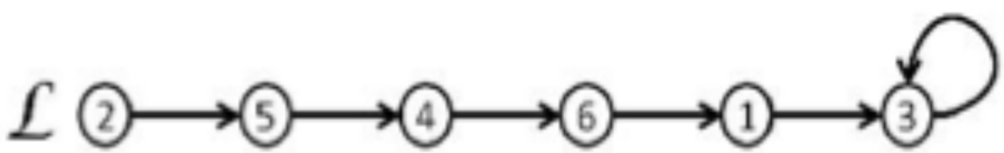
List Ranking

Chapter 4

Problem on linked lists
2-level memory model

- List Ranking problem

  Given a (mono directional) linked list L of n items, compute the distance of each item from the tail of L.



Id	1	2	3	4	5	6
Succ	3	5	3	6	4	1
Rank	1	5	0	3	4	2

List ranking

- Easy sequential solution in the RAM model, O(n).
  - Compute the predecessor of I, such that Pred[Succ[i]] = i;
  - Scan the list starting from the tail, setting Rank[tail]=0 and incrementing the value at each item.
- Recursive solution

```
ListRank(i):
 if (Succ[i]==i) Rank[i]=0;
    else Rank[i]=ListRank(Succ[i]) +1;
```

First call: ListRank(head)

O(n) and no additional space to store Pred.

2-level model

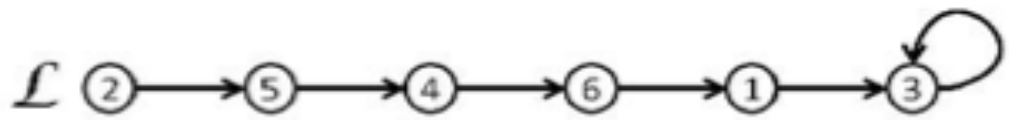
- Very bad in this model $\Theta(n)$ I/O accesses!!
  far from the lower bound $\Omega(n/B)$.
- Solution: use a technique coming from the theory of parallel algorithms, the pointer jumping, that can be done in parallel for every item.
- At each iteration update the pointer with the pointer of the pointed item:
- Succ[i]= Succ[succ[i]] and compute the rank accordingly.

Parallel List ranking

```
1: for  1 ≤ i ≤ n  pardo
      if Succ(i) == i then Rank(i) = 0
          else Rank(i) = 1
2: for  1 ≤ i ≤ n  pardo
      while (Succ(i) ≠ i) do
              Rank(i) := Rank(i) + Rank(Succ(i));
              Succ(i) := Succ(Succ(i))
      end
```

Parallel List ranking

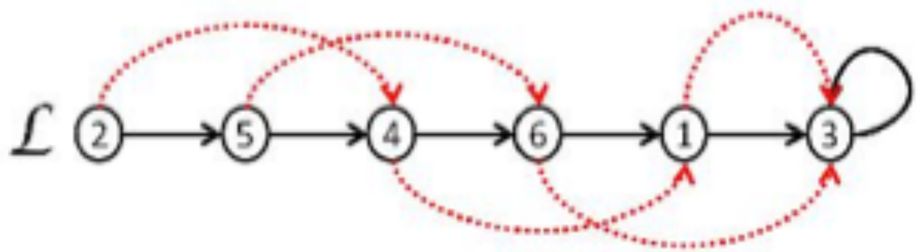
Initial step:



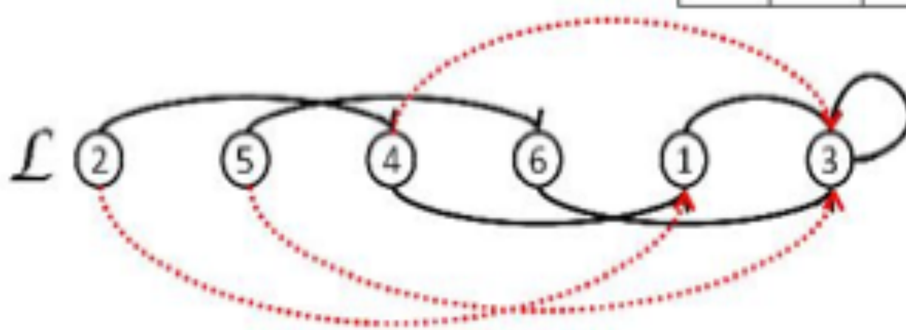
Id	1	2	3	4	5	6
Succ	3	5	3	6	4	1
Rank	1	1	0	1	1	1

Pointer Jumping

Step 2 and 3 of the while:



Rank	1	2	0	2	2	2



Rank	1	4	0	3	4	2

At step 4 (last) only Rank[2] becomes 5.

