Parallel Algorithms and Parallel Computers (ii)

IN4026

Lecture 2

Cees Witteveen

Algorithmics Group

Electrical Engineering, Mathematics and Computer Science

TU Delft

Plan of Lecture 2

- Exercises
  - proof of correctness: program cumulative frequencies
  - construction program for detecting weakly increasing runs

during lecture

- Prefix computations and balanced trees
  - inner product computations
  - matrix multiplication
- Pointer jumping
  - searching for the root of a tree
  - determining the distance to the root
- Divide & Conquer
  - finding the minimum 1-index in an array
  - finding the maximum of an array

Last week's exercise

Let A[1..n] be an array of n integers.
A is said to contain a *weakly increasing run* of size k if there exists a j ≥ 1, such that A[i] ≤ A[i+1] for i = j,…, j+(k-1).

Given such an array A and a number k, decide in $O(\log n)$ parallel time whether A contains a weakly increasing run of at least size k.

Solution

Idea:

1. For i=1, ..., n-1, copy the results of comparing $A[i] \leq A[i+1]$ in an array B[0..n], where B[0] = B[n] = 0.

2. Use a balanced tree computation to determine the length of the runs and whether there is a run of at least length k.

To this end we need to define a new binary associative operator # that can be used to determine lengths of runs.

We use an example to help in defining this operator.

Example

We have an array A = 1 1 2 4 1 2 1 2 2 3 5 6 4 2
and need to decide whether A contains a run of length k ≥ 5

1. copy results of comparing A[i] with A[i+1] in an array B

A = 1 1 2 4 1 2 1 2 2 3 5 6 4 2

B = 0 1 1 1 0 1 0 1 1 1 1 1 0 0 0 0 % fill until n is power of 2

Example

We have an array A = 1 1 2 4 1 2 1 2 2 3 5 6 4 2
and need to decide whether A contains a run of length k ≥ 5

1. copy results of comparing A[i] with A[i+1] in an array B

A = 1 1 2 4 1 2 1 2 2 3 5 6 4 2

B = 0 1 1 1 0 1 0 1 1 1 1 1 0 0 0 0   % fill until n is power of 2

2. determine run lengths by balanced tree method

B: 0 1 1 1 0 1 0 1 1 1 1 1 0 0 0 0

\# # # # # # # #

B: (0,1) 2 (0,1) (0,1) 2 2 0 0   % keep track of boundaries of runs

\# # # #

B: (0,3) (0,1) 4 0   % if you see a run of insufficient length, delete it

\# #

B: (0,1) (4,0)

\#

B: true   % the previous parts form a run of length 6

Solution (i): defining the operator

$$
\begin{array}{lll}
0 \# 0 & = 0 & \\
0 \# x & = (0,x) & \text{if } x > 0 \\
0 \# (0,x) & = (0,x) & \\
0 \# (x,0) & = \text{true} & \text{if } x \geq k-1 \\
 & = 0 & \text{else} \\
0 \# (x,0,y) & = \text{true} & \text{if } x \geq k-1 \\
 & = (0,y) & \text{else}
\end{array}
$$

$$
\begin{array}{ll}
(0,x) \# 0 & = \text{true if } x \geq k-1 \\
 & = 0 \text{ else} \\
(0,x) \# y & = (0, x+y) \quad y > 0 \\
(0,x) \# (0,y) & = \text{true if } x \geq k-1 \\
 & = (0,y) \text{ else} \\
(0,x) \# (y,0) & = \text{true if } x+y \geq k-1 \\
 & = 0 \text{ else} \\
(0,x) \# (y,0,z) & = \text{true if } x+y \geq k-1 \\
 & = (0,z) \text{ else}
\end{array}
$$

$$
\begin{array}{ll}
(x,0) \# 0 & = (x,0) \\
(x,0) \# y & = (x,0,y) \\
(x,0) \# (y,0) & = \text{true if } y \geq k-1 \\
 & = (x,0) \text{ else} \\
(x,0) \# (0,y) & = (x,0,y) \\
(x,0) \# (y,0,z) & = \text{true if } y \geq k-1 \\
 & = (x,0,z) \text{ else}
\end{array}
$$

