PARALLEL AND DISTRIBUTED ALGORITHMS

BY
DEBDEEP MUKHOPADHYAY
AND
ABHISHEK SOMANI

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

PRAM ALGORITHMS: LIST RANKING AND COLORING

THE LIST RANKING PROBLEM

Given a linked list L of n nodes whose order is specified by an array S or Succ) such that S(i) contains a pointer to the node following i on L, for $1 \le i \le n$

We assume S(i)=0 when i is the end of the list.

The List Ranking problem is to determine the distance of each node i from the end of the list.

The List ranking problem is one of the most elementary problems in list processing whose sequential complexity is trivially linear.

The pointer jumping (PJ) technique can be used to derive a parallel algorithm for the list ranking problem.

The corresponding running time is O(log n), and the corresponding total number of operations is O(n log n) => Non-optimal solution.

OPTIMAL LIST RANKING

PJ can be made optimal if we can somehow reduce the size of the list to O(n/log n) nodes using a linear number of operations.

The standard approach to achieve optimality would be:

1. Partition the input list into approximately n/log n blocks, each containing O(log n) nodes.

2. Rank each node within each block by using an optimal sequential algorithm, called the preliminary rank.

3. Combine the preliminary ranks using an O(log n) time parallel algorithm.

Unfortunately each block can have O(log n) sublists due to PJ, in which case the size of the input list to the O(log n) time parallel algorithm would not have been reduced to O(n/log n) nodes.

ALTERNATIVE STRATEGY: SYMMETRY BREAKING AND DETERMINISTIC COIN TOSSING (COLE, VISHKIN'86)

Step 1: Shrink the linked list L to L' until only O(n/log n) nodes remain.

Step 2: Apply the pointer jumping technique on the short list L'.

- Requires O(lg n) time, with cost O(n)

Step 3: Restore the original list and rank all the nodes removed in step 1.

Step 1 is the main difficult step, which needs to be performed in O(log n) time with a cost of O(n)

INDEPENDENT SETS

The method for shrinking L consists of removing a selected set of nodes from L and updating the intermediate R values of the remaining nodes. (R(i) is the rank or distance of node i from the end of the list).

The key to a fast parallel algorithm lies in using an Independent Set of nodes which can be deleted in parallel.

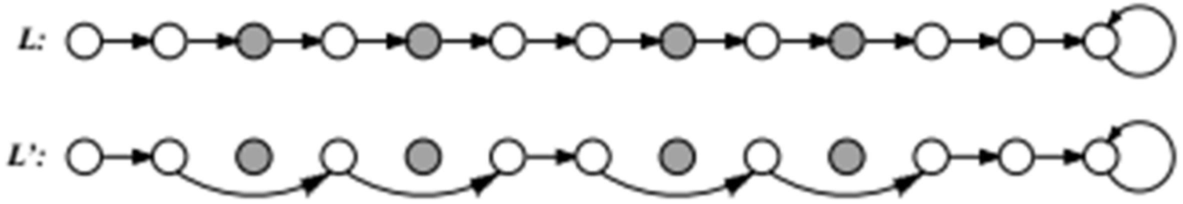
A set I of nodes is independent if, whenever i∈I, $\text{if } s(\mathrm{i}) \neq i, s(i) \notin I.$

We can remove each node i∈I, by adjusting the successor pointer of the predecessor of i. Since, I independent this process can be applied concurrently to all the nodes in I.

DELETION OF THE INDEPENDENT NODES

$I \subset L\ is\ an\ independent\ set(IS) \Rightarrow \forall i \in I\ can\ be\ deleted$
$from\ L\ in\ parallel$.

Proof: If $I \subset L$ is an IS, then $\forall i \in I, P[i] \in I$.



IDENTIFYING THE INDEPENDENT SET

We can handle the problem of finding an independent set by coloring the nodes of the list L.

Recall that a k-coloring of L is a mapping from the set of nodes in L into {0,1,…,k-1} such that no adjacent vertices are assigned the same color.

A node u is a **local minimum (or maximum)** wrt. this coloring if the color of u is smaller (larger) than the colors of its predecessor and successor.

A RESULT

Let k≥2 be a constant and consider any k-coloring c of elements x of L, ie. $\forall x \in L, 1 \leq c(x) \leq k$ and $c(Pred[x] \neq c(x) \neq c(Succ[x])$. Then the set of local minima of coloring c is an IS of size Ω(n/k) and there is a work-optimal parallel algorithm to determine the local minima.

Proof: Let u and v be 2 local minima of c such that no other local minima exists between them.

Then u and v cannot be adjacent.

Colors of elements between u and v must form a bitonic sequence of at most (k-2)+1+(k-2)=2k-3 colors.

Thus, $|I| \geq \frac{n}{2k-2} = \Omega(\frac{n}{k})$.

Given a coloring, determining its local minima is trivial on EREW PRAM just by inspecting predecessor's color and successor's color for all elements in parallel.

REDUCING TO 3-COLORING

A large IS can be obtained by 3-coloring the list.

For a 3-coloring $|I| \geq \left\lfloor \frac{n}{4} \right\rfloor$.

In order to reduce n-element list L to L' with n/(log n) elements, we must remove ISs based on local minima of 3-coloring repeatedly.

BOUND ON NUMBER OF ITERATIONS

$\mathrm{O}(\log logn)$ iterations consisting in removing ISs of local minima of 3-colorings are needed to reduce L to L' with $|L'| \leq n/\log n$.

Proof: Let m be the number of iterations required to reduce L to L'.

Let L_k be the length of L after k iterations and I_k be the IS of local minima of a 3-coloring of L_k.

Then $|I_k| \geq |L_k|/4$, and $|L_{k+1}| = |L_k| - |I_k| \leq (3/4)|L_k|$.

By recursive definition for $|L_k|$ and using $|L| = |L_0| = n$, we have $|L_k| \leq (3/4)^k n$.

Since, $|L_m| \leq n/\log n$, m must fulfil condition $(3/4)^m n \leq n/\log n$, which is equivalent to $m \geq \log_{4/3} logn = \frac{loglogn}{\log\left(\frac{4}{3}\right)} = 2.4loglogn$.

SOLVING THE 3-COLORING PROBLEM

The problem has reduced to the problem of 3-coloring of a linked list.

- Sequentially this is trivial. We just need to traverse the list, assigning alternate colors 0 and 1 (add a color 2 in case of a cycle)
- To do it in parallel we need to break the symmetry of the nodes assigned to every processor

Due to the fact that the indices are random (ie. Succ does not have any locality), nodes in a sublist of size log n assigned to each processor looks alike.

We need to partition them into classes such that all nodes can be assigned the same color in parallel.

We describe an elegant deterministic method called Deterministic Coin Tossing (DCT) to break the symmetry.

- Based on the idea that the only nonsymmetry among elements of the list is their unique identification numbering.
- The identifications are used as an initial n-coloring and it is then transformed into a 3-coloring.

A BASIC PARALLEL COLORING SCHEME USING DCT FOR A DIRECTED CYCLE

Assume the arcs of G are specified by an array S st:

- If (i,j)ϵE, we have s(i)=j, for 1≤i,j≤n
- We start with an initial coloring of c(i)=I for all i.
- The binary expansion of the color c is $c_{t-1}…c_k…c_1c_0$
- The kth LSB is c_k.