Parallel List ranking

The parallel algorithm, using n processors, takes O(logn) time and O(nlogn) operations.

Observation: The distances from the tail, at each step, of pointer jumping do not grow linearly, but duplicate. This means that the most distant item will take O(logn) steps to be ranked.

The overall operations are O(nlogn) because, at each step. O(n) processors are working in parallel.

Idea: Use the simulation of the pointer jumping technique for the 2-level model and Sort and Scan primitives for Triples.

Parallel algorithm simulation in a 2-level model

The simulation is performed via a constant number of Sort and Scan primitives over n triples of integers.

Sort is very complicate in the 2-level model (see future lectures). We use here a primitive of complexity $\tilde{O}(n/B)$ I/Os operations,

$\tilde{O}$: polylog factors (in M, n, B) are hidden.

Scan is easy and takes $O(n/B)$ I/Os operations.

Express the two basic parallel operations in the same way:

Rank (i) += Rank(Succ(i))

Succ(i) =Succ(Succ(i))

A(ai) op A(bi)

op is sum and a assignment for the Rank array (A=Rank)

op is a copy for the update of the Succ array (A=Succ)

Parallel simulation in a 2-level model

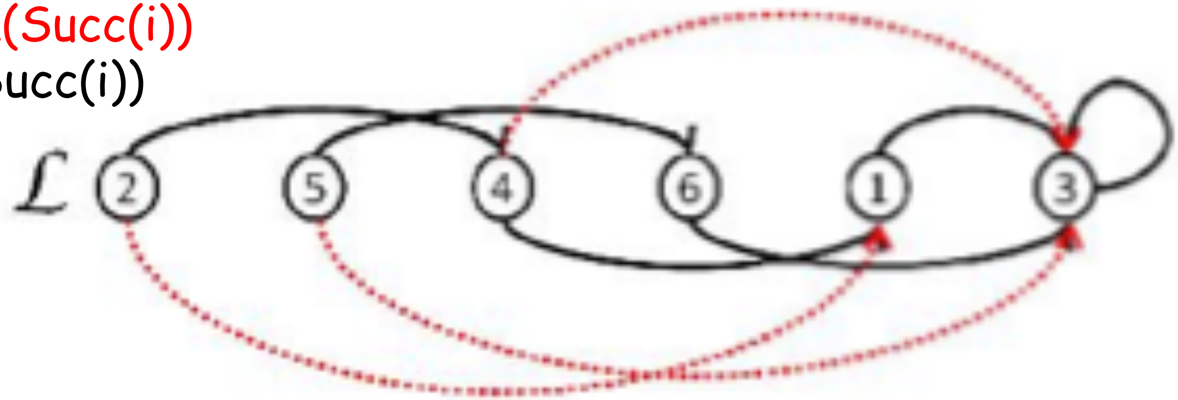
The simulation of A(ai) op A(bi) can be implemented simultaneously over all i=1,2…,n. 5 steps:

1. Scan the disk and create a triple <ai, bi, 0>.
2. Sort the triples according to the second component;
3. Scan the triples and array *A* to create the new triple <ai, bi, *A*[bi]>. The coordinate scan allows to copy *A*[bi] into the triple.
4. Sort the triples according to the first component (ai).
5. Scan the triples and the array *A* and, for every triple update the memory content of cell *A*(ai).

Parallel simulation in a 2-level model

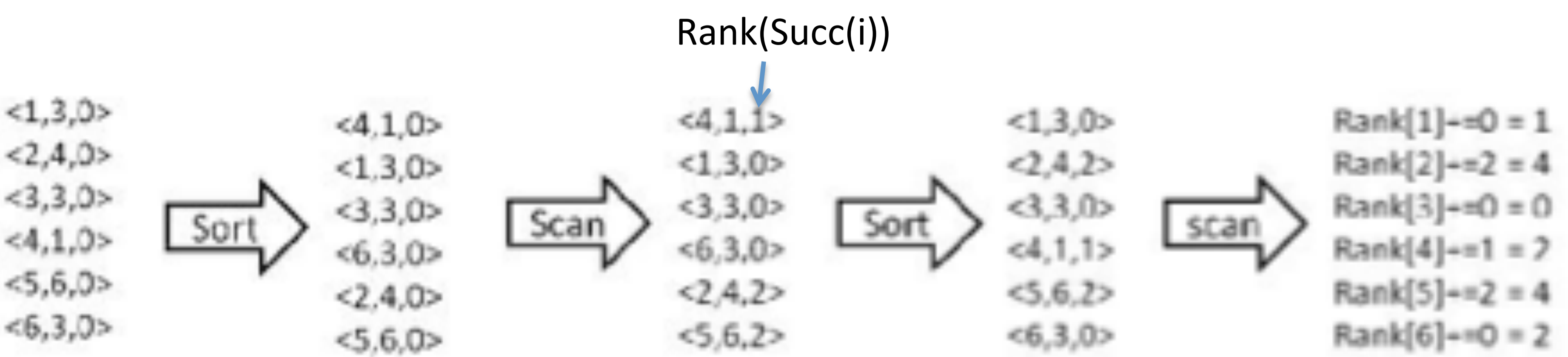
Rank (i) += Rank(Succ(i))