$$
\begin{array}{lll}
x \# y & = x+y & \text{if } x,y > 0 \\
x \# 0 & = (x,0) & \text{if } x > 0 \\
x \# (0,y) & = (x,0,y) & \text{if } x,y > 0 \\
x \# (y,0) & = \text{true} & \text{if } x+y \geq k-1 \\
 & = (x+y,0) \text{ else} & \\
x \# (y,0,z) & = \text{true} & \text{if } x+y \geq k-1 \\
 & = (x+y,0,z) \text{ else} &
\end{array}
$$

$$
\begin{array}{ll}
(x,0,y) \# 0 & = \text{true if } y \geq k-1 \\
 & = (x,0) \text{ else} \\
(x,0,y) \# z & = \text{true if } y+z \geq k-1 \\
 & = (x,0,y+z) \text{ else} \\
(x,0,y) \# (z,0) & = \text{true } y+z \geq k-1 \\
 & = (x,0) \text{ else} \\
(x,0,y) \# (0,z) & = \text{true if } y \geq k-1 \\
 & = (x,0,z) \text{ else} \\
(x,0,y) \# (u,0,v) & = \text{true if } y+u \geq k-1 \\
 & = (x,0,v) \text{ else}
\end{array}
$$

$$
\begin{array}{l}
\bullet \, \# \, \text{true} = \text{true} \\
\text{true} \, \# \, \bullet = \text{true}
\end{array}
$$

Solution (ii): using the balanced tree

Use the following balanced tree scheme:

```
begin
    1. for 1 ≤ i ≤ n-1 pardo
            if ( A[i] ≤ A[i+1] ) then B[i] := 1 else B[i] := 0;
       B[0] :=0, B[n] :=0;
    2. for  h = 1 to log n+1 do
            for 0 ≤ j ≤ ((n +1)/2^h) -1 pardo
                    B[j] := B[2j] # B[2j+1];
    3. return B[0] ;
end
```

$T(n) = O(\log n)$
$W(n) = O(n)$

Plan of Lecture 2

- Exercises
  - proof of correctness program cumulative frequencies
  - construction program for detecting weakly increasing runs
- Prefix computations and balanced trees
  - inner product
  - matrix multiplication
- Pointer jumping
  - searching for the root of a tree,
  - determining the distance to the root
- Divide & Conquer
  - finding the minimum 1-index in an array
  - finding the maximum of an array

Inner product computations

- Let $\mathbf{u}$ = [u_i] and $\mathbf{v}$ = [v_j] be two n x 1 column vectors.
  The inner product $\mathbf{u}^T v$ is defined as

$$\mathbf{u}^T \mathbf{v} = \sum_{i=1..n} u_i v_i = u_1 \times v_1 + u_2 \times v_2 + \ldots + u_n \times v_n$$

Computing the inner product by a balanced tree method:

```
input:   U[1..n], V[1..n]  where n = 2^k;  output: U^T. V
begin
  1. for 1 ≤ i≤ n pardo
         C[i] = U[i] × V[i];
  2. for h=1 to log n do
         for 1 ≤ k ≤ n/2^h pardo
                 C[k] := C[2k-1] + C[2k];
  3. return  C[1];
end
```

$T(n) = O(\log n)$, $W(n) = O(n)$

Matrix-vector product

input: A_{nxn}, B_{nx1}, $n = 2^k$;

output: $C_{nx1} = A \times B$

Remember:
A.B can be written as a vector of inner products $A[i]^T.B$

begin

1. for $1 \leq i,k \leq n$ pardo
   $C'[i,k] := A[i,k] \times B[k]$;

$T(n) = O(1)$, $W(n) = O(n^2)$

2. for h=1 to log n do
   for $1 \leq i \leq n, 1 \leq k \leq n/2^h$ pardo
   $C'[i,k] := C'[i,2k-1] + C'[i,2k]$;

$T(n) = O(\log n)$, $W(n) = O(n^2)$

3. for $1 \leq i \leq n$ pardo
   $C[i] := C'[i,1]$;

$T(n) = O(1)$, $W(n) = O(n)$

end

Total: $T(n) = O(\log n)$, $W(n) = O(n^2)$

Matrix vector product c'td

- On a p-PRAM, the time needed by the balanced tree algorithm is

$$T_p(n) = O(W(n)/p + T(n)) = O(n^2/p + \log n)$$

- This implies that for $p = O(n^2/\log n)$ processors
  the algorithm is *cost-optimal*.