Parallel Reduction of the number of initial colors:

For 1≤i≤n, in parallel we:

1. Set k to the LSB in which c(i) and c(S(i)) differ.

2. Set $c'(i)=2k+c(i)_k$

Note if initial coloring is a t-bit value, max value of c'=2(t-1)+1=2t-1, which can be represented by a $\lceil\log(t)\rceil+1$. Thus there is an exponential reduction in the number of colors!

Is the coloring correct?

CORRECTNESS

As the starting coloring is correct such a differing k must exist.

Suppose by contradiction the derived coloring is incorrect.

Thus for an edge $(i,j)\in E$, $c'(i)=c'(j)$.

- Thus, $2k+c(i)_k=2l+c(j)_l$.
- Note this can be only possible if $k=l$, but then $c(i)_k=c(j)_k$, which defies the definition of k.
- Hence, $c'(i)\neq c'(j)$, for any $(i,j)\in E$.

Assuming that the LSB in which two binary numbers differ can be found on O(1) time, when the binary values are of size O(logn) bits, the algorithm is a constant time algorithm.

How do you convert this to a 3-coloring algorithm?

RECURSIVE APPLICATION OF THE ALGORITHM

The algorithm can be recursively applied reducing the number of colors till $t>3$.

- Note for $t=3$ bits, the max color value is $2.3-1=5$, which also requires 3 bits.
- Thus the number of colors is $0 \leq c'i) \leq 5$.

Iterations of DCT can reduce the number of colors of a coloring only to 6.

We next estimate the number of iterations required to reach this stage.

- Let $\log^{(i)}(x)=\log(\log^{(i-1)}(x))$, $\log^{(1)}(x)=\log(x)$.
- Let $\log^{*}x=\min\{i \mid \log^{(i)}(x) \leq 1\}$
- The function $\log^{*}x$ is an extremely slowly growing function that is bounded by 5 for all $x \leq 2^{65536}$.

Starting with the initial coloring $c(i)=i$, for $1 \leq i \leq n$, then each iteration reduces the number of colors: after 1st iteration $O(\log n)$, after 2nd $O(\log^2(n))$.

Thus number of colors will be reduced to 6 after $O(\log^{*}n)$ iterations.

THE FINAL CUT!

We apply a further recolor.

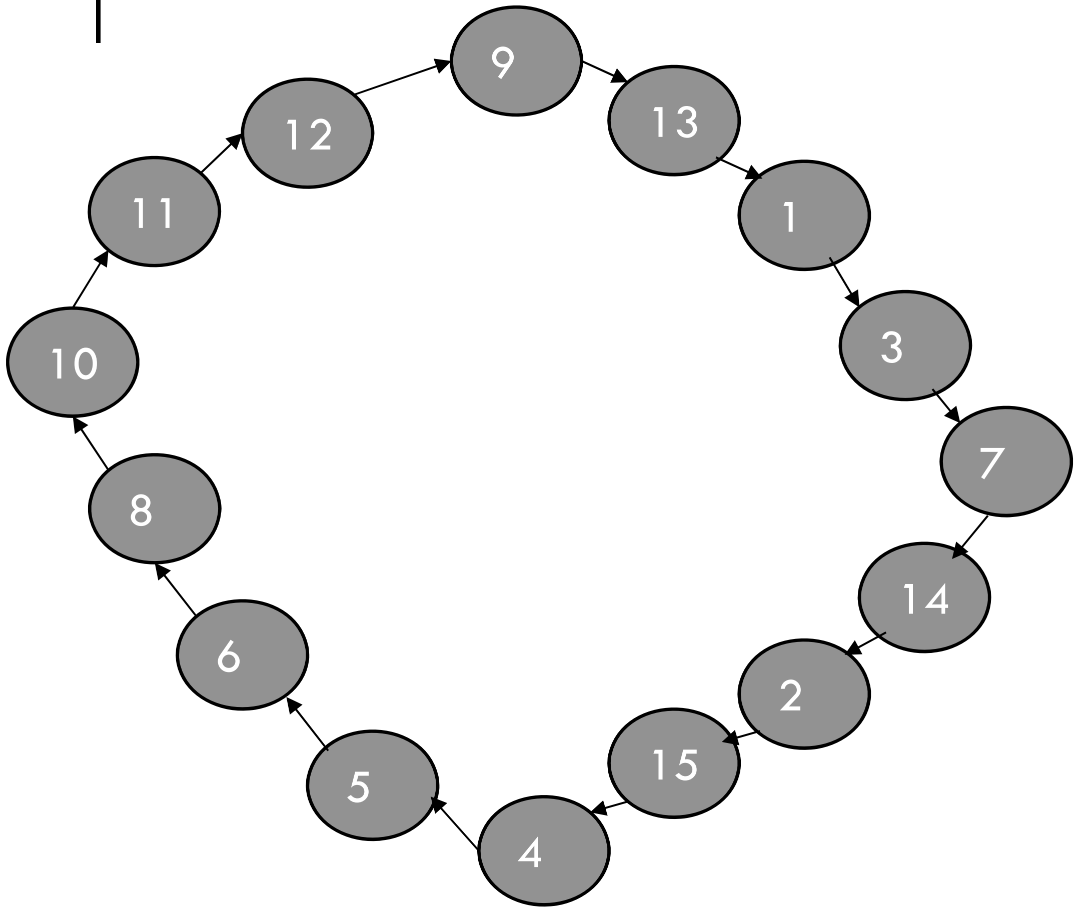
The additional recoloring procedure consists of 3 iterations, each of which handles vertices of a specific color.

For each color which lies between 3 and 5, ie. $3 \leq l \leq 5$, we recolor all vertices i with color l with the smallest possible color from {0,1,2} (ie. Smallest color different from predecessor and successor).

Each iteration takes O(1) time with n processors.

- Note when two nodes with color 3 is handled, they are never adjacent.
- Thus the correctness is ensured.

