PARALLEL AND DISTRIBUTED ALGORITHMS
BY
DEBDEEP MUKHOPADHYAY
AND
ABHISHEK SOMANI

http://cse.iitkgp.ac.in/~debdeep/courses_iitkgp/PAlgo/index.htm

THE EULER TOUR TECHNIQUE: EVALUATION OF TREE FUNCTIONS

2

OVERVIEW

Tree contraction

Evaluation of arithmetic expressions

PROBLEMS IN PARALLEL COMPUTATIONS OF TREE FUNCTIONS

Computations of tree functions are important for designing many algorithms for trees and graphs.

Some of these computations include *preorder*, *postorder*, *inorder* numbering of the nodes of a tree, number of descendants of each vertex, level of each vertex etc.

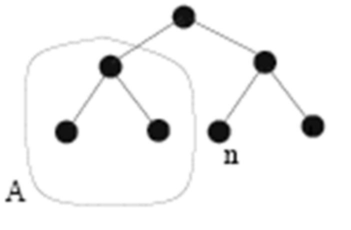
PROBLEMS IN PARALLEL COMPUTATIONS OF TREE FUNCTIONS

Most sequential algorithms for these problems use depth-first search for solving these problems.

However, depth-first search seems to be inherently sequential in some sense.

5

PARALLEL DEPTH-FIRST SEARCH

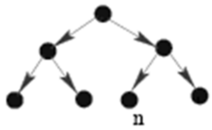


It is difficult to do depth-first search in parallel.

We cannot assign depth-first numbering to the node *n* unless we have assigned depth-first numbering to all the nodes in the subtree *A*.

6

PARALLEL DEPTH-FIRST SEARCH



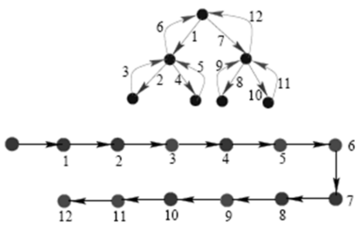
There is a definite order of visiting the nodes in depth-first search.

We can introduce additional edges to the tree to get this order.

The Euler tour technique converts a tree into a list by adding additional edges.

7

PARALLEL DEPTH-FIRST SEARCH

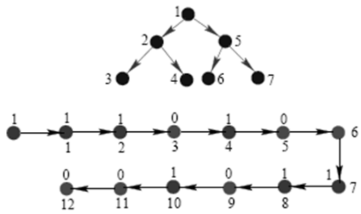


The red (or, magenta) arrows are followed when we visit a node for the first (or, second) time.

If the tree has n nodes, we can construct a list with $2n - 2$ nodes, where each arrow (directed edge) is a node of the list.

8

EULER TOUR TECHNIQUE



For a node $v \in T$, $p(v)$ is the parent of v.

Each red node in the list represents an edge of the nature $< p(v) , v >$.

We can determine the preorder numbering of a node of the tree by counting the red nodes in the list.

9

EULER TOUR TECHNIQUE

Let $T = (V, E)$ be a given tree and let $T' = (V, E')$ be a directed graph obtained from T.

Each edge $(u, v) \in E$ is replaced by two edges $< u, v >$ and $< v, u >$.

Both the indegree and outdegree of an internal node of the tree are now same.

The indegree and outdegree of a leaf is 1 each.

Hence T' is an Eulerian graph: ie. it has a directed circuit that traverses each arc exactly once.

10

EULER TOUR TECHNIQUE

An Euler circuit of a graph is an edge-disjoint circuit which traverses all the nodes.

A graph permits an Euler circuit if and only if each vertex has equal indegree and outdegree.

An Euler circuit can be used for optimal parallel computation of many tree functions.

To construct an Euler circuit, we have to specify the successor edge for each edge.

11

CONSTRUCTING AN EULER TOUR

Each edge on an Euler circuit has a unique successor edge.

For each vertex $v \in V$ we fix an ordering of the vertices adjacent to v.