Matrix product: WT

input: A_{nxn}, B_{nxn}, $n = 2^k$;

output: $C_{nxn} = A \times B$

Remember:
in C = A.B every element C[i,j] is
an inner product A[i]T.B[j]

begin

1. for $1 \le i,j,k \le n$ pardo

   C'[i,j,k] := A[i,k] x B[k,j]

$T(n) = O(1)$, $W(n) = O(n^3)$

2. for h=1 to log n do

   for $1 \le i,j \le n$, $1 \le k \le n/2^h$ pardo

   C'[i,j,k] := C'[i,j,2k-1] + C'[i,j,2k]

$T(n) = O(\log n)$, $W(n) = O(n^3)$

3. for $1 \le i,j \le n$ pardo

   C[i,j] := C'[i,j,1]

$T(n) = O(1)$, $W(n) = O(n^2)$

end

Total: $T(n) = O(\log n)$, $W(n) = O(n^3)$

Matrix product: analysis

- Since $T(n) = O(\log n)$, $W(n) = O(n^3)$, for p processors we have

$$T_P(n) = O(n^3/p + \log n)$$

- This implies cost-optimality for $p = O(n^3/\log n)$ processors

Plan of Lecture 2

- Exercises
  - proof of correctness program cumulative frequencies
  - construction program for detecting weakly increasing runs
- Prefix computations and balanced trees
  - inner product
  - matrix multiplication
- Pointer jumping
  - searching for the root of a tree,
  - determining the distance to the root
- Divide & Conquer
  - finding the minimum 1-index in an array
  - finding the maximum of an array

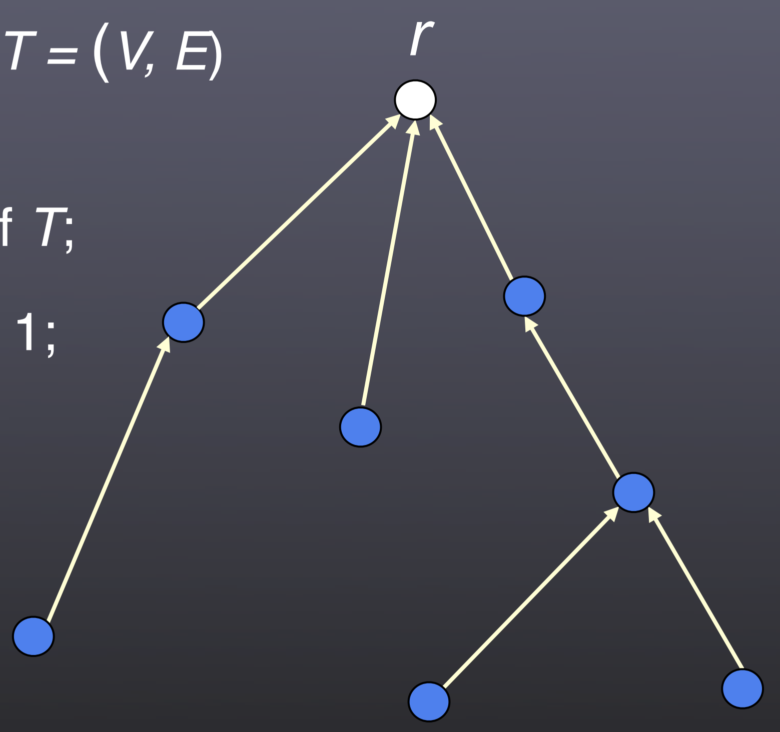
Pointer jumping

- pointer jumping is a technique suitable for fast access in pointer accessible data structures having the form of a forest of *rooted-directed trees.*