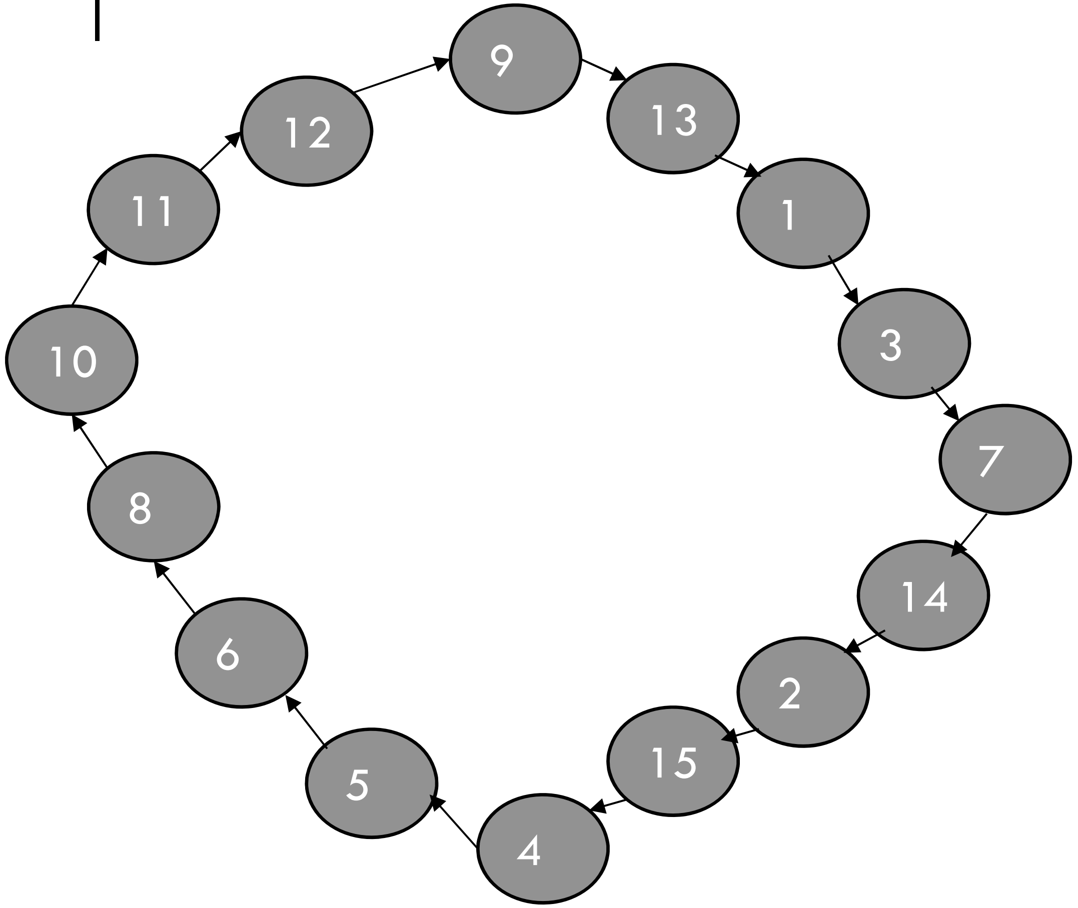
EXAMPLE



Note now there are 6 colors: 0-5

v	c	k	c'
1	0001	1	2
3	0011	2	4
7	0111	0	1
14	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	3
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	4
13	1101	2	5

EXAMPLE



Note now there are 3 colors: 0-2

v	c	k	c'
1	0001	1	2
3	0011	2	0
7	0111	0	1
14	1110	2	2
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	0
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	1
13	1101	2	0

COMPLEXITY

Using DCT, we can construct a 3-coloring on p processors in time T(n,p)=O(nlog*n/p) with C(n,p)=O(nlog*n).

When p=n, T=O(log*n), with C=O(nlog*n).

Optimal Algorithm for 3-coloring:

Apply the 3-coloring once.

For the O(log n) remaining colors we apply the re-coloring scheme.

We can 3-color in time O(log n) time, with a cost of O(n).

LIST RANKING USING COLORING

```
1. Set n0=n, k=0
2. While nk>n/log n do
    2.1 Set k=k+1
    2.2 Color the list with 3 colors, and identify the set I of local minima
    2.3 Remove the nodes in I, and store the appropriate information regarding the removed nodes (discuss later)
    2.4 Let nk be the size of the remaining list. Compact list into consecutive memory locations.
3. Apply PJ to the resulting list.
4. Restore the original list and rank all the removed nodes by reversing the process in Step 2
```

Note step 2 needs to be repeated O(loglogn) times. 2.2 takes O(log n) time using O(n) operations.

We need to discuss Steps 2.3.

REMOVING NODES OF AN INDEPENDENT SET

Input: 1) Arrays S and P of length n representing, respectively, the successor and the predecessor relations of a linked list, 2) an independent set I of nodes, 3) a value R(i) for each node i.

Output: The list obtained after removal of all the nodes in I with the updated R values.

```
Begin
   1. Assign consecutive serial numbers N(i) to the elements of I, where
1≤N(i)≤|I|=n'.
    2. for all i∈I in parallel
            U(N(i))=(i,S(i),R(i))
            R(P(i))=R(P(i))+R(i)
             P(S(i))=P(i)
             S(P(i))=S(i)
```

CORRECTNESS, COMPLEXITY, RESTORATION

Given a linked list L of size n and an independent set I, the previous Algorithm correctly removes the nodes of I and updates the R values in O(log n) time using O(n) operations.

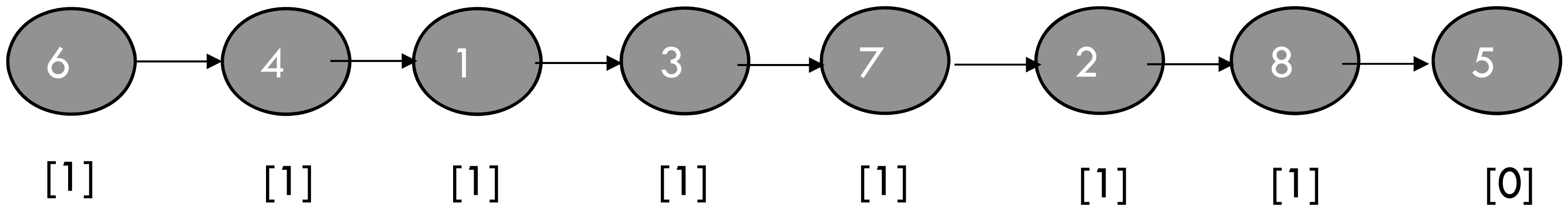
Proof: Correctness follows from the fact that no two nodes of I are adjacent.

As for the running time, step 1 takes O(log n) time using O(n) operations by a pre-fix sum computation on the nodes of L, such that a weight of 1 is assigned to each node in I, and a weight of 0 is assigned to each of the remaining nodes.

Step 2 can be executed in O(1) time, using O(n) operations.

Restoration: Once the ranks of the nodes in the contracted list are determined, it is easy to obtain the ranks of the deleted nodes and to restore the original list using the information stored in the U array.

EXERCISE



THE EULER TOUR TECHNIQUE: EVALUATION OF TREE FUNCTIONS

OVERVIEW

The Euler tour technique

Computation of different tree functions

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.

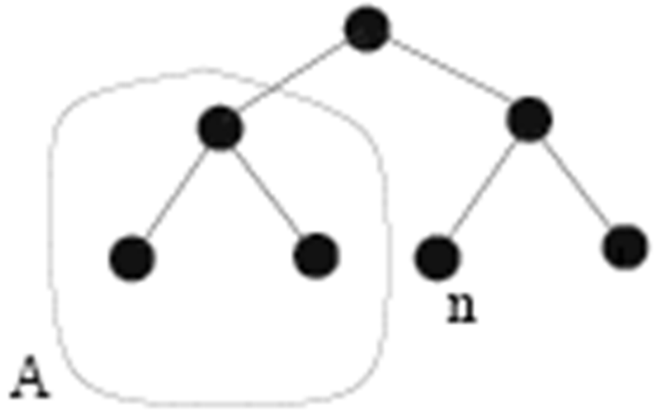
Some of these computations include *preorder*, *postorder*, *inorder* numbering of the nodes of a tree, number of descendants of each vertex, level of each vertex etc.

