List Ranking

- List ranking is a fundamental problem in parallel computing.
- Given a list of elements, find the distance of the elements from one end of the list.
- In sequential computation, not a serious problem.
  - Can simply traverse the list from one end.
- But this approach does not scale well for parallel architectures.

List Ranking

Rank[i] = j if i is the j^{th} element of the linked list.

List

i=10

1 → 8 → 5 → 11 → 2 → 6 → 10 → 4 → 3 → 7 → 12 → 9

Succ S

1	2	3	4	5	6	7	8	9	10	11	12
8	6	7	3	11	10	12	5	-- (-1)	$\boxed{4}$	2	9

Rank

1	2	3	4	5	6	7	8	9	10	11	12
1	5	9	8	3	6	10	2	12	7	$\boxed{4}$	11

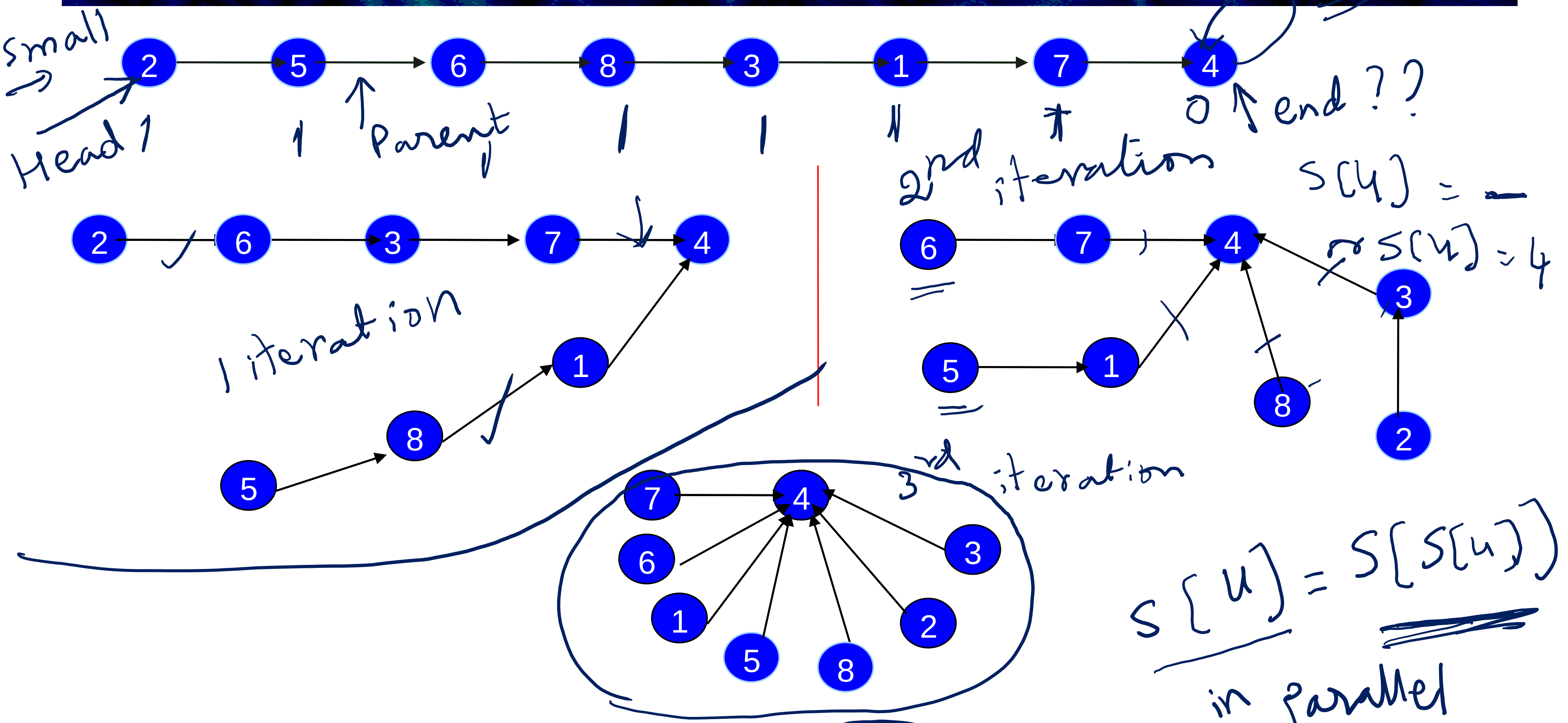
- Representation via an array of successor pointers.

index i S[i] stores the "next" value of element i in the linked list.