Pointer jumping

- pointer jumping is a technique suitable for fast access in pointer accessible data structures having the form of a forest of *rooted-directed trees.*

- a rooted-directed tree is a directed graph $T = (V, E)$ where
  - V contains a special node r, the root of T;
  - every node $v \in V - \{r\}$ has out degree 1; (r has out degree 0)
  - for every $v \in V - \{r\}$ there is a *unique* path from v to r



- a *forest* is just a set of trees

Pointer jumping: problem statement

- Given:
  - a forest *F* = (*V*, *E*) where *V* ={1, . . ., n}.
    F is represented as an array P[1 . . n]
    with P[i]= j iff (i , j) ∈ *E*, i.e., j is parent of i in a tree of F.
- Question:
  - Find the array S[1..n] such that for every j , 1 ≤ j ≤ n ,
    S[j] is the root of the tree containing j .
    The algorithm should have minimal run time T(n).

Sequential strategy:
Reverse pointers in every tree and perform depth first search. Cost: O(n).

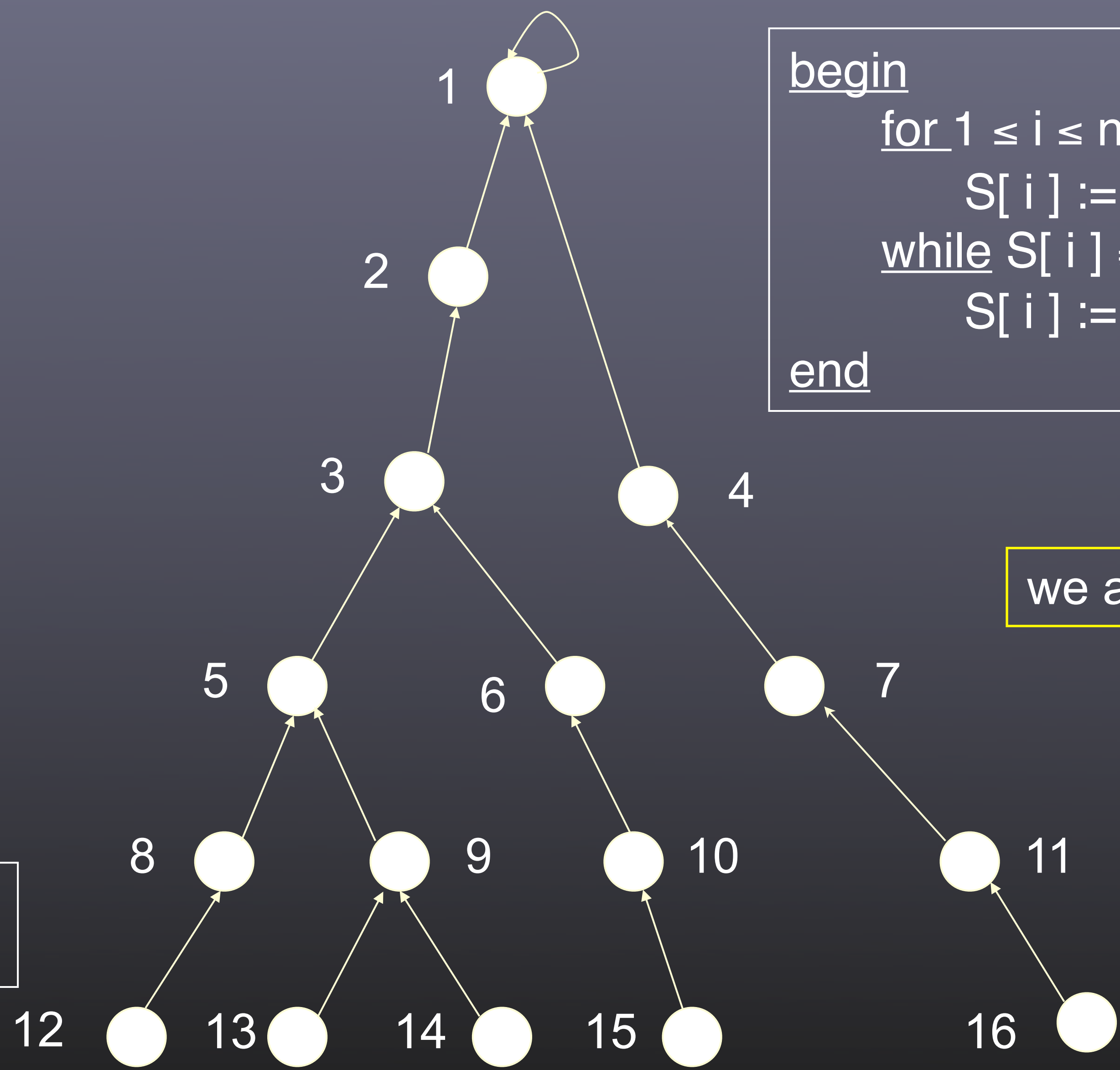
Parallel strategy:
pointer jumping (path doubling): for each node, update the successor of that node by that successor's successor.
Cost: T(n) = O(log h), h = height of the tree

