PARALLEL AND DISTRIBUTED ALGORITHMS

BY
DEBDEEP MUKHOPADHYAY
AND
ABHISHEK SOMANI

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

PRAM ALGORITHMS: POINTER JUMPING

LIST RANKING

Consider the problem of finding, for each element of n elements on a linked list, the suffix sums of the last i elements of the list, where

$i \leq i \leq n.$

The suffix sum problem is a variant of the prefix sum problem.

- Array is replaced by a linked list.
- Sums are computed from the end.

If the elements of the list are 0 or 1, and the associative operation is addition, the problem is called the list ranking problem.

LINK RANKING

One way to solve this is to traverse the list and count the number of links traversed between the list element and the end of the list.

Only a single pointer can be followed in one step, and there are n-1 pointers between the first element and the end of the list.

- How can any algorithm traverse such a list in less than $\Theta(n)$ time?

PARALLELISATION

We associate a processor with every list element and jump pointers in parallel!

- The distance to the end of the list is cut in half through the instruction :

$$next[i] \leftarrow next[next[i]]$$

Hence, a logarithmic number of pointer jumpings are sufficient to collapse the list so that every element points to the last list element.

If a processor adds to its own link traversal count, position[i], the current link traversal count of the successors it encounters, the list position will be correctly determined.

ILLUSTRATING THE PROCESS OF LIST RANKING

List ranking problem

- Given a singly linked list L with n objects, for each node, compute the distance to the end of the list

If d denotes the distance

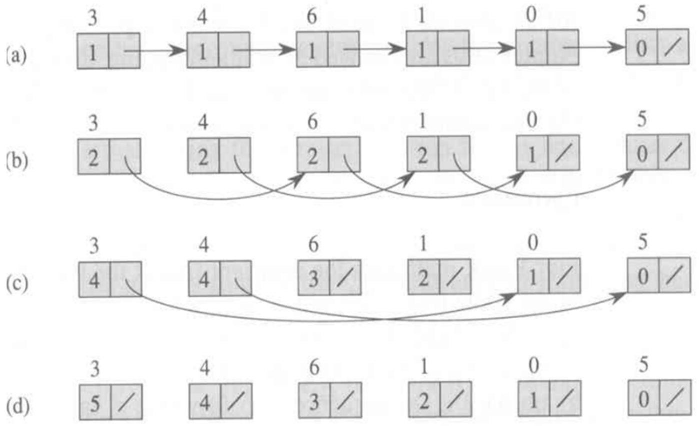
$$\text{node.d} = \begin{cases} 0 & \text{if node.next = nil} \\ \text{node.next.d} + 1 & \text{otherwise} \end{cases}$$

Serial algorithm: O(n)

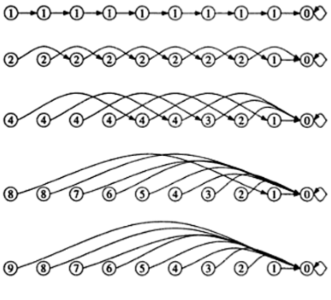
Parallel algorithm

- Assign one processor for each node
- Assume there are as many processors as list objects
- For each node i, perform
  1. i.d = i.d + i.next.d
  2. i.next = i.next.next // pointer jumping

LIST RANKING – EXAMPLE 1

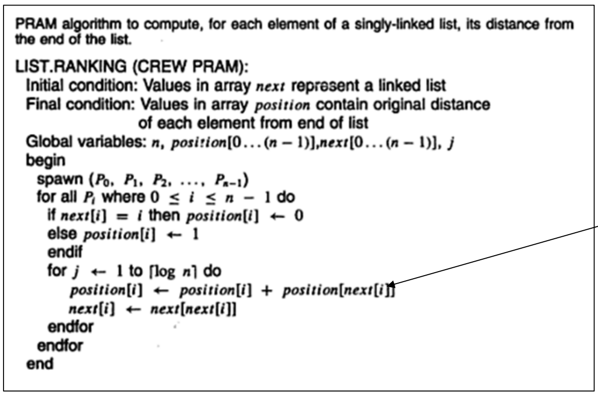


LIST RANKING – EXAMPLE 2



The position of each item on the n-element list can be determined in $\lceil \log n \rceil$ pointer jumping steps.