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.

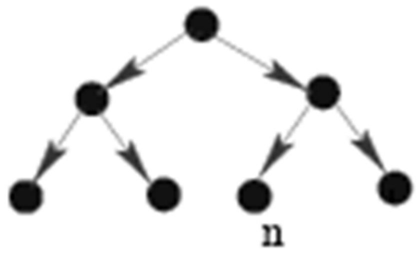
PARALLEL DEPTH-FIRST SEARCH



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*.

PARALLEL DEPTH-FIRST SEARCH

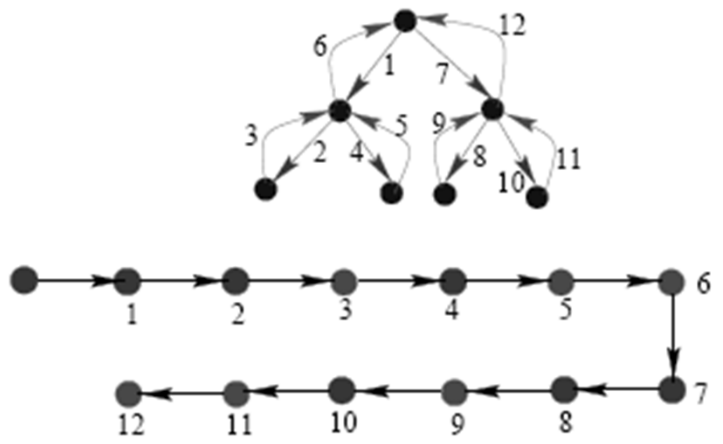


There is a definite order of visiting the nodes in depth-first search.

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.

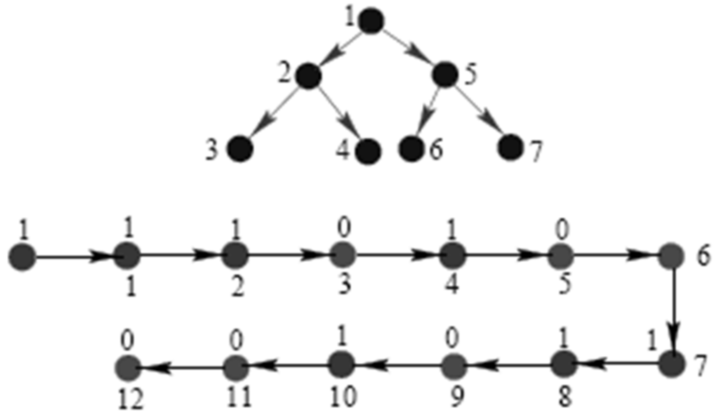
PARALLEL DEPTH-FIRST SEARCH



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 2*n* - 2 nodes, where each arrow (directed edge) is a node of the list.

EULER TOUR TECHNIQUE



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

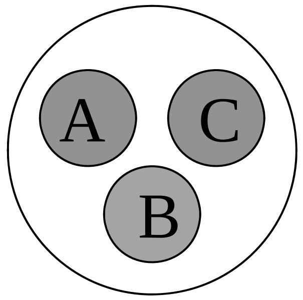
Each red node in the list represents an edge of the nature $< p(v) , v >$.

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

THE EULER-TOUR TECHNIQUE

The problem of computing the depth of each node in an n-node binary tree

BINARY TREE



Let T be a binary tree stored in a PRAM

Each node *i* has fields parent[i], left[i] and right[i], which point to node i's parent, left child and right child respectively

Let's assume that each node is identified by a non-negative integer

Also we associate not one but 3 processes with each node; we call these node's A,B and C processors

Mapping between each node *i* and its 3 processors A,B and C: 3i, 3i+1, 3i+2

COMPUTING DEPTH OF EACH NODE IN AN N NODE TREE TAKES O(N) TIME ON A SERIAL RAM

A simple parallel algorithm to compute depths propagates a “wave” downward from the root of the tree.

- The wave reaches all nodes at the same depth simultaneously, and thus by incrementing a counter carried along with the wave, we can compute the depth of each node.

This parallel algorithm works well on a complete binary tree, since it runs in time proportional to the tree’s height.

But the height of the tree

could be as large as n-1

USING THE EULER-TOUR TECHNIQUE WE CAN COMPUTE NODE DEPTHS IN O(LOG N) TIME ON AN EREW PRAM

An Euler-tour of a graph is a cycle that traverses each edge exactly once, although it may visit a vertex more than ones

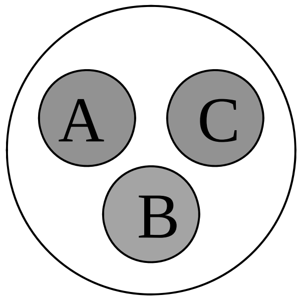
- A connected, directed graph has an Euler tour if and only if for all vertices v, the in-degree of v equals the out degree of v
- Since each undirected edge (u,v) in an undirected graph maps to two directed edges (u,v) and (v,u) in the directed version, the directed version of any connected, undirected graph (and therefore of any undirected tree) has an Euler tour

