Parallel List Ranking

Advanced Algorithms & Data Structures

Lecture Theme 17

Prof. Dr. Th. Ottmann

Summer Semester 2006

Two parallel list ranking algorithms

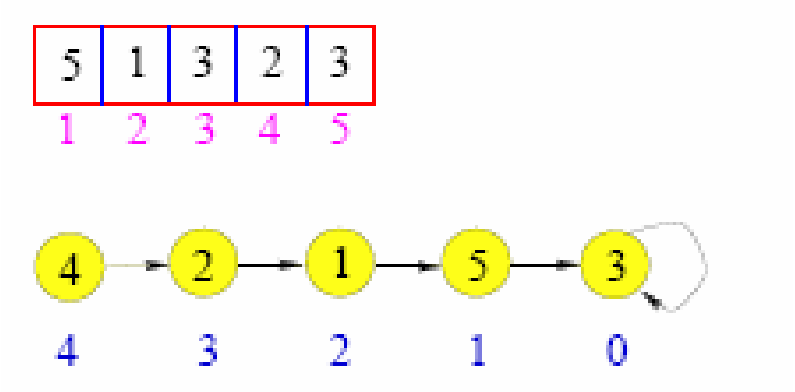
- An $O(\log n)$ time and $O(n \log n)$ work list ranking algorithm.
- An $O(\log n \log\log n)$ time and $O(n)$ work list ranking algorithm.

List ranking

Input: A linked list *L* of *n* elements.

L is given in an array *S* such that the entry *S*(*i*) contains the index of the node which is the successor of the node *i* in *L*.

Output: The distance of each node *i* from the end of the list.



Sequential list ranking algorithm

List ranking can be solved in $O(n)$ time sequentially for a list of length n.

Exercise : Design an algorithm.

•Hence, a work-optimal parallel algorithm should do only $O(n)$ work.

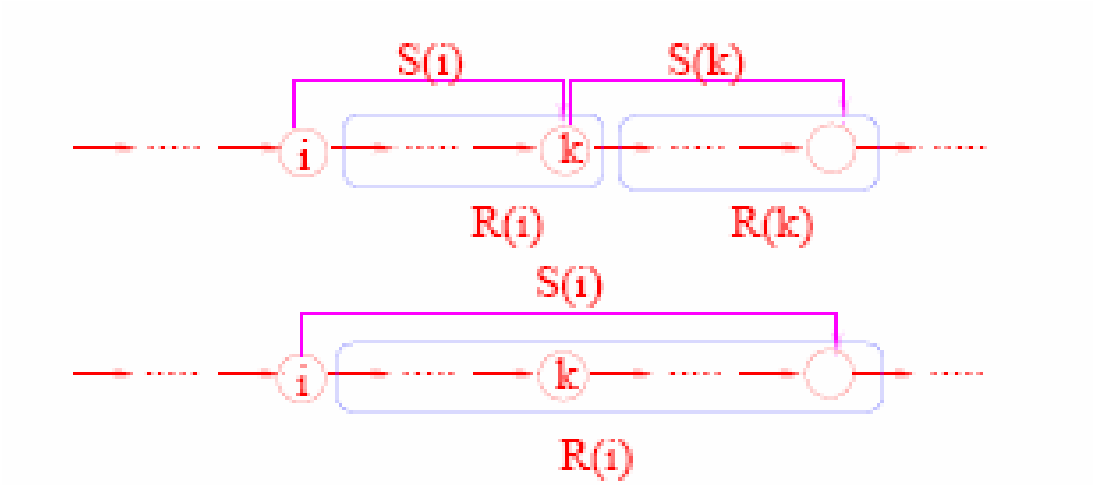
A simple parallel list ranking algorithm

Output: For each $1 \leq i \leq n$, the distance $R(i)$ of node i from the end of the list.

```
begin
        for 1 ≤ i ≤ n do in parallel
                if S(i) ≠ 0 then R(i) := 1
                        else R(i) := 0
        endfor
while S(i) ≠ 0 and S(S(i)) ≠ 0 do
        Set R(i) := R(i) + R(S(i))
        Set S(i) := S(S(i))
end
```