THE PRAM ALGORITHM



PRAM algorithm to compute, for each element of a singly-linked list, its distance from the end of the list.

```
LIST.RANKING (CREW PRAM):
 Initial condition: Values in array next represent a linked list
 Final condition: Values in array position contain original distance
                  of each element from end of list
 Global variables: n, position[0...(n − 1)], next[0...(n − 1)], j
 begin
   spawn (P_0, P_1, P_2, ..., P_{n−1})
   for all P_i where 0 ≤ i ≤ n − 1 do
     if next[i] = i then position[i] ← 0
     else position[i] ← 1
     endif
     for j ← 1 to ⌈log n⌉ do
        position[i] ← position[i] + position[next[i]]
        next[i] ← next[next[i]]
     endfor
   endfor
 end
```

Note this step does not depend on j.
There are $\lceil \log n \rceil$ steps.
There are n processors.
So total cost is:
$\Theta(n \log n)$
Not cost optimal!

THE SAME CODE USING POINTER NOTATIONS

```
List_ranking(L)
for all P_i for each node i, do
    if i->next = null then i.d = 0
    else i.d = 1
    while(i->next != null) do
        i.d = i.d + i->next.d
        i->next = i->next->next
```

LIST RANKING - DISCUSSIONS

Synchronization is important
- In step 6 (i->next = i->next->next), all processors must read right hand side before any processor write left hand side

The list ranking algorithm is EREW
- If we assume in step 5 (i.d = i.d + i.next.d) all processors read i.d and then read i.next.d
- If j.next = i, i and j do not read i.d concurrently

Work performance
- performs O(n log n) work since n processors in O(log n) time

Work efficient
- A PRAM algorithm is work efficient w.r.t another algorithm if two algorithms are within a constant factor
- Is the link ranking algorithm work-efficient w.r.t the serial algorithm?
  - No, because O(n log n) versus O(n)

Speedup
- S = n / log n

PREORDER TREE TRAVERSAL

Sometimes it is appropriate to reduce a complicated looking problem into a simpler form for which a parallel algorithm is already known.

Let us consider **the problem of numbering the vertices of a rooted tree in preorder (depth first search order).**

At first glance this problem looks sequential!

RECURSIVE PREORDER TRAVERSAL

```
PREORDER.TRAVERSAL(nodeptr):
Begin
    if nodeptr≠null then
        nodecount ← nodecount + 1
        nodeptr.label ← nodecount
        PREORDER.TRAVERSAL(nodeptr.left)
        PREORDER.TRAVERSAL(nodeptr.right)
    endif
End
```

Where is the parallelism?
The fundamental operation assigns a label to a node.

We cannot assign labels to the vertices in the right subtree of the left subtree, until we know how many vertices are on the left subtree of the left subtree, and so on.

The algorithm seems inherently sequential!

Can we parallelize this?

IDENTIFY THE CHARACTER

IDENTIFY THE CHARACTER

IDENTIFY THE CHARACTER

Robert Endre Tarjan (born April 30, 1948) is an American computer scientist and mathematician. He is the discoverer of several graph algorithms, including Tarjan's off-line least common ancestors algorithm, and co-inventor of both splay trees and Fibonacci heaps.

Tarjan is currently the James S. McDonnell Distinguished University Professor of Computer Science at Princeton University, and the Chief Scientist at Intertrust Technologies (Source: Wiki)

PARALLELIZATION OF THE TRAVERSAL

Instead of focusing on the vertices, let us look into the edges.

When we perform a preorder traversal, we systematically work our way through the edges of the tree.

- We pass along every vertex twice: one heading down from the parent to the child, and one going from the child to the parent.

- ***If we divide each tree edge into two edges, one corresponding to the downward traversal, and one corresponding to the upward traversal, then the problem of traversing a tree turns into the problem of traversing a single linked list.***

TARJAN AND VISHKIN (1984)

4 steps:

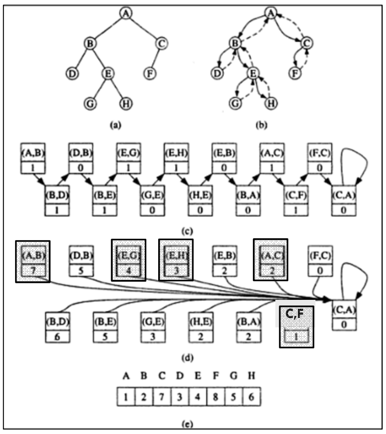
1. The algorithm constructs a singly linked list. Each vertex of the linked list corresponds to a downward or upward edge traversal.

2. Algorithm assigns weights to the vertices of the newly created single linked list.
   - For vertices corresponding to downward edges, the weight is 1 (it contributes to node count).
   - For vertices corresponding to upward edges, the weight is 0 (it does not contribute to node count).

3. For each element of the singly-linked list, the rank of each element is determined (by pointer jumping).

4. The processors associated with the downward edges use the ranks they have computed to assign a preorder traversal number to their associated tree nodes (the tree node at the end of the downward edge).