Pointer jumping: example

S = P

1	1
2	1
3	2
4	1
5	3
6	3
7	4
8	5
9	5
10	6
11	7
12	8
13	9
14	9
15	10
16	11

node - parent
table P



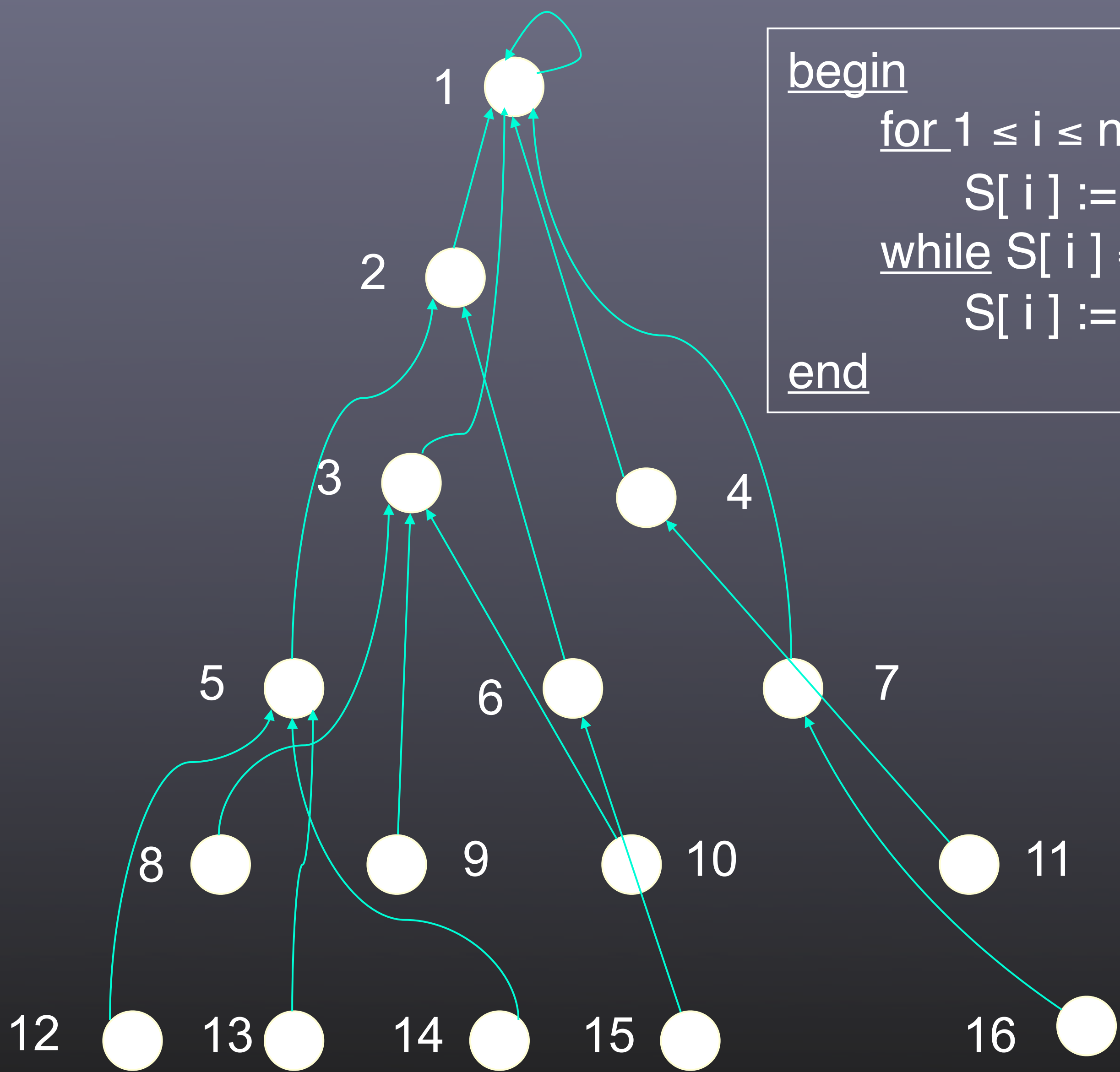

```
begin
    for 1 ≤ i ≤ n pardo
        S[ i ] := P[i];
    while S[ i ] ≠ S[ S [ i ] ] do
        S[ i ] := S[ S [ i ] ];
end
```