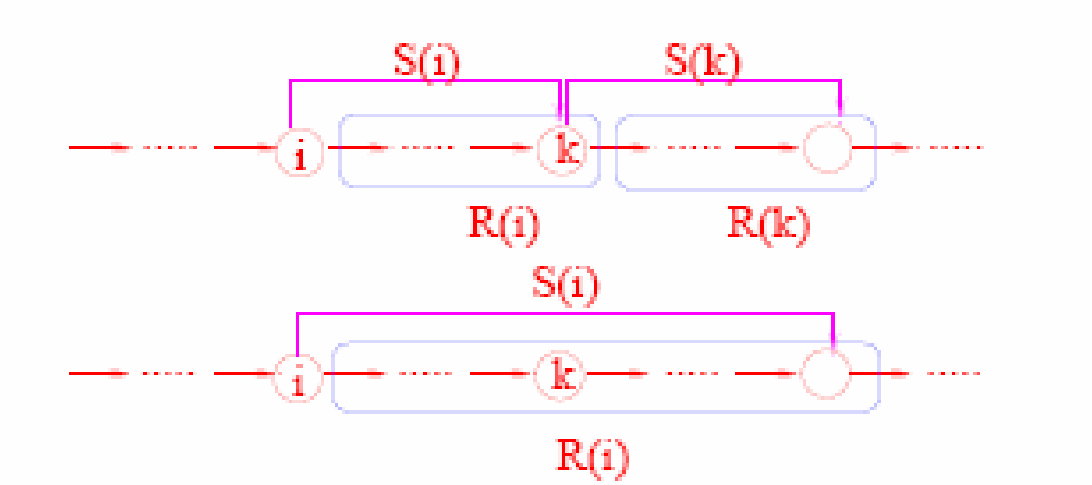
Initialization phase of the list ranking algorithm

- At the start of an iteration of the while loop, *R*(*i*) counts the nodes in a sublist starting at *i* (a subset of nodes which are adjacent in the list).



Iteration phase

- After the iteration, *R*(*i*) counts the nodes in a sublist of double the size.
- When the while loop terminates, *R*(*i*) counts all the nodes starting from *i* and until the end of the list



Complexity and model

- The algorithm terminates after $O(\log n)$ iterations of the while loop.
- The work complexity is $O(n \log n)$ since we allocate one processor for each node.
- We need the CREW PRAM model since several nodes may try to read the same successor (S) values.

Exercise :

Modify the algorithm to run on the EREW PRAM with the same time and processor complexities.

Discussion of the algorithm

- Our aim is to modify the simple algorithm so that it does optimal $O(n)$ work.
- The best algorithm would be the one which does $O(n)$ work and takes $O(\log n)$ time.
- There is an algorithm meeting these criteria, however the algorithm and its analysis are very involved.
- We will study an algorithm which does $O(n)$ work and takes $O(\log n \log\log n)$ time.
- However, in future we will use the optimal algorithm for designing other algorithms.

The strategy for an optimal algorithm

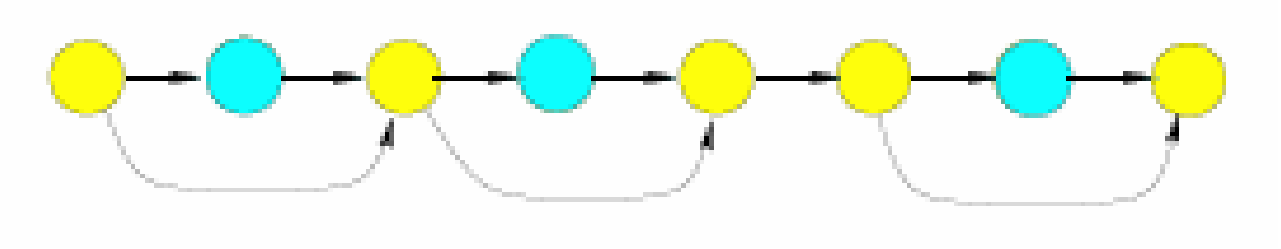
1. Shrink the initial list L by removing some of the nodes.
   The modified list should have $O(n / \log n)$ nodes.
2. Apply the pointer jumping technique (the suboptimal algorithm) on the list with $O(n / \log n)$ nodes.
3. Restore the original list and rank all the nodes removed in Step 1.

The important step is Step1. We need to choose a subset I of nodes for removal.

Independent sets

Definition

A set I of nodes is independent if whenever $i \in I$, $S(i) \notin I$.