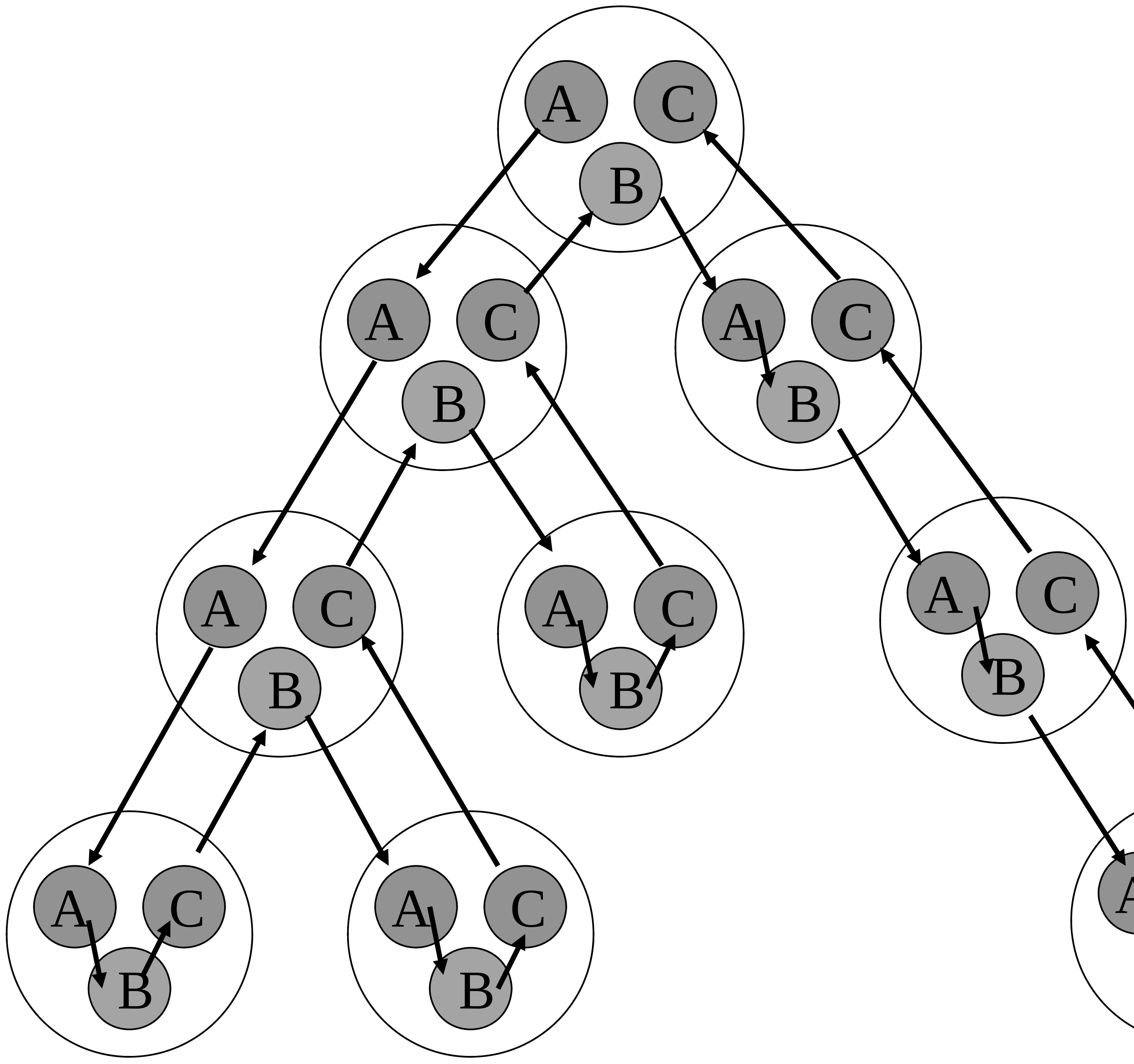
DEPTH OF NODES COMPUTATION

First we form an Euler tour of the directed version of T.

The tour corresponds to walk of the tree with the following structure:

- A node’s A processor points to the A processor of its left child, if it exist, and otherwise to its own B processor
- A node’s B processor points to the A processor of its right child, if it exist, and otherwise to its own C processor
- A node’s C processor points to the B processor of its parent, if it is a left child and to the C processor of its parent if it is a right child. The root’s C processor points to NIL.


A
C
B


A
C
B

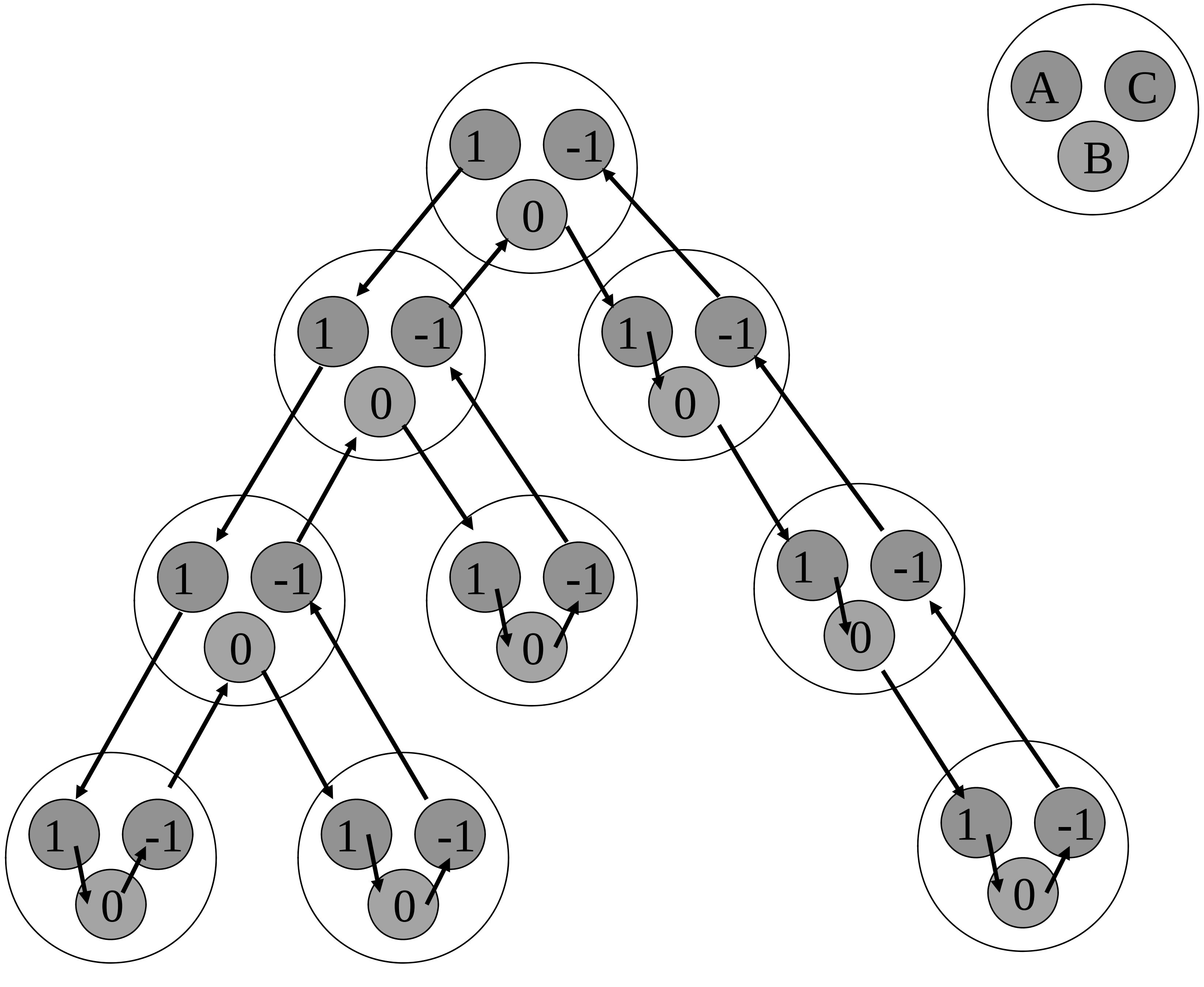
FIRST STEP

Thus, the head of the linked list formed by the Euler tour is the root's A processor, and the tail is the root's C processor.

Given the pointers composing the original tree, an Euler tour can be constructed in O(1) time.