Succ(i) =Succ(Succ(i))



Item	1	2	3	4	5	6
Rank	1	2	0	2	2	2
Succ	3	4	3	1	6	3

Item	1	2	3	4	5	6
Rank	1	4	0	3	4	2
Succ	3	1	3	3	3	3



Parallel simulation in a 2-level model

More general result:

Every parallel algorithm using n processors and taking T steps can be simulated in a 2-level memory by a disk-aware sequential algorithm taking ($\tilde{O}(n/B)$ T) I/Os operations, and O(n) space.

It is convenient when: T = o(B) that is sub-linear number of I/O's.

Exploit algorithms for the PRAM model

Parallel simulation in a 2-level model: with Divide&Conquer

Divide&Conquer

- Divide: Divide the problem in k sub-problems of size $n_1, \ldots, n_k$.
- Conquer: Solve the sub-problems recursively, or directly if $n_k = O(1)$.
- Combine: Combine the sub-problems to fins the solution to the original problem.

Complexity with recursion relation:

$$T(n) = D(n) + R(n) + \sum_{i=1,\ldots,k} T(n_i)$$

Master Theorem to solve recurrence of the kind:

$$T(n) = aT(n/b) + f(n)$$

With $a \geq 1$, $b > 1$, constant, $f(n)$ be a function

Master Theorem for recurrence relations

$T(n) = aT(n/b) + f(n)$
With $a \geq 1$, $b > 1$, constant, $f(n)$ a function.
$T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon>0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon>0$, then $T(n) = \Theta(f(n))$ if the regolarity condition holds.

Regolarity c.: $a(f(n/b) \leq cf(n)$ for some constant $c<1$ and sufficiently large n.

Ex: $T(n) = 4T(n/2)+n$; $T(n) = 4T(n/2)+n^2$; $T(n) = 4T(n/2)+n^3$.

Master Theorem for recurrence relations

$T(n) = aT(n/b) + f(n)$
With $a \geq 1$, $b > 1$, constant, $f(n)$ a function.
$T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a-\varepsilon})$ for some constant $\varepsilon>0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a}\log n)$
3. If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon>0$, then $T(n) = \Theta(f(n))$ if the regolarity condition holds.

Regolarity c.: $a(f(n/b) \leq cf(n)$ for some constant $c<1$ and sufficiently large n.

Ex: $T(n) = 4T(n/2)+n$; $T(n) = 4T(n/2)+n^2$; $T(n) = 4T(n/2)+n^3$.

Master Theorem for recurrence relations

1. $T(n) = 4T(n/2)+n$;
2. $T(n) = 4T(n/2)+n^2$;
3. $T(n) = 4T(n/2)+n^3$.

1) $a=4.\ b=2,\ n^{\log_b a} = n^2,\ f(n) = n$: $f(n)=O(n^{\log_b a-\varepsilon})$, per $0\leq\varepsilon\leq 1$

   case 1 of Th $T(n) = \Theta(n^2)$

2) $a=4,\ b=2,\ n^{\log_b a} = n^2,\ f(n)=n^2$: $f(n)=\Theta(n^{\log_b a})$

   case 2 of Th $T(n)= \Theta(n^2 \log n)$

3) $a=4,\ b=2,\ n^{\log_b a} = n^2, f(n) = n3$: $f(n)=\Omega(n^{\log_b a+\varepsilon})$, per $0\leq\varepsilon\leq 1$

   Case 3 of Th $T(n)= \Theta(n^3)$

   se $af(n/b)\leq cf(n)$ $4(n/2)^2\leq cn^2$ $n^2\leq cn^2$, true for $c<1$.