The blue nodes form an independent set in this list

- The main task is to pick an independent set correctly.
- We pick an independent set by first coloring the nodes of the list by two colors.

2-coloring the nodes of a list

Definition: A k-coloring of a graph G is a mapping: $c : V \rightarrow \{0,1,\ldots,k-1\}$ such that

$c(i) \neq c(j)$ if $< i, j > \in E$.

- It is very easy to design an $O(n)$ time sequential algorithm for 2-coloring the nodes of a linked list.

We will assume the following result:

Theorem: A linked list with n nodes can be 2-colored in $O(\log n)$ time and $O(n)$ work.

Independent sets

- When we 2-color the nodes of a list, alternate nodes get the same color.
- Hence, we can remove the nodes of the same color to reduce the size of the original list from n to $n/2$.
- However, we need a list of size $\frac{n}{\log n}$ to run our pointer jumping algorithm for list ranking.
- If we repeat the process loglog n time, we will reduce the size of the list to $\frac{n}{2^{\log\log n}}$

  i.e., to $\frac{n}{\log n}$

Preserving the information

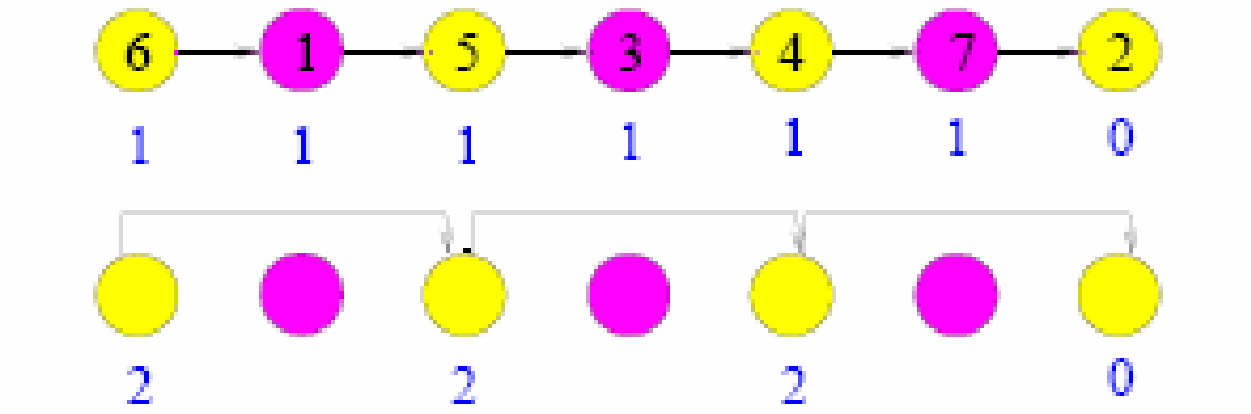
- When we reduce the size of the list to $\frac{n}{\log n}$, we have lost a lot of information because the removed nodes are no longer present in the list.

- Hence, we have to put back the removed nodes in their original positions to correctly compute the ranks of all the nodes in the list.

- Note that we have removed the nodes in $O(\log \log n)$ iterations.

- So, we have to replace the nodes also in $O(\log \log n)$ iterations.