If d is the degree of vertex v, the vertices adjacent to v are:

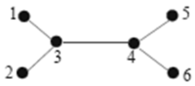
$adj(v) = < u_0, u_1, \dots, u_{d-1} >$

The successor of edge $< u_i, v >$ is:

$s(< u_i, v >) = < v, u_{(i+1) \bmod d} >, 0 \leq i \leq (d - 1)$

12

CONSTRUCTING AN EULER TOUR



v	adj(v)
1	3
2	3
3	2, 1, 4
4	3, 5, 6
5	4
6	4

edge	successor
<3,1>	<1,3>
<3,2>	<2,3>
<2,3>	<3,1>
<1,3>	<3,4>
<4,3>	<3,2>
<3,4>	<4,5>
<5,4>	<4,6>
<6,4>	<4,3>
<4,5>	<5,4>
<4,6>	<6,4>

Successor function table

<3,1> → <1,3> → <3,4> → <4,5> → <5,4> → <4,6>

<3,1> ← <2,3> ← <3,2> ← <4,3> ← <6,4>

The resulting Eulerian Circuit

CORRECTNESS OF EULER TOUR

Consider the graph $T' = (V, E')$, where E' is obtained by replacing each $e \in E$ by two directed edges of opposite directions.

Lemma: The successor function s defines only one cycle and not a set of edge-disjoint cycles in T'.

Proof: We have already shown that the graph is Eulerian.

We prove the lemma through induction.

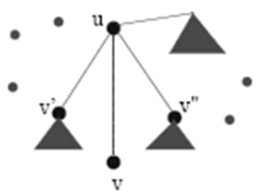
CORRECTNESS OF EULER TOUR

basis: When the tree has 2 nodes, there is only one edge and one cycle with two edges.

Suppose, the claim is true for n nodes. We should show that it is true when there are $n + 1$ nodes.

15

CORRECTNESS OF EULER TOUR



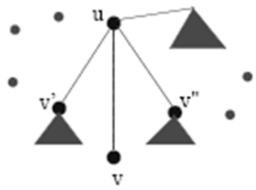
We can introduce an extra node by introducing a leaf to an existing tree, like the leaf v.

Initially, $adj(u) = <\ldots, v', v'', \ldots>$. Hence,

$s(< v', u >) = < u, v'' >$.

16

CORRECTNESS OF EULER TOUR



After the introduction of *v*, *adj*(*u*) = <…, *v*′, *v*, *v*″, …>

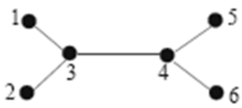
s(< *v*′, *u* >) = < *u*, *v* > and

s(< *v*, *u* >) = < *u*, *v*″ >

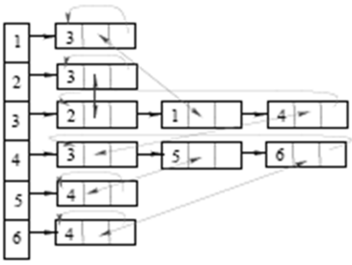
Hence, there is only one cycle after *v* is introduced.

17

CONSTRUCTION OF EULER TOUR IN PARALLEL



v	adj(v)
1	3
2	3
3	2, 1, 4
4	3, 5, 6
5	4
6	4



18

CONSTRUCTION OF EULER TOUR IN PARALLEL

We assume that the tree is given as a set of adjacency lists for the nodes. The adjacency list $L[v]$ for v is given in an array.

Consider a node v and a node u_i adjacent to v.

We need:

- The successor $< v, u_{(i+1) \bmod d} >$ for $< u_i, v >$. This is done by making the list circular.
- $< u_i, v >$. This is done by keeping a direct pointer from u_i in $L[v]$ to v in $L[u_i]$.

19

CONSTRUCTION OF EULER TOUR IN PARALLEL

We can construct an Euler tour in $O(1)$ time using $O(n)$ processors.

One processor is assigned to each node of the adjacency list.

There is no need of concurrent reading, hence the EREW PRAM model is sufficient.

20

ROOTING A TREE

For doing any tree computation, we need to know the parent $p(v)$ for each node v.

Hence, we need to root the tree at a vertex r.

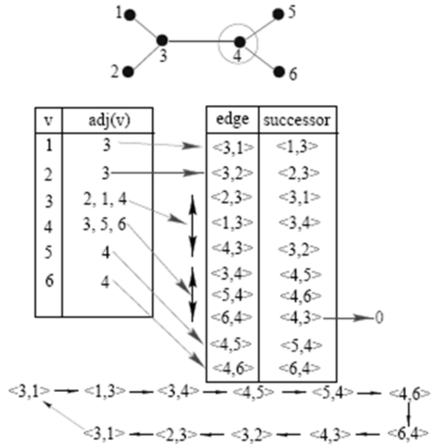
We first construct an Euler tour and for the vertex r, set $s(< u_{d-1}, r >) = 0$.

u_{d-1} is the last vertex adjacent to r.