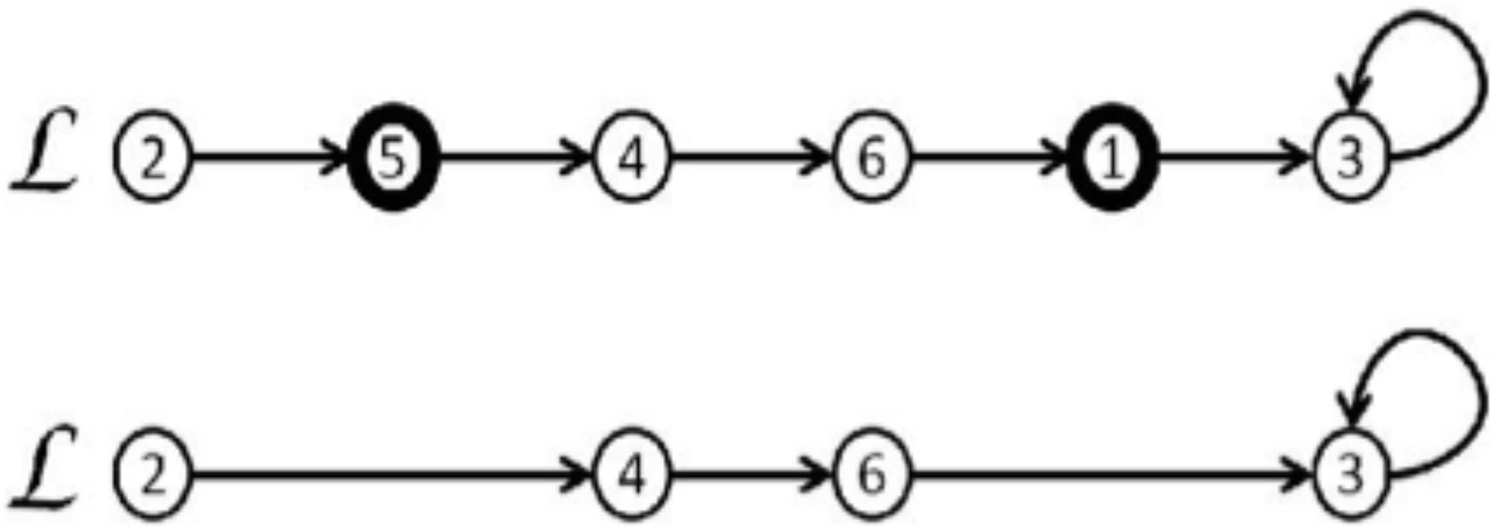
A Divide&Conquer approach for List Ranking

1. Divide: Identify a set I of items from the list L, such that I is an independent set for L, that is the successor of each item in I does not belong to I.

   |I|≤n/2. In the alg |I| is also kept ≥ n/c
2. Conquer: Form the list L* =L-I by pointer jumping on the predecessor of the items in I. At any step, rank[x] is the distance between x and the current succ[x] in the input list. Solve recursively on L*, where n/2≤|L*|≤ (1-1/c)/n.
3. Combine: Assume that the list rank is correctly been computed for all L*. Now derive the final rank of each item x in I as rank[x]=1+rank[succ[x]] as for pointer jumping.

Divide&Conquer for List Ranking

Rank	1	1	0	1	1	1



Rank	1	**2**	0	1	1	**2**

I= {5, 1} Rank update su L*: Rank[2] = 1+Rank[Succ[5]]= 1 +Rank[4]=2
Rank[6] = 1+Rank[Succ[6]] =Rank[6]+Rank[1] = 2

Correctness of rank computation

1. The independent set property on I assures that Succ[i] ε L*, so its rank is available.
2. By induction: Rank(Succ[x]] accounts for the distance of x from the tail of L and and Rank[x] accounts for number of items between x and Succ(x) in the input list.

Pointer Jumping is applied only to the predecessors of the removed items and the others have their Succ pointer unchanged.

The I/O efficiency of the algorithm depends onto the Divide step.

List ranking over L of n elements

I/O's complexity via Divide&Conquer:

$T(n) = I(n) + T((1-1/c))n + \tilde{O}(n/B)$

Where I(n) is the cost of selecting the independent set;
$\tilde{O}(n/B)$ is for the recombine step that can be solved by a costant number of Sort and Scan as before.

Identify a large independent set I from L avoiding many I/Os :

1. Randomized solution
2. Deterministic coin tossing

Select an independent set from L by randomized solution

Algorithm: toss a fair coin for each item i of L.

If coin(i) = H select item i if Coin(Succ(i))=T.

Probability: item I is selected with prob. ¼ (this happens for the configuration HT)

Algorithm repeats the coin tossing until $|I| \geq n/c$, for some $c > 4$. By Chernoff bound it can be proved that the repetition is executed a small number of time.

The check for the coin values, for selecting the I's items, can be simulated via Sort and Scan primitives in $\tilde{O}(n/B)$ I/O's on average.

Hence: $T(n) = \tilde{O}(n/B) + T(3/4n)$ and by Master Th:

$T(n) = \tilde{O}(n/B)$ on average.

Select an independent set from L by deterministic coin tossing

Simulate deterministically the coin-tossing.

Instead of assigning to each item 2 possible values (H,T) assign n values (0,1, …, n-1) that eventually will be reduce to 3 (0,1,2).

Selection: Pick the items that are local minima, that is their values is less than its two adjacent items.

Algorithm

Initialize Assign to each item i coin(i) = i-1. The binary representation of coin(i), $bit_b(i)$ takes $b = \lceil \log n \rceil$.

Deterministic coin tossing

Get 6-coin values. Repeat the following steps until $\mathtt{coin}(i) < 6$, for all i:

- Compute the position $\pi(i)$ where $\mathtt{bit}_b(i)$ and $\mathtt{bit}_b(\mathtt{Succ}[i])$ differ, and denote by $z(i)$ the bit-value of $\mathtt{bit}_b(i)$ at that position.
- Compute the new coin-value for i as $\mathtt{coin}(i) = 2\pi(i) + z(i)$ and set the new binary-length representation as $b = \lceil \log b \rceil + 1$.

Get just 3-coin values. For each element i, such that $\mathtt{coin}(i) \in \{3, 4, 5\}$, do $\mathtt{coin}(i) = \{0, 1, 2\} - \{\mathtt{coin}(\mathtt{Succ}[i]), \mathtt{coin}(\mathtt{Pred}[i])\}$.

Select I. Pick those items i such that $\mathtt{coin}(i)$ is a local minimum, namely it is smaller than $\mathtt{coin}(\mathtt{Pred}[i])$ and $\mathtt{coin}(\mathtt{Succ}[i])$.

Deterministic coin tossing
Example

$Bit_b(i)$	$Bit_b(succ(i))$	$\pi(i)$	$z(i)$	new coin(i)
01111011_2	00101111_2	2	0	4
00101111_2	01101011_2	2	1	5
01101011_2	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\cdots$				
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
01111011_2	00101111_2	2	0	4
00101111_2	01101111_2	6	0	12

Deterministic coin tossing

$Bit_b(n)=128$ $n=2^{128}$

From b to logb+1

$$2^{128} \rightarrow 2 \cdot 128 = 2^8,$$

$$2^8 \rightarrow 2 \cdot 8 = 2^4,$$

$$2^4 \rightarrow 2 \cdot 4 = 2^3,$$

$$2^3 \rightarrow 2 \cdot 3 = 6.$$

Deterministic coin tossing

Get 6-coin values   The step is repeated until coin(i) <6 for all i. Coin(i)={0,1,…,5}

Observe: for all i :   coin(i) is different from coin(i) of its adjacent elements.

Proof by contraddiction: assume coin(i) = coin(succ(i)) then $2\pi(i)+z(i) = 2\pi(succ(i))+z(succ(i))$ and it must be $z(i) = z(succ(i))$ then this is contradiction since I and succ(i) differs at position $\pi(i)$. A similar argument holds for i and pred(i).

In addition: $2\pi(i)+z(i) \leq 2(b-1)+1=2b-1$

This max value can be represented by $\lceil \log b \rceil+1$ bits

Deterministic coin tossing

Get 3-coin values

The different values of coin(i) are (0,1, …5), since every 2 adjacent c(i) are different.

Hence:

if coin(i)= {3,4,5} the new value will be {0,1,2} – {coin(pred(i)), coin(succ(i))}

Deterministic coin tossing

The number of step to get 6 values {0,1,…, 5} is log*n.
At each step:

b bits   becomes   logb+1 bits.

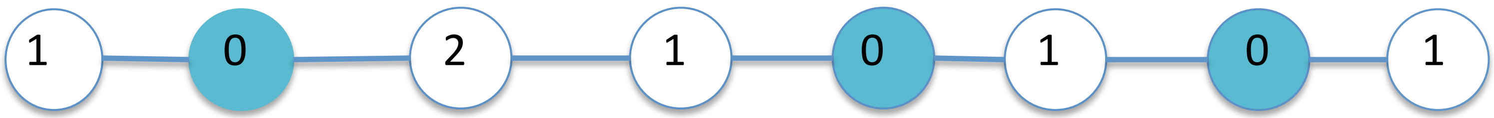
Log*n is the repeated application of the log function until value 1 is reached.

16
log(16)=4
log(4)=2
log(2)=1

log*(16)=3

Log*n is a function that grows very very slowly!

Select independent set
Local minima



The list ranking problem is solved with coin tossing alg. with $\tilde{O}(n/B)$ worst case I/Os