Once we have linked list representing the Euler tour of T, we place

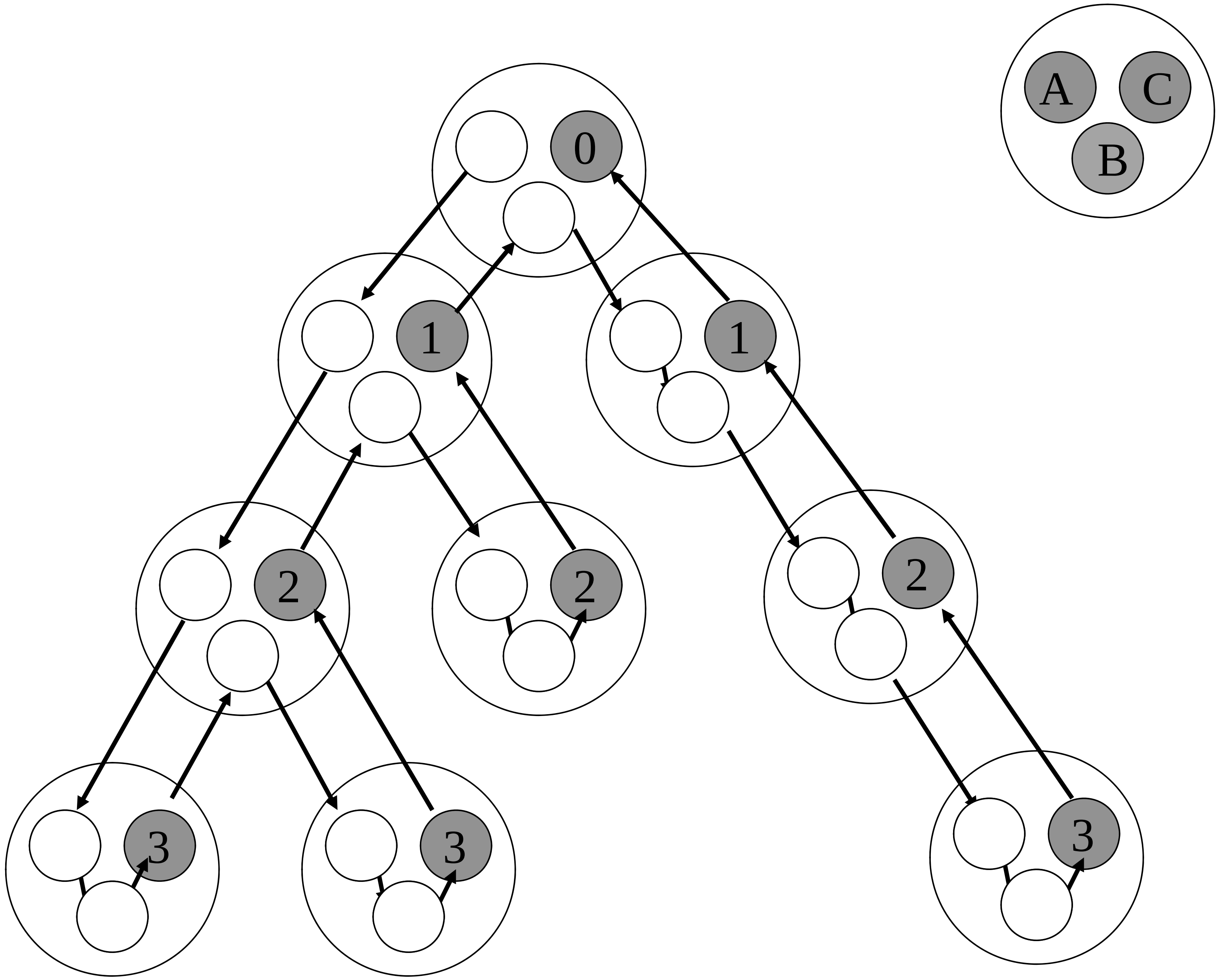
- a 1 in each A processor,
- a 0 in each B processor and
- a –1 in each C processor


A
C
B
1 -1 0
1 -1 0
1 -1 0
1 -1 0
1 -1 0
1 -1 0
1 -1 0
1 -1 0
1 -1 0
1 -1 0

SECOND STEP

We then perform a parallel prefix computation using ordinary addition as the associative operation

We claim that after performing the parallel prefix computation, the depth of each node resides in the node’s C processor. Why?


A
C
B
0
1
1
2
2
2
3
3
3

WHY ???

The numbers are placed into the A,B and C processors in such a way that the net effect of visiting a subtree is to add 0 to the running sum

The A processor of each node *i* contributes 1 to running sum

The B processor of each node *i* contributes 0 because the depth of the node *i's* left child equals the depth of the node *i's* right child

The C processor contributes –1, so the entire visit to the subtree rooted at node *i* has no effect on the running sum.

CONCLUSION

The list representing Euler-tour can be computed in O(1) time.

It has 3n objects, and thus the parallel prefix computation takes only O(log n) time

Thus the total amount of time to compute all node depths is O(log n).

Because no concurrent memory accesses are needed, the algorithm is an EREW algorithm.

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.

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.

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.

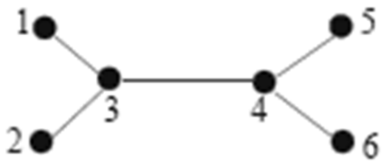
If d is the degree of vertex v, the vertices adjacent to v are:

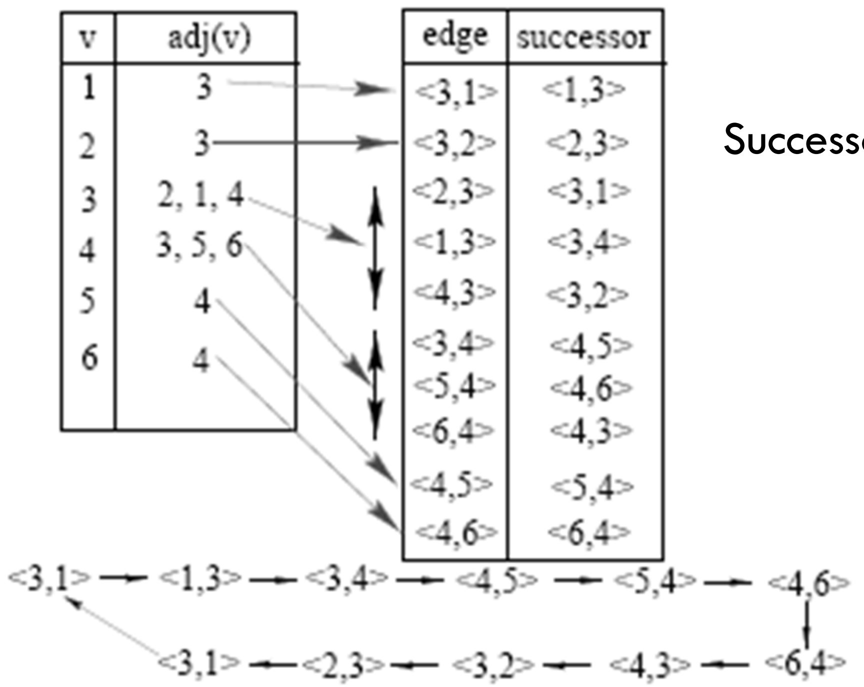
$adj(v) = < u_0, u_1, \ldots, u_{d-1} >$

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

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

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> → <6,4> → <4,3> → <3,2> → <2,3> → <3,1>

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.