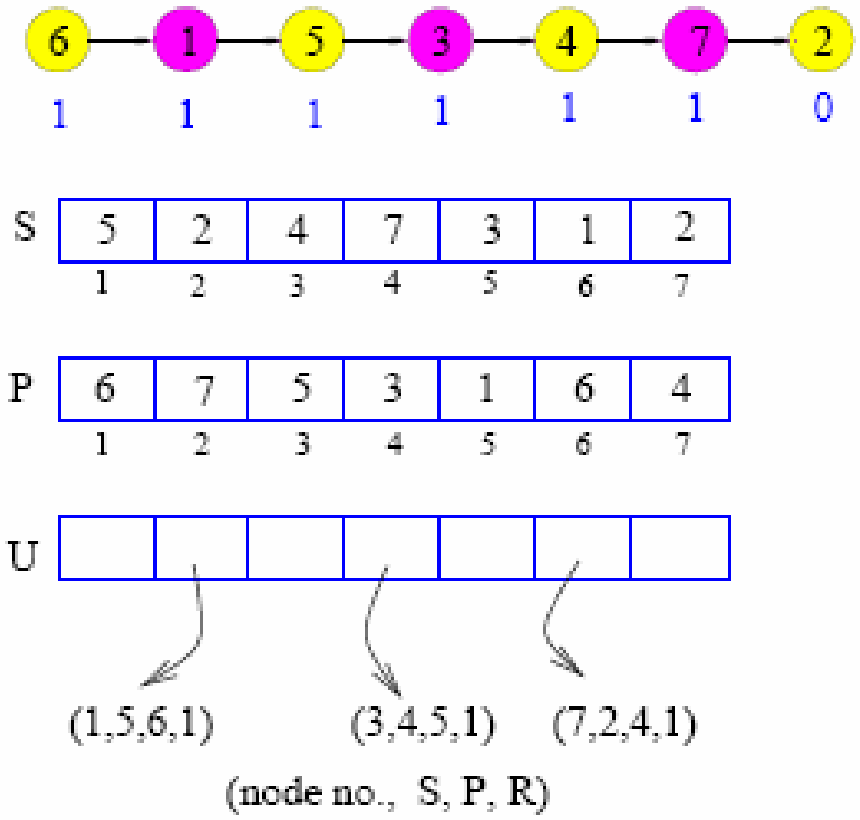
Preserving the information



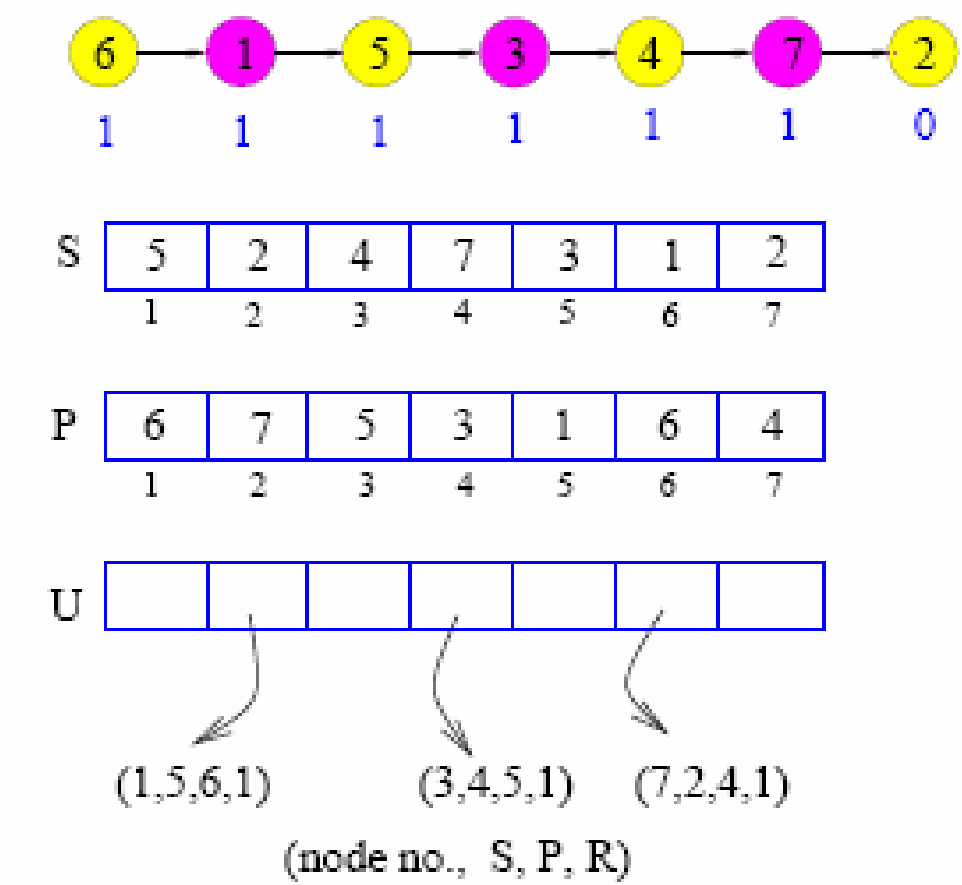
- The magenta nodes are removed through 2-coloring. The number of a node is shown in black.
- The initial rank of each node is shown below the node in the figure.
- The modified rank of an existing yellow node is shown in the second figure.
- The modified rank is with respect to the next existing node in the list.