we assume P[r] = r

Pointer jumping: example

S

1	1	1
2	1	1
3	2	1
4	1	1
5	3	2
6	3	2
7	4	1
8	5	3
9	5	3
10	6	3
11	7	4
12	8	5
13	9	5
14	9	5
15	10	6
16	11	7

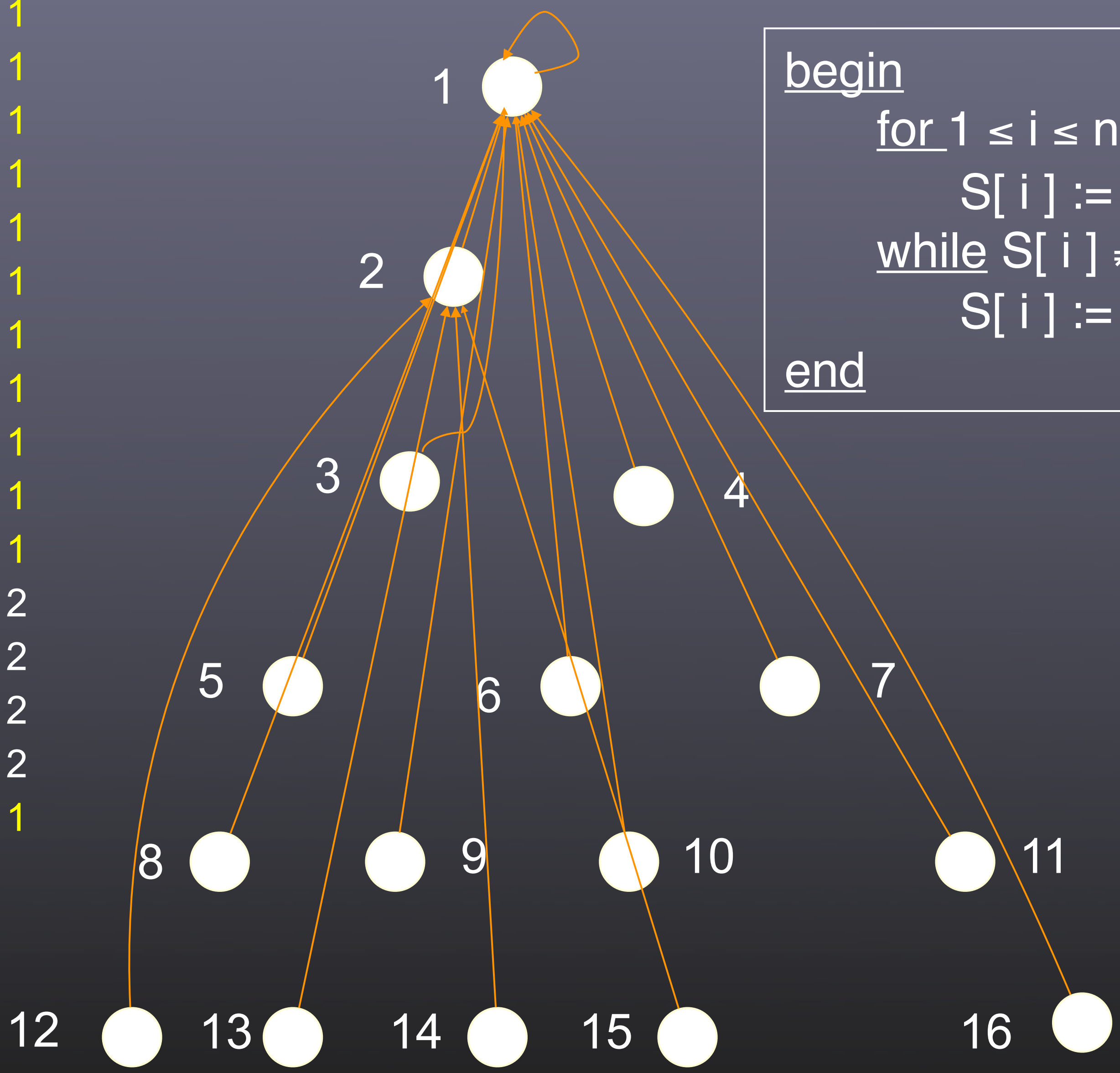



```
begin
    for 1 ≤ i ≤ n pardo
        S[ i ] := P[i];
    while S[ i ] ≠ S[ S [ i ] ] do
        S[ i ] := S[ S [ i ] ];
end
```