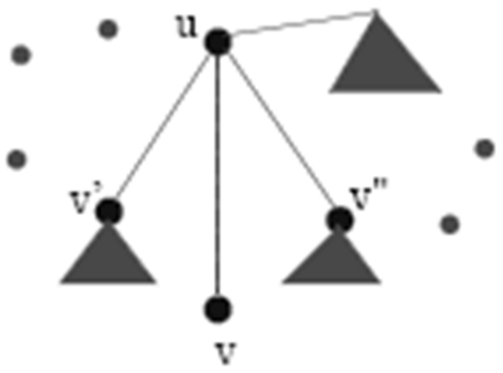
We prove the lemma through induction.

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.

CORRECTNESS OF EULER TOUR

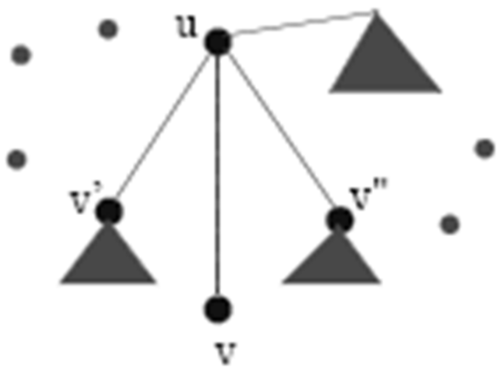


We can introduce an extra node by introducing a leaf to an existing tree, like the leaf *v*.

Initially, *adj*(*u*) = <…, *v*', *v*'', …> . Hence,

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

CORRECTNESS OF EULER TOUR



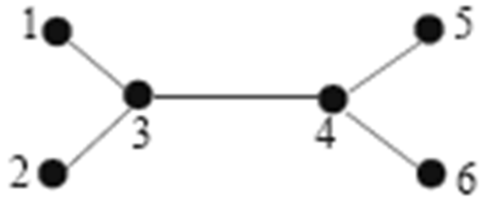
After the introduction of *v*, *adj*(*u*) = <…, *v*', *v*, *v*'', …>

s(< *v*', *u* >) = < *u*, *v* > and

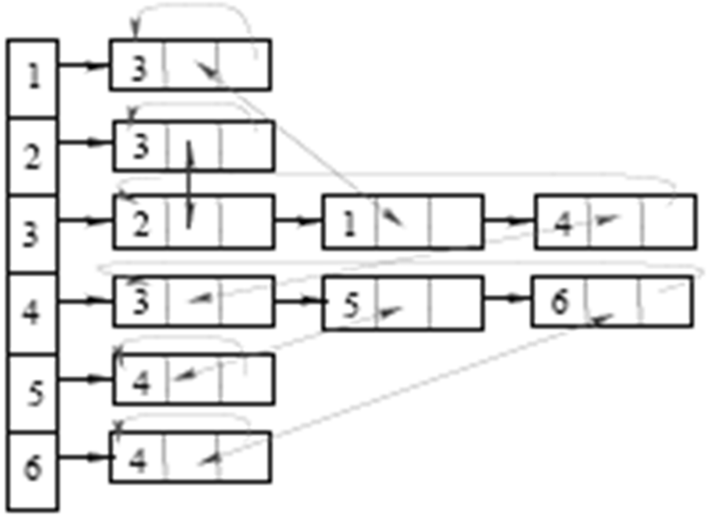
s(< *v*, *u* >) = < *u*, *v*'' >

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

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



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 $L[u_i]$.

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.

ROOTING A TREE

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

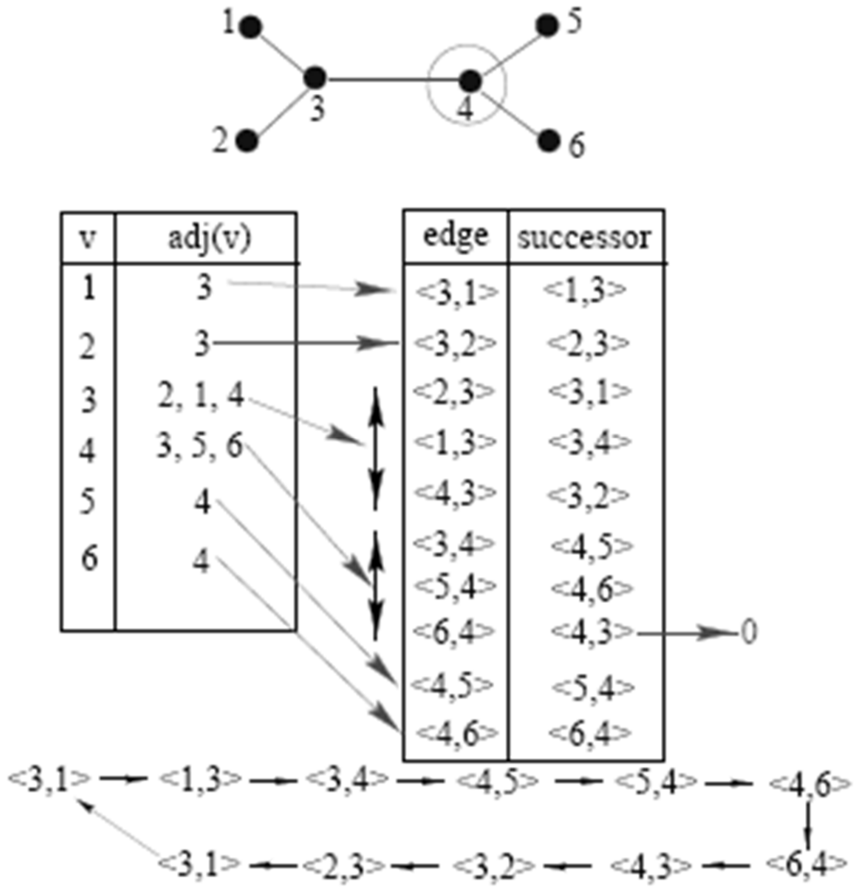
Hence, we need to root the tree at a vertex r.

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.

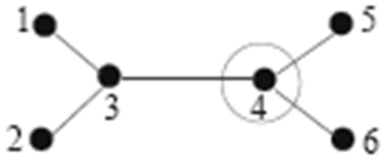
ROOTING A TREE



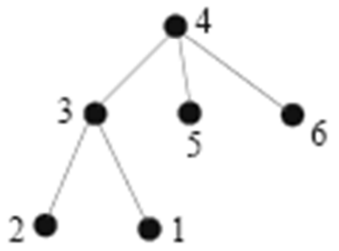
v	adj(v)
1	3
2	3
3	2, 1, 4
4	3, 5, 6
5	4
6	4

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>

ROOTING A TREE



$$\begin{tikzcd}
\langle 3,1\rangle \arrow[r] & \langle 1,3\rangle \arrow[r] & \langle 3,4\rangle \arrow[r] & \langle 4,5\rangle \arrow[r] & \langle 5,4\rangle \arrow[r] & \langle 4,6\rangle \arrow[d] \\
 & \langle 3,1\rangle \arrow[ul] & \langle 2,3\rangle \arrow[l] & \langle 3,2\rangle \arrow[l] & \langle 4,3\rangle \arrow[l] & \langle 6,4\rangle
\end{tikzcd}$$



$$\begin{tikzcd}
\underset{1}{\langle 4,3\rangle} \arrow[r] & \underset{1}{\langle 3,2\rangle} \arrow[r] & \underset{1}{\langle 2,3\rangle} \arrow[r] & \underset{1}{\langle 3,1\rangle} \arrow[r] & \langle 1,3\rangle \arrow[d, "1"] \\
\underset{1}{\langle 6,4\rangle} & \underset{1}{\langle 4,6\rangle} \arrow[l] & \underset{1}{\langle 5,4\rangle} \arrow[l] & \underset{1}{\langle 4,5\rangle} \arrow[l] & \underset{1}{\langle 3,4\rangle} \arrow[l]
\end{tikzcd}$$