Preserving the information

- *P* is the array which holds the predecessor indices.
- *S* is the array which holds the successor indices.
- *i* is the number of the removed node.
- *U* is the array where we store information for the removed nodes. This information is used later to restore the removed nodes.



Preserving the information



Preserving the information

The code executed for removing the nodes in the independent set I :

for all $i \in I$ pardo

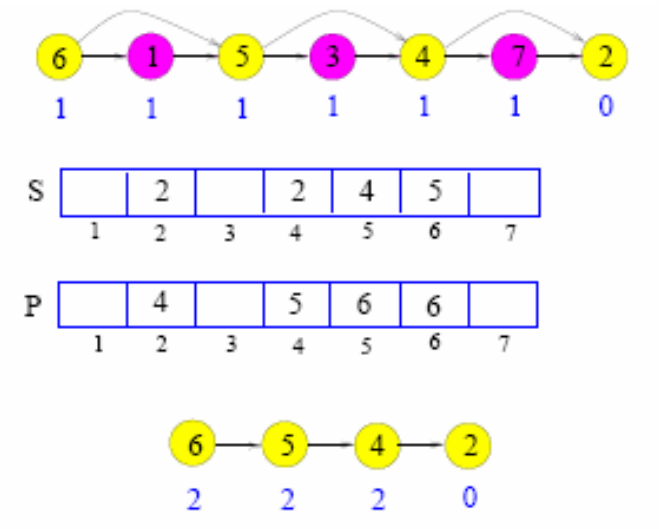
Set $U(i) := (i, S(i), P(i))$

Set $R(P(i)) := R(P(i)) + R(i)$

Set $S(P(i)) := S(i)$

Set $P(S(i)) := P(i)$

Preserving the information



- The predecessor and successor pointers are updated
- The ranks of nodes which are immediately before a removed node are updated.

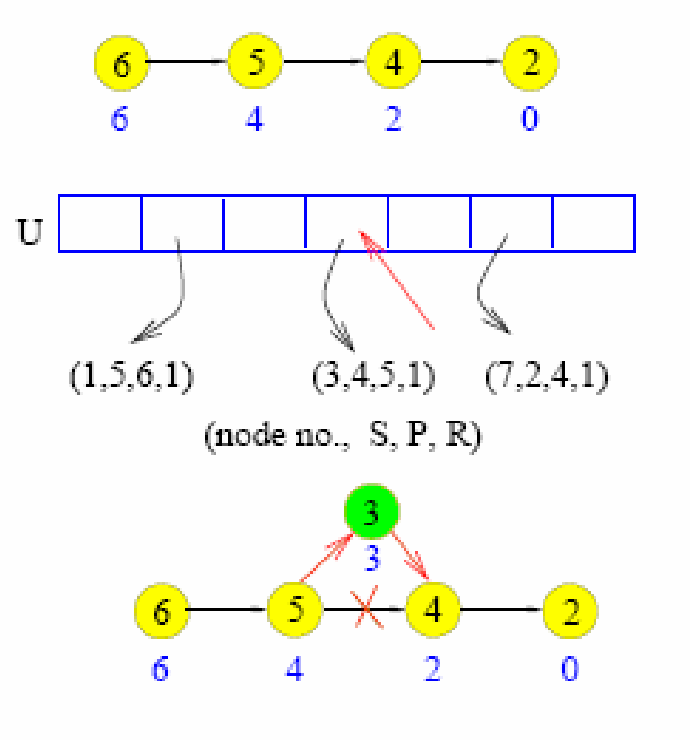
Preserving the information

- The array *P* and *S* can be compacted in $O(n)$ work and $O(\log n)$ time after each phase of removal of nodes. This is done through a parallel prefix computation.

- Recall that we have to do the removal of nodes for loglog *n* phases to reduce the number of nodes to $\frac{n}{\log n}$.

- We can keep a separate *U* array for each phase.

Putting back a node



- A removed node can be reinserted in $O(1)$ time

Complexity

- 2-coloring in each stage can be done in O(log n) time and O(n) work.
- The compaction of the P and S arrays is done through a parallel prefix computation in O(log n) time and O(n) work.
- There are in all O(log log n) stages in which we have to remove nodes. Each stage takes O(log n) time due to the 2-coloring and parallel prefix computations.
- At the end, we do one list ranking for a list of size O(n/log n). This takes O(n) work and O(log n) time.
- So, the overall complexity is O(n) work and O(log n loglog n) time.