Pointer jumping: example

S

1	1	1	1
2	1	1	1
3	2	1	1
4	1	1	1
5	3	2	1
6	3	2	1
7	4	1	1
8	5	3	1
9	5	3	1
10	6	3	1
11	7	4	1
12	8	5	2
13	9	5	2
14	9	5	2
15	10	6	2
16	11	7	1

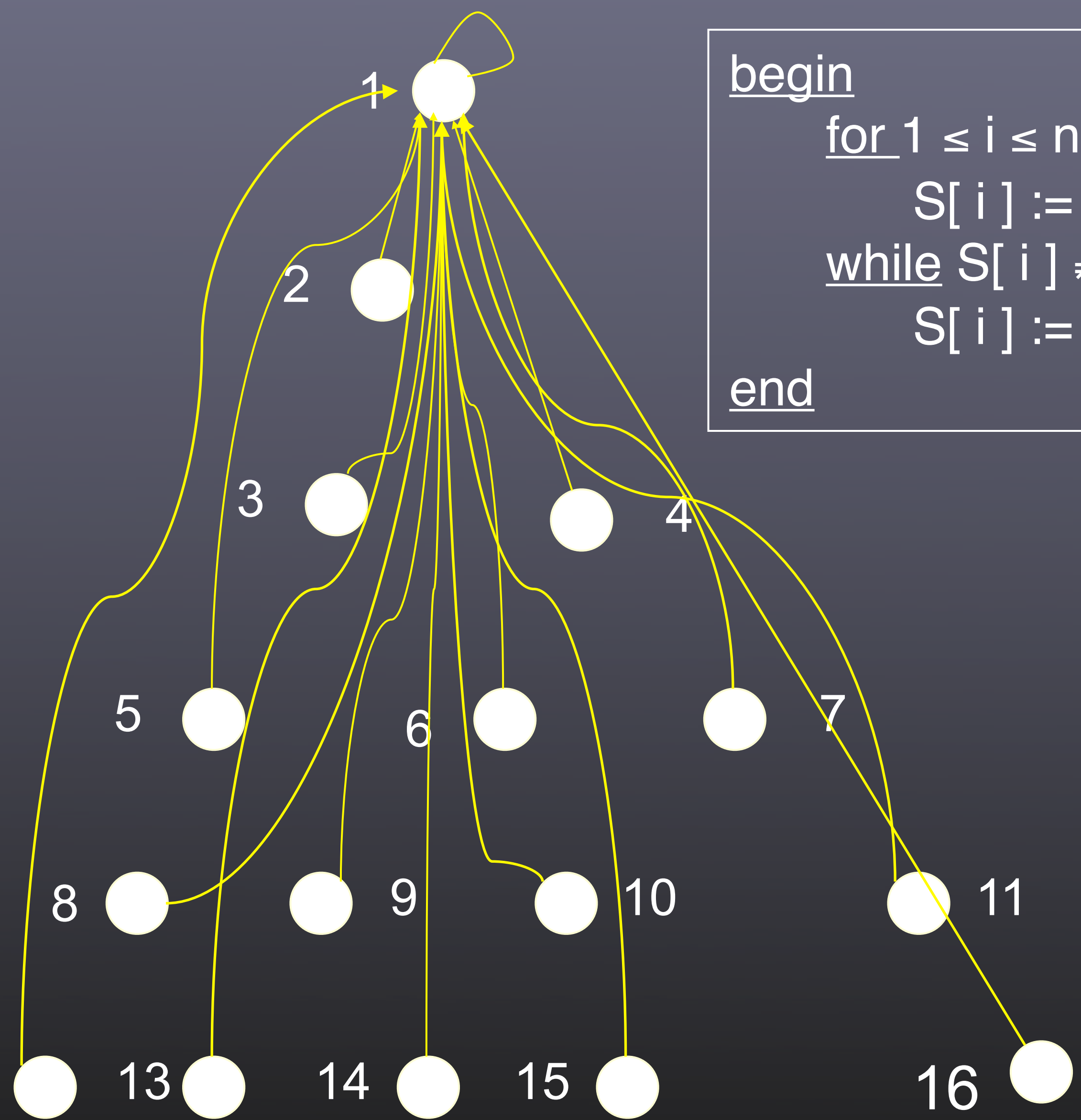



```
begin
    for 1 ≤ i ≤ n pardo
        S[ i ] := P[i];
    while S[ i ] ≠ S[ S [ i ] ] do
        S[ i ] := S[ S [ i ] ];
end
```