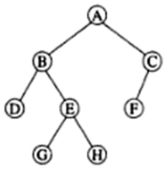
EXAMPLE

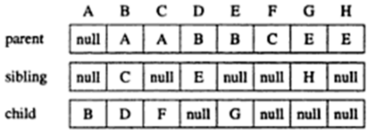


a) Tree
b) Double Tree Edges, distinguishing downward edges from upward edges.
c) Build linked list out of directed tree edges. Associate 1 with downward edges, and 0 with upward edges.
d) Use pointer jumping to compute total weight from each vertex to end of list.

The elements of the linked list which correspond to downward edges, have been shaded.
Processors managing these elements assign preorder values.
For example, (E,G) has a weight 4, meaning tree node G is 4th node from end of preorder traversal list.
The tree has 8 nodes, so it can compute that tree node G has label 5 in preorder traversal (=8-4+1)

DATA STRUCTURE FOR THE TREE





	A	B	C	D	E	F	G	H
parent	null	A	A	B	B	C	E	E
sibling	null	C	null	E	null	null	H	null
child	B	D	F	null	G	null	null	null

For every tree node, the data structure stores the node's parent, the node's immediate sibling to the right, and the node's leftmost child.

Representing the node this way keeps the amount of data stored a constant for each tree node and simplifies the tree traversal.

PROCESSOR ALLOCATION

The PRAM algorithm spawns 2(n-1) processors.

A tree with nodes have (n-1) edges.

We are dividing each edge into two edges, one for the downward traversal and one for the upward traversal.

So, the algorithm needs 2(n-1) processors to manipulate each of the 2(n-1) edges of the singly-linked list of elements corresponding to the edge traversals.

CONSTRUCTION OF THE LINKED LIST

Once all the processors have been activated they construct the linked list:

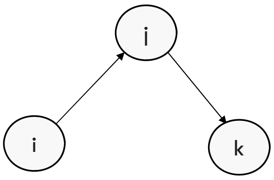
- P(i,j): The processor for the edge (i,j)
- Note (j,i) has a different processor P(j,i)

Given an edge (i,j), P(i,j) must compute the successor of (i,j) and store in a global array: succ[1…2(n-1)].

- If the successor of (i,j) is (j,k), then succ[(i,j)]←(j,k)

HANDLING UPWARD EDGES

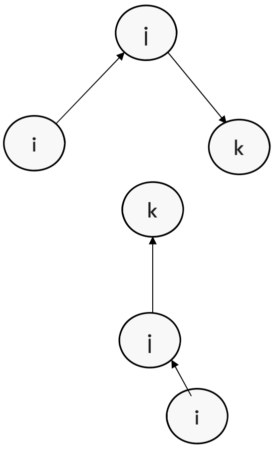
Edge (i,j), such that parent(i)=j



```
If sibling[i]≠NULL
   succ[(i,j)]←(j,sibling[i])
```

HANDLING UPWARD EDGES

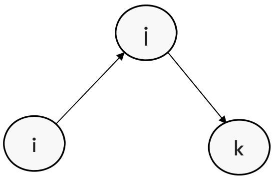
Edge (i,j), such that parent(i)=j



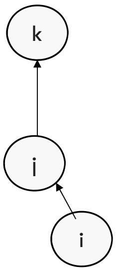
```
If sibling[i]≠NULL
   succ[(i,j)]←(j,sibling[i])

Else If parent[i]≠NULL
   succ[(i,j)]←(j,parent[i])
```

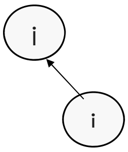
HANDLING UPWARD EDGES

Edge (i,j), such that parent(i)=j



```
If sibling[i]≠NULL
    succ[(i,j)]←(j,sibling[i])

Else If parent[i]≠NULL
    succ[(i,j)]←(j,parent[i])

Else
    succ[(i,j)]←(i,j)
```

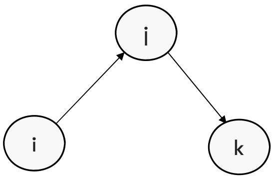




The edge is at the end of the tree traversal, so we put a loop at the end of the element list.

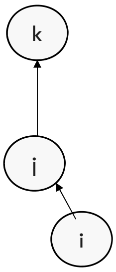
HANDLING UPWARD EDGES

Edge (i,j), such that parent(i)=j

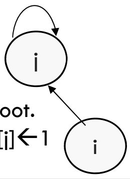


```
If sibling[i]≠NULL
    succ[(i,j)]←(j,sibling[i])

Else If parent[i]≠NULL
    succ[(i,j)]←(j,parent[i])

Else
    succ[(i,j)]←(i,j)
```

position[1…2(n-1)] is a global array to hold the edge ranks.