ROOTING A TREE

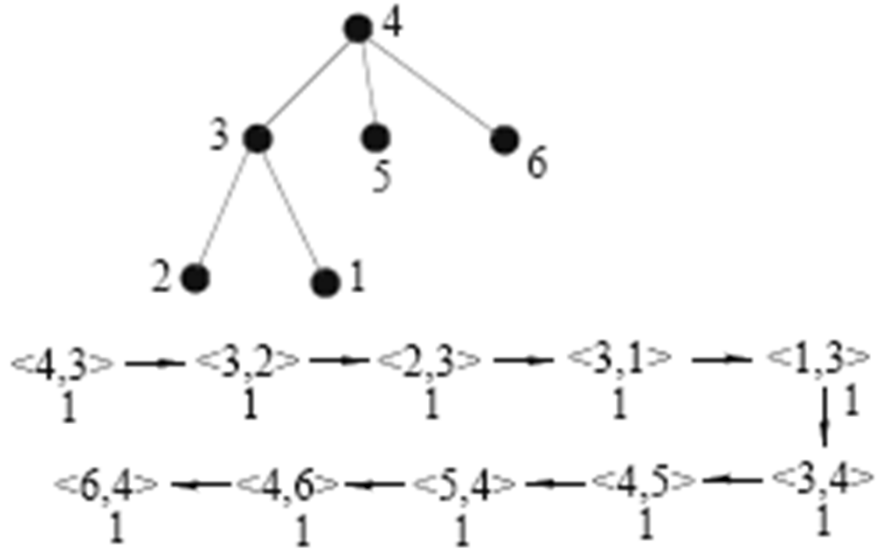
Input: The Euler tour of a tree and a special vertex r.

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

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
```

ROOTING A TREE



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

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

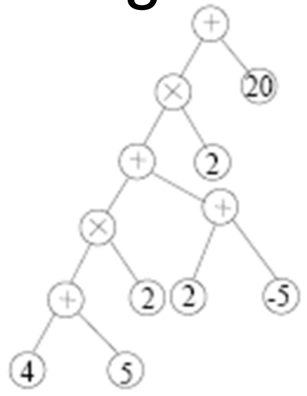
We can compute:

- 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.



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

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.

THE RAKE OPERATION

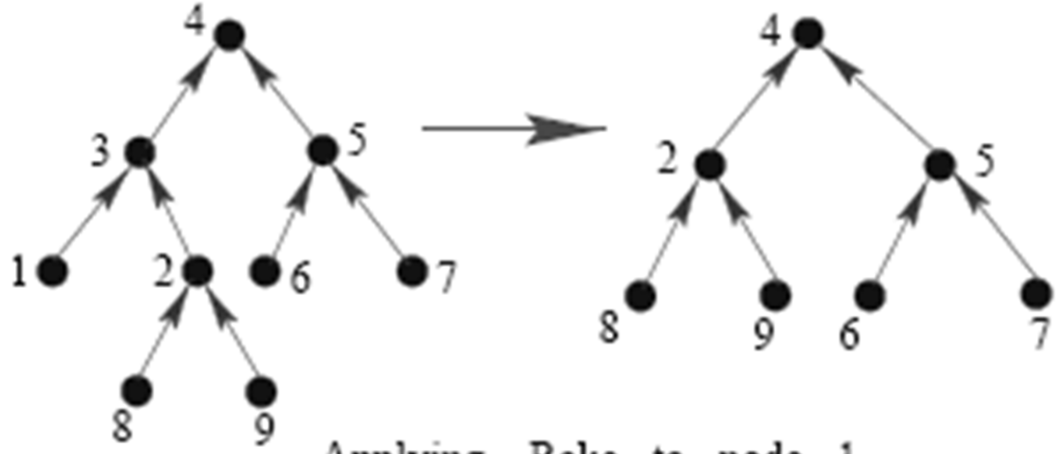
Let $T = (V, E)$ be a rooted binary tree and for each vertex v, $p(v)$ is its parent.

$sib(v)$ 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 $p(p(u))$.

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.

THE RAKE OPERATION

But we cannot apply rake operation to nodes whose parents are consecutive 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.

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.

THE RAKE OPERATION

We now store all the n leaves in an array A.

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

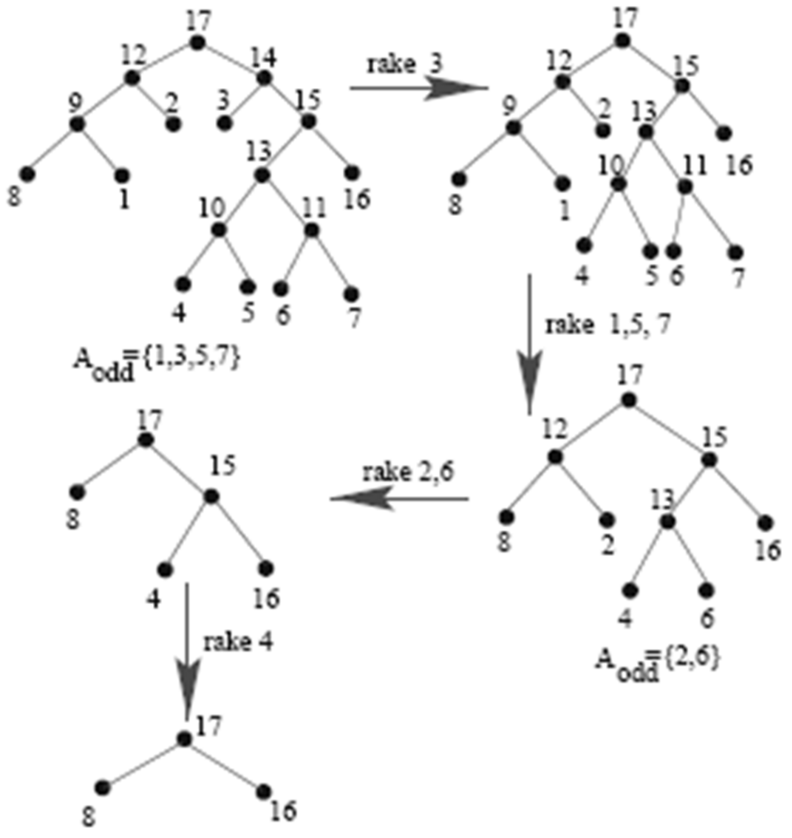
A_{even} is the subarray consisting of the even-indexed elements of A.

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

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
```

TREE CONTRACTION ALGORITHM



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: $O(\sum_i (\frac{n}{2^i})) = O(n)$

TREE COMPUTATIONS

Rooting a tree: