this can never rake two siblings in one time step, so is a valid rake sequence. Furthermore, since one level of PT is removed at each step, the number of parallel steps is exactly the depth of PT, which has expected value $O(\log n)$. It should be noted that by first raking subtrees of size $O(\log n)$ consecutive nodes sequentially, using $O(n/\log n)$ total processors, we can make this an expected work-optimal tree contraction algorithm. We can turn this into an incremental algorithm by maintaining the RBSTS as described in Section 2.

We maintain a second data structure called the *rake tree* (denoted RT) to keep track of how the tree evaluation labels are manipulated by the tree contraction algorithm. Note that in the list prefix data structure, the required computations are easily derived from the RBSTS; however, for tree contraction this is not the case. In tree contraction, we must keep track of labels of internal nodes of T, which do not even exist in the RBSTS (recall that only leaves of T are represented in the RBSTS). For incremental tree contraction, the rake tree exactly reflects the changes to internal node labels that need to occur in order to heal a wounded tree.

Consider a single rake of the original tree contraction algorithm to be in two parts. Let v be the node we are raking, and let p denote its parent and w denote its sibling. First, we do what is called a "small-rake", where v is raked into its parent p, and the label of p is updated accordingly. Secondly, p is removed by merging it with w, and the label of w is updated to reflect this contraction. We expand rakes like this so that all operations on labels during the tree contraction phase are binary. The rake is now a binary operation on the labels of v and p, followed by a binary operation on the label of w and the new label of p.

The rake tree is structured as follows. Any time the tree contraction algorithm modifies a node's label, it must be by a binary operation on two labels (by the preceding paragraph) so join the rake tree nodes corresponding to those two labels under a single parent node, and the parent node corresponds to the new label produced by the binary operation. The node is then labeled with the function that produces the new label from the old ones. This is clearly a binary tree, and there is a one-to-one correspondence between the nodes of this tree and all the labels assigned by the tree contraction algorithm. Since each internal node of RT corresponds to a rake operation, and each left child in the RBSTS also corresponds to a rake operation, we maintain links between the rake tree nodes and the nodes of the RBSTS. Thus by identifying a wound in the RBSTS, we can in parallel quickly discover the corresponding wounded nodes in RT.

Clearly, a tree evaluation on the rake tree will compute the values of the labels of all nodes in T at all times during the tree contraction process. To be able to perform concurrent updates, we will have to show that the functions labeling the nodes of RT are valid tree contraction operations, and we show this in the proof of the following theorem.

Theorem 4.2 *We can perform a set of updates on a set of nodes U of our incremental tree contraction data structure in $O(\log(|U| \log n))$ expected time using $O(\frac{|U| \log n}{\log(|U| \log n)})$ processors. We can also perform a single update with a single processor in $O(\log n)$ time.*

Proof: In this proof, we prove the more difficult of the statements in the theorem — that concurrent updates may be done in parallel. The sequential algorithm is much simpler: just start at the leaves and propagate the simple changes that are required toward the root.

We perform a parallel update on our data structure in two phases. First, we must locate the wound (i.e., $PT(U)$) in the RBSTS. This phase proceeds exactly as in the incremental list ranking problem. In particular, adding a new leaf to the tree T involves adding a new leaf to the RBSTS, which is accomplished as described in Section 2. It is in this phase that processors get allocated to the parts of the tree that will be changing, exactly as in the case of the list prefix problem. These processors remain activated after this phase to complete the relabeling described below. It is trivial to make the corresponding updates to RT, once $PT(U)$ has been activated.

Note that if the request to the incremental algorithm is either a query or simply a node label update, then there are no structural changes to the RBSTS or the rake tree. In this case, the first stage simply identifies the wound and activates processors for the second phase.

When the first phase is over, we have done all the required updates to the structure of the RBSTS and the rake tree, but the labels in RT may need to be updated. Let W denote the set of all wounded nodes in RT, and let $RT(W)$ denote the subtree of RT consisting of all paths from a node of W to the root of RT. For a node v not in $RT(W)$, neither the node itself or any of its descendants have changed, so the label of v can not be changed by this update. In other words, all the changes in RT due to an incremental update occur in the subtree $RT(W)$.

For any node w in $RT(W)$, we first make sure both of its children are in $RT(W)$. If they are not, then they are added as new leaves of $RT(W)$ — these leaves are labeled with their values from the previous incremental step, and as we have shown that these values cannot be changed in the current step. It is clear then that evaluating the resulting tree will give the correctly updated labels for all nodes in $RT(W)$, and thus all of RT will have correct labels. This can easily be done sequentially in optimal $O(|RT(W)|)$ time. We next show that the tree evaluation can be done by parallel tree contraction.

We consider the case of T being over a commutative ring (which is the case for the vast majority of tree contraction applications), like in the paper of Kosaraju and Delcher [11]. In such a case, the label at each node of the tree T consists of an operation label (which never changes), and a pair (A, B). The meaning of this label is that if x is the value of the subtree rooted at this node, then $Ax + B$ is the value of this contracted node. Initially, all internal nodes are given the pair $(1, 0)$ as a label, and all leaves are given the pair $(0, v)$, where v is the value of that leaf in the expression tree.