In other words, we break the Euler tour at r.

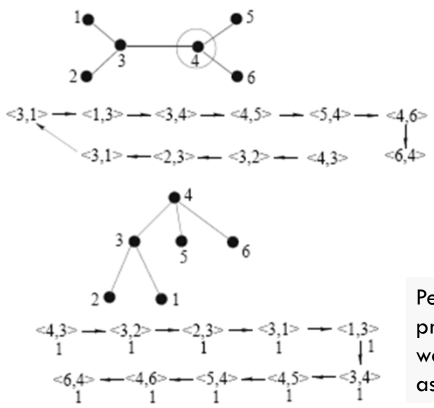
21

ROOTING A TREE



22

ROOTING A TREE



Perform a parallel prefix sum with a weight of one assigned to each arc.

23

ROOTING A TREE

Input: The Euler tour of a tree and a special vertex r.

Output: For each vertex $v \neq r$, the parent $p(v)$ of v in the tree rooted at r.

24

ROOTING A TREE

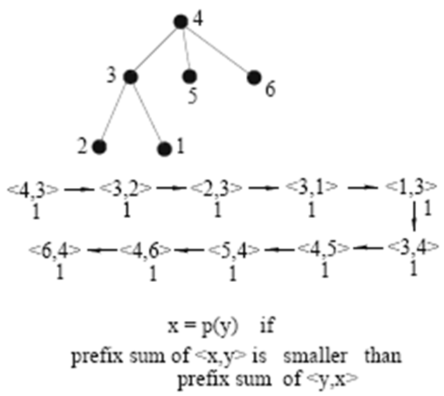
begin

1. Set $s(< u, r >) = 0$, where u is the last vertex in the adjacency list of r.
2. Assign a weight 1 to each edge of the list and compute parallel prefix.
3. For each edge $< x, y >$, set $x = p(y)$ whenever the prefix sum of $< x, y >$ is smaller than the prefix sum of $< y, x >$.

end

25

ROOTING A TREE



x = p(y) if
prefix sum of <x,y> is smaller than
prefix sum of <y,x>

26

POSTORDER NUMBERING

Input: A rooted tree with root r, and the corresponding Euler path defined by the function s.

Output: For each vertex *v*, the postorder number *post*(*v*) of each vertex *v*.

POSTORDER NUMBERING

The Euler path (EP) can be used to solve this.

The EP visits each vertex several times, the first time by the arc <p(v),v>, and the last time by the arc <v,p(v)>, after visiting all the descendants of v.

Thus ordered sublist of all vertices obtained by retention of the last occurrence of each vertex defines precisely the postorder traversal of the vertices of T.

How can we do that?

POSTORDER NUMBERING

begin

1. For each vertex v≠r, assign the weights w(<v,p(v)>)=1, and w(<p(v),v>)=0.
2. Perform parallel prefix sum on the list of arcs defined by s.
3. For each vertex v≠r, set post(v) equal to the prefix sum of <v,p(v)>. For v=r, set post(r)=n, where n is the number of vertices in the given tree.

end

COMPUTATION OF TREE FUNCTIONS

Given a tree *T*, for many tree computations:

- We first construct the Euler tour of *T*
- Then we root the tree at a vertex

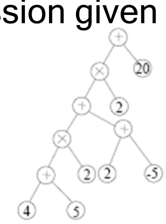
We can compute:

- The postorder number of each vertex
- The preorder number of each vertex
- The inorder number of each vertex
- The level of each vertex
- The number of descendants of each vertex.

TREE CONTRACTION

Some tree computations cannot be solved efficiently with the Euler tour technique alone.

An important problem is evaluation of an arithmetic expression given as a binary tree.



((4 + 5) * 2 + (-5 + 2)) * 2 + 20

31

TREE CONTRACTION

Each leaf holds a constant and each internal node holds an arithmetic operator like +,∗.

The goal is to compute the value of the expression at the root.

The tree contraction technique is a systematic way of shrinking a tree into a single vertex.

We successively apply the operation of merging a leaf with its parent or merging a degree-2 vertex with its parent.

32

THE RAKE OPERATION

Let $T = (V, E)$ be a rooted binary tree and for each vertex v, $p(v)$ is its parent.