S[2] = 6

Pointer Jumping Solution

Wyllie 1980



- Each node updating its parent to be its grandparent.

S[7] = 4 S[S[7]] = — or 4

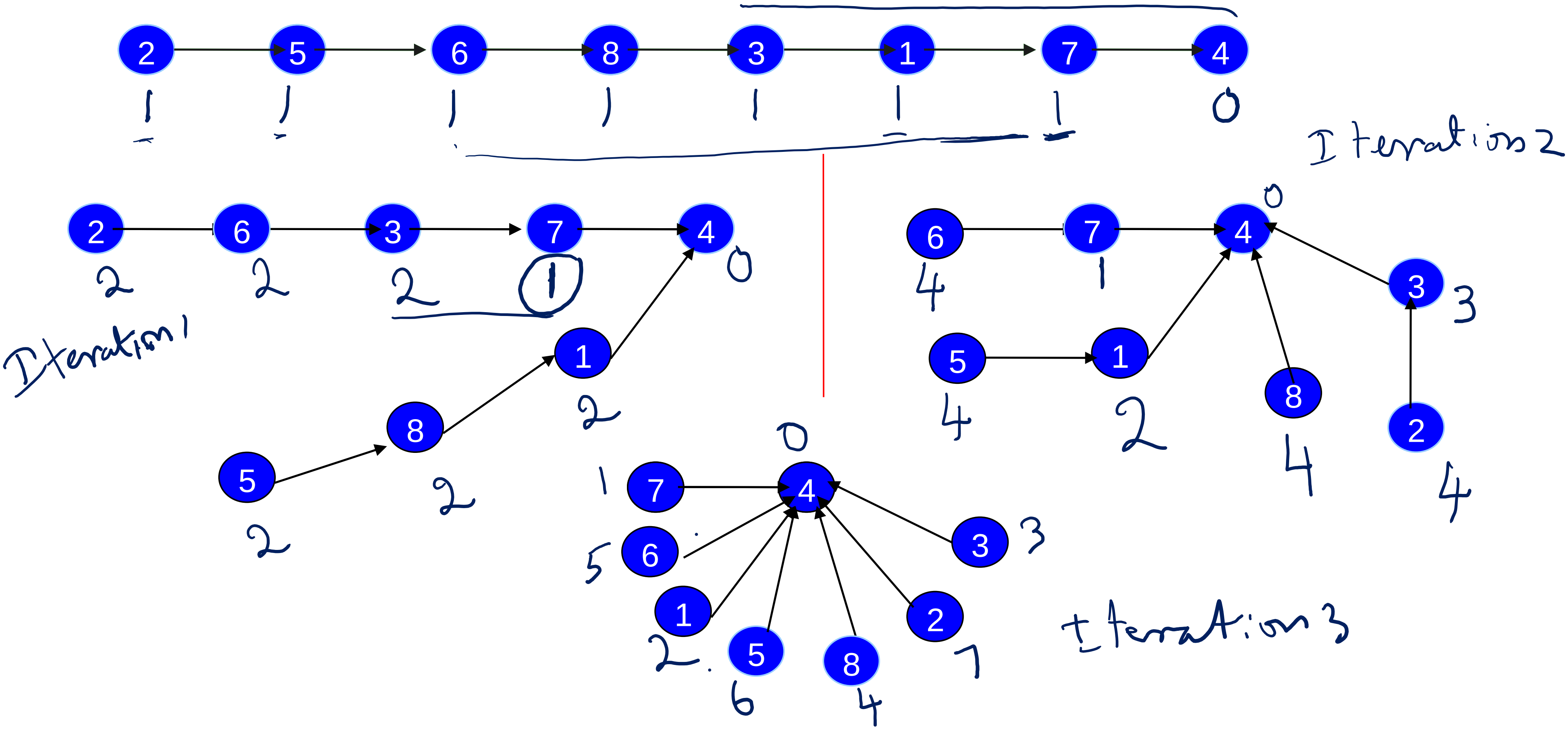
Pointer Jumping Solution


```
Algorithm FindRoot
for 1 ≤ i ≤  n do in parallel
    R(i) = 1
    R(i) = 0 if node i is the last node
    while P(i) ≠ P(P(i)) do
        R(i) = R(i) + R(P(i))
        P(i) = P(P(i))
end.
```

R[i]

- The pseudo code above computes the rank of every element in parallel.
  - R() refers to the rank, P() refers to the parent.