There are three basic label manipulation functions that label the internal nodes of RT. First, if p is an internal node of RT with children v (the leaf being raked) and w, and p corresponds to a "small-rake", then the exact operation depends on the ring operation op_w labeling w in T. Let (A, B) be the label of v and (C, D) be the label of w. Then if op_w is an addition, the function labeling node p takes (A, B) and (C, D) as input, and produces label $(C, CB+D)$. If op_w is a multiplication, then the function must produce label (CB, D). Finally, if p is corresponds to a "small-compress", and has children v (the node in T being removed) and w with labels (A, B) and (C, D), respectively, then the label update function must produce the new label $(AC, AD + B)$.

The important point to notice about all the label update functions is that the function for each label component is linear in the input components. Since composition of linear functionals is associative, this is all we need in order to be able to perform tree contraction on RT. Thus we can recompute all the changed labels of RT in $O(\log |RT(W)|) = O(\log(|U| \log n))$ time using $O(\frac{|RT(W)|}{\log |RT(W)|}) = O(\frac{|U| \log n}{\log(|U| \log n)})$ processors. ▮

5 Applications of Dynamic Parallel Tree Contraction

Here we show that dynamic parallel tree contraction is a broadly applicable technique for the design of dynamic parallel algorithms. Standard parallel tree contraction has been shown to provide a basis for many efficient parallel algorithms, and we can use many of the reductions to the tree contraction problem in designing dynamic algorithms. The following theorems are a sampling of the results derived using this technique.

Theorem 5.1 *Incrementally maintaining the standard tree properties, (such as preorder, number of ancestors), as well as Eulerian tour and expression evaluation using our dynamic parallel tree contraction algorithm takes expected parallel time $O(\log(|U| \log n))$ using $O(\frac{|U| \log n}{\log(|U| \log n)})$ processors.*

Theorem 5.2 *Incrementally maintaining least common ancestor and canonical forms of trees using our incremental parallel tree contraction algorithm takes $O(\log(|U| \log n))$ expected parallel time using $O(\frac{|U| \log n}{\log(|U| \log n)})$ processors.*

6 Further Work

In a subsequent paper, we apply our dynamic parallel tree contraction technique to various incremental problems on graphs with constant separator size, for example: parallel series graphs, outerplanar graphs, Helin networks, bandwidth-limited networks, etc. In these graph problems, we incrementally maintain in parallel various properties, such as coloring, minimum covering set, maximum matching, etc. some of which are NP complete for general graphs.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] D. Armon and J.H. Reif, "Strictly Polylog Time, Linear Space Algorithms for Nearest Neighbor Search and Dynamic Separators in d Dimensions", manuscript, Oct 1992

[3] R. F. Cohen and R. Tamassia. "Dynamic Trees and Their Applications," *2nd ACM-SIAM SODA*, pp. 52–61, 1991.

[4] R. Cole and U. Vishkin. "Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking", *Inform. and Control*, Vol. 70, pp. 32–53, 1986.

[5] G. N. Frederickson. "Ambivalent Data Structures for Dynamic 2-Edge-connectivity and k Smallest Spanning Trees", *FOCS*, pp. 632–641, 1991.

[6] G. N. Frederickson. "A Data Structure for Dynamically Maintaining Rooted Trees", to appear in *SODA*, Jan. 1993.

[7] H. Gazit, G. L. Miller, and S. H. Teng. "Optimal Tree Contraction in the EREW Model", in *Concurrent Computations: Algorithms, Architecture, and Technology*, Plenum, New York, pp. 139–156, 1988.

[8] X. He and Y. Yesha. "Binary Tree Algebraic Computation and Parallel Algorithms for Simple Graphs", *Journal of Algorithms*, Vol. 9, pp. 92–113, 1988.

[9] J. JáJá, *An introduction to parallel algorithms,* Addison-Wesley, 1992.

[10] R. M. Karp and V. Ramachandran. "Parallel Algorithms for Shared-Memory Machines", in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, Jan Van Leeuwen, ed., The MIT Press, pp. 869–943, 1990.

[11] S. R. Kosaraju and A. L. Delcher, "Optimal Parallel Evaluation of Tree-Structured Computations by Raking", *Proc. 3rd Aegean Workshop on Computing*, Springer Verlag Lecture Notes in Computer Science, Vol. 319, pp. 101–110, 1988.

[12] G. L. Miller and J. H. Reif. "Parallel Tree Contraction Part 1: Fundamentals", in *Randomness and Computation*, Vol. 5, S. Micali, ed., JAI Press, Greenwich, CT, pp. 47–72, 1989.

[13] G. L. Miller and J. H. Reif. “Parallel Tree Contraction Part 2: Further Applications”, *SIAM J. Comput.*, Vol. 20, No. 6, pp. 1128–1147, 1991.

[14] J. I. Munro, T. Papadakis, and R. Sedgewick. “Deterministic Skip Lists”, *SODA*, pp. 367–375, 1991.

[15] J.H. Reif, P. Spirakis, M. Yung, ”Re-Randomization and Average Case Analysis of Fully Dynamic Graph Algorithms ”, manuscript, Oct, 1992.

[16] D. Sleator and R. E. Tarjan. “A Data Structure for Dynamic Trees”, *J. Comput. Sys. Sci.*, Vol. 26, pp. 362–391, 1983.