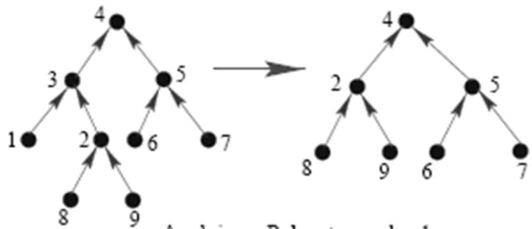
$sib(v)$ is the child of $p(v)$. We consider only binary trees.

In the rake operation for a leaf u such that $p(u) \neq r$.

- Remove u and $p(u)$ from T, and
- Connect $sib(u)$ to $p(p(u))$.

33

THE RAKE OPERATION



Applying Rake to node 1

In our tree contraction algorithm, we apply the rake operation repeatedly to reduce the size of the binary tree.

We need to apply rake to many leaves in parallel in order to achieve a fast running time.

34

THE RAKE OPERATION

But we cannot apply rake operation to nodes whose parents are adjacent on the tree.

For example, rake operation cannot be applied to nodes 1 and 8 in parallel.

We need to apply the rake operation to non-consecutive leaves as they appear from left to right.

35

THE RAKE OPERATION

We first label the leaves consecutively from left to right.

In an Euler path for a rooted tree, the leaves appear from left to right.

We can assign a weight 1 to each edge of the kind $(v, p(v))$ where v is a leaf.

We exclude the leftmost and the rightmost leaves. These two leaves will be the two children of the root when the tree is contracted to a three-node tree.

We do a prefix sum on the resulting list and the leaves are numbered from left to right.

36

THE RAKE OPERATION

We now store all the n leaves in an array A *(except the left most and right most leaves)*.

A_{odd} is the subarray consisting of the odd-indexed elements of A.

A_{even} is the subarray consisting of the even-indexed elements of A.

We can create the arrays A_{odd} and A_{even} in $O(1)$ time and $O(n)$ work.

37

TREE CONTRACTION ALGORITHM

begin

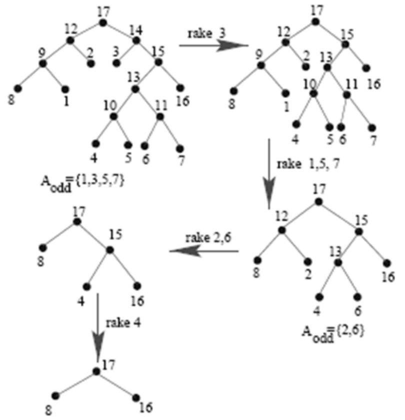
for $\lceil \log(n+1) \rceil$ iterations do

1. Apply the rake operation in parallel to all the elements of A_{odd} that are left children
2. Apply the rake operation in parallel to the rest of the elements in A_{odd}.
3. Set $A := A_{even}$.

end

38

TREE CONTRACTION ALGORITHM



39

CORRECTNESS OF TREE CONTRACTION

Whenever the rake operation is applied in parallel to several leaves, the parents of any two such leaves are not adjacent.

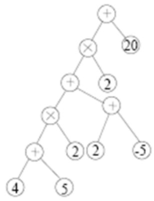
The number of leaves reduces by half after each iteration of the loop. Hence the tree is contracted in O(log *n*) time.

Euler tour takes O(*n*) work.

The total number of operations for all the iterations of the loop is: $O(\sum_i (\frac{n}{2^i})) = O(n)$

40

EVALUATION OF ARITHMETIC EXPRESSIONS



((4 + 5) * 2 + (-5 + 2)) * 2 + 20

If we evaluate an expression tree bottom-up, it will take O(n) time for a long and skinny tree.

Hence we apply tree contraction.

41

EVALUATION OF ARITHMETIC EXPRESSIONS

We do not completely evaluate each internal node. We evaluate the internal nodes partially.

For each internal node v, we associate a label (a_v, b_v). a_v and b_v are constants.

The value of the expression at node is:

(a_v X + b_v), where X is an unknown value for the expression of the subtree rooted at v, *and* + is the operator stored in node v.

42

KEEPING AN INVARIANT

Invariant:

Let u be an internal node which holds the operation $\oplus \in \{+, \times\}$.

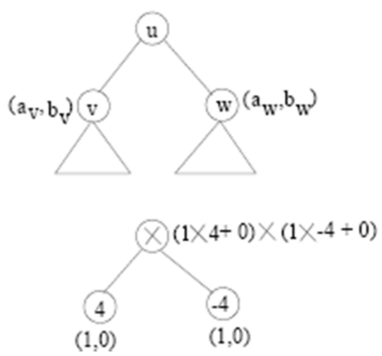
Let v and w are the children of u with labels (a_v, b_v) and (a_w, b_w).

Then the value at u is:

$val(u) = (a_v val(v) + b_v) \oplus (a_w val(w) + b_w)$

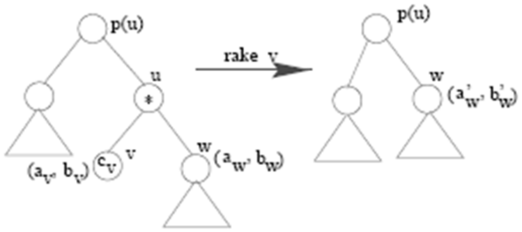
43

KEEPING AN INVARIANT



44

APPLYING THE RAKE OPERATION



The value at node u is:

$val(u) = (a_v c_v + b_v) \times (a_w X + b_w)$

X is the unknown value at node w.

45

APPLYING THE RAKE OPERATION

The contribution of $val(u)$ to the value of node $p(u)$ is:

$a_u \times val(u) + b_u = a_u[(a_v c_v + b_v) \times (a_w X + b_w)] + b_u$

We can adjust the labels of node w to (a'_w, b'_w)

$a'_w = a_u(a_v c_v + b_v)\, a_w$

$b'_w = a_u(a_v c_v + b_v)\, b_w + b_u$

46

COMPLEXITY OF EXPRESSION EVALUATION

The correctness of the expression evaluation depends on correctly maintaining the invariants.

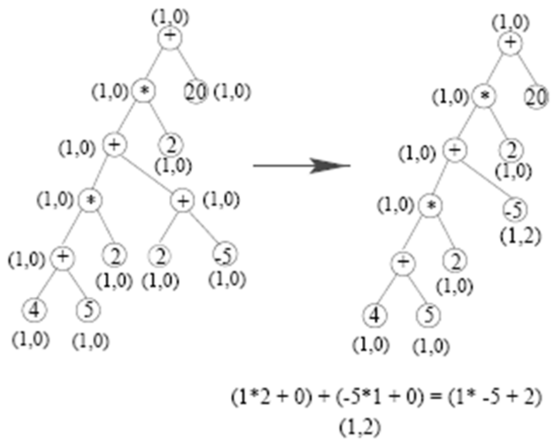
We start with a label (1, 0) for each leaf and correctly maintain the invariant at each rake operation.

We have already proved the correctness of the rake operation.

Hence, evaluation of an expression given as a binary tree takes O(n) work and O(log n) time.

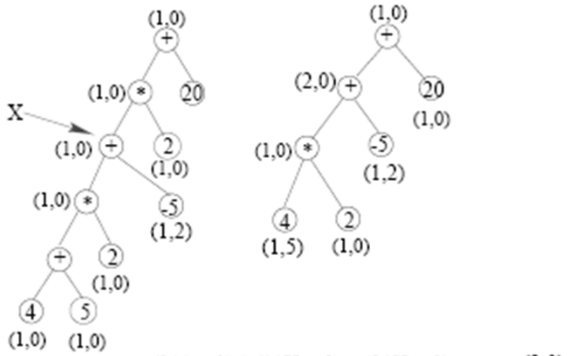
47

AN EXAMPLE



48

AN EXAMPLE

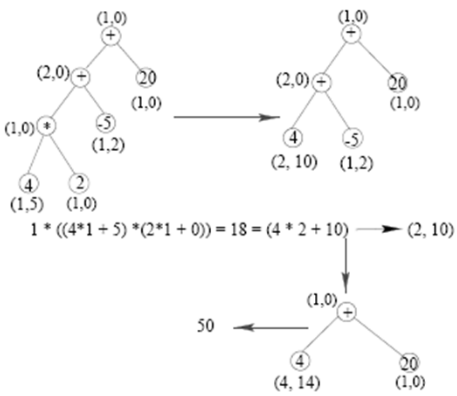


(2*1 + 0) * (1*X + 0) = (2*X + 0) ⟶ (2,0)

(4*1+0) + (5*1 +0) = (4*1 + 5) ⟶ (1,5)

49

AN EXAMPLE



1 * ((4*1 + 5) *(2*1 + 0)) = 18 = (4 * 2 + 10) ⟶ (2, 10)

50

2*((4*2 + 10) + (-5 * 1 + 2)) = 30 = (4 * 4 + 14) ⟶ (4, 14)

50