Pointer Jumping Solution

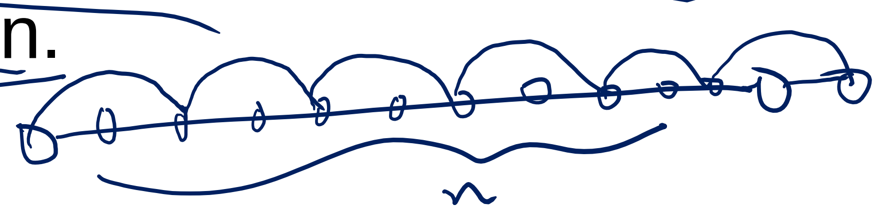


- Each node updating its parent to be its grandparent.

Pointer Jumping Solution

```
Algorithm FindRoot
for 1 ≤ i ≤  n do in parallel
     R(i) = 1
     R(i) = 0 if node i is the last node
     while P(i) ≠ P(P(i)) do
          R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

- Claim: Algorithm FindRoot finishes in O(log n) time. iterations
- Proof: Show that the distance between a node and the root reduces by a factor of 2 every iteration of the while loop.
  - Maximum distance is n.



Pointer Jumping Solution

```
Algorithm FindRoot
for 1 ≤ i ≤  n do in parallel
     R(i) = 1
     R(i) = 0 if node i is the last node
     while P(i) ≠ P(P(i)) do
          R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

seq time complexity $= O(n)$

- Claim: The above algorithm has a work complexity of O(n log n).
- Proof: Each processor needs at most O(log n) work.
- Therefore, our algorithm is sub-optimal.
  - Can be made optimal using Technique 1. Details follow.

Pointer Jumping Solution

```
Algorithm FindRoot
for 1 ≤ i ≤  n do in parallel
     R(i) = 1
     R(i) = 0 if node i is the last node
     while P(i) ≠ P(P(i)) do
          R(i) = R(i) + R(P(i))
          P(i) = P(P(i))
end.
```

- Few implementation issues
  - In the PRAM model, synchronous execution means that all n processors execute each step in the while loop at the same time.
  - Any problems otherwise?

Pointer Jumping Solution

```
Algorithm FindRoot
for 1 ≤ i ≤  n do in parallel
     R(i) = 1
     R(i) = 0 if node i is the last node
     while P(i) ≠ P(P(i)) do
          R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

- Few implementation issues
  - In the PRAM model, synchronous execution means that all n processors execute each step in the while loop at the same time.
- Any problems otherwise?
  - Inconsistent results!

Pointer Jumping Solution

```
Algorithm FindRoot
for 1 ≤ i ≤  n do in parallel
     R(i) = 1
     R(i) = 0 if node i is the last node
     while P(i) ≠ P(P(i)) do
          R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

atomic

word

P | R

64 bits

- To get around, one can consider packing R and P values of a node into a single word.
- If list has no more than 2^{32} elements, can use 64 bit architectures with each word packing two 32 bit numbers.
- Synchronize iterations to get consistent results.

Pointer Jumping Solution

```
Algorithm FindRoot
for 1 ≤ i ≤  n do in parallel
     R(i) = 1
     R(i) = 0 if node i is the last node
     while P(i) ≠ P(P(i)) do
          R(i) = R(i) + R(P(i))
         P(i) = P(P(i))
end.
```

- Claim: The above algorithm has a work complexity of $O(n \log n)$.
- Therefore, our algorithm is sub-optimal.
  - Can be made optimal using Technique 1. ~~Details follow.~~

Advanced Optimal Solutions

$n \log n$

$n' = n/\log n$

1 → 8 → 5 → 11 → 2 → 6 → 10 → 4 → 3 → 7 → 12 → 9

Succ	8	6	7	3	11	10	12	5	--	4	2	9

8 11

$n' \log n' = \frac{n}{\log n} \log \frac{n}{\log n} = n$

- General technique suggests that we solve a smaller problem and extend the solution to the larger problem.
- To apply our technique we should use the pointer jumping based solution on a sub-list of size n/log n.
- How to identify such a sublist?
  - More so given that the input is an array of successor elements.
  - Cannot take equi-distant parts of the array since that may not be a valid list anymore.