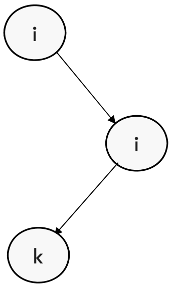
j is the root.
position[j]←1



The edge is at the end of the tree traversal, so we put a loop at the end of the element list.

HANDLING DOWNWARD EDGES

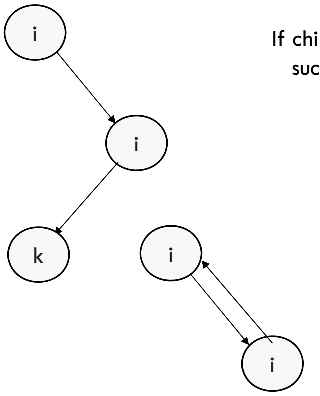
Edge (i,j), such that parent[i]≠j.



```
If child[j]≠NULL
    succ[(i,j)]←(j,child[j])
```

HANDLING DOWNWARD EDGES

Edge (i,j), such that parent[i]≠j.



```
If child[j]≠NULL
    succ[(i,j)]←(j,child[j])
```

```
else
    succ[(i,j)]←(j,i)
```

ie. j is a leaf and the successor is the edge back from the child to the parent.

ASSIGNING EDGE RANKS

After the processors construct the list, they assign position values:

- 1 to those elements corresponding to downward edges
- 0 to those elements corresponding to upward edges.
- Note the root is already handled.

if parent[i]=j, position[(i,j)]←0
Else position[(i,j)]←1

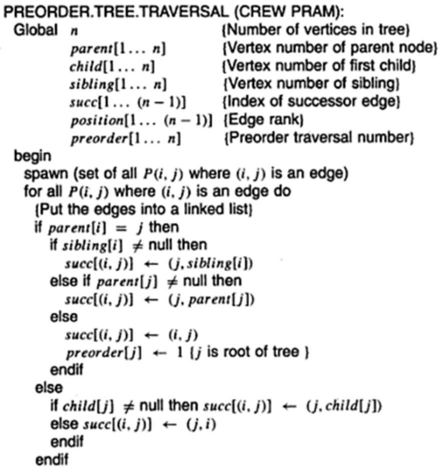
POINTER JUMPING: SUFFIX SUM

The pointer jumping follows subsequently to compute the suffix sum.

The final position values indicate the number of preorder traversal nodes between the list element and the end of the list.

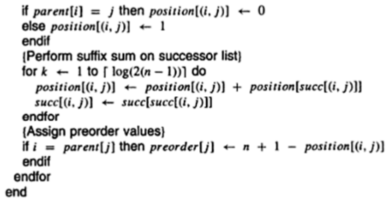
To compute each node’s preorder traversal label compute (n-position+1).

PRAM PROGRAM



```
PREORDER.TREE.TRAVERSAL (CREW PRAM):
Global  n                      {Number of vertices in tree}
        parent[1 ... n]        {Vertex number of parent node}
        child[1 ... n]         {Vertex number of first child}
        sibling[1 ... n]       {Vertex number of sibling}
        succ[1 ... (n − 1)]    {Index of successor edge}
        position[1 ... (n − 1)] {Edge rank}
        preorder[1 ... n]      {Preorder traversal number}
begin
  spawn (set of all P(i, j) where (i, j) is an edge)
  for all P(i, j) where (i, j) is an edge do
    {Put the edges into a linked list}
    if parent[i] = j then
      if sibling[i] ≠ null then
        succ[(i, j)] ← (j, sibling[i])
      else if parent[j] ≠ null then
        succ[(i, j)] ← (j, parent[j])
      else
        succ[(i, j)] ← (i, j)
        preorder[j] ← 1 {j is root of tree }
      endif
    else
      if child[j] ≠ null then succ[(i, j)] ← (j, child[j])
      else succ[(i, j)] ← (j, i)
      endif
    endif
```

PRAM ALGORITHM (CONTD.)



```
    if parent[i] = j then position[(i, j)] ← 0
    else position[(i, j)] ← 1
    endif
    {Perform suffix sum on successor list}
    for k ← 1 to ⌈ log(2(n − 1))⌉ do
      position[(i, j)] ← position[(i, j)] + position[succ[(i, j)]]
      succ[(i, j)] ← succ[succ[(i, j)]]
    endfor
    {Assign preorder values}
    if i = parent[j] then preorder[j] ← n + 1 − position[(i, j)]
    endif
  endfor
end
```

Time Complexity:$O(\lceil \log(n) \rceil$
Processors: O(n)
Cost: O(nlogn)