Pointer jumping: example

S

1	1	1	1	1
2	1	1	1	1
3	2	1	1	1
4	1	1	1	1
5	3	2	1	1
6	3	2	1	1
7	4	1	1	1
8	5	3	1	1
9	5	3	1	1
10	6	3	1	1
11	7	4	1	1
12	8	5	2	1
13	9	5	2	1
14	9	5	2	1
15	10	6	2	1
16	11	7	1	1




```
begin
    for 1 ≤ i ≤ n pardo
        S[ i ] := P[i];
    while S[ i ] ≠ S[ S [ i ] ] do
        S[ i ] := S[ S [ i ] ];
end
```

Pointer jumping: correctness

input: array P such that (i, P[i]) represents edge in E
output: array S with S[i] the root of the tree of node i

<u>begin</u>

<u>for</u> $1 \le i \le n$ <u>pardo</u>

S[i] := P[i];

$\boxed{P_0 := S;\ j := 0}$ % dummy statement helping in the proof

<u>while</u> S[i] ≠ S[S [i]] <u>do</u>

S[i] := S[S [i]];

$\boxed{j := j+1;\ P_j := S;}$ % dummy statement helping in the proof

<u>end</u>

invariant: *$P_k[i] = j$ iff there is a path p from node i to node j and ((j is not root of i and length of p is 2^k) or (j is root of i and length of p $\le 2^k$)).*

Pointer jumping: correctness

Since F is a forest, for every node i, there is a unique finite path of length ≤ *h* from i to its root r, where *h* = max {height of tree in F}.

Consider the following invariant :

$P_k[i] = j$ iff there is a path from node i to node j and ((j is not root of i and length of path is 2^k) or (j is root of i and length of path ≤ 2^k)) .

This invariant can be proven correct by induction over k = 0,1, ..., log *h*.
It follows that after *k= log h* steps, for every node *i*, we must have $P_k[i]=P_{k+1}[i]$, that is, $P_k[i]$ contains the root of *i*.

Therefore, for *k= log h*, for each *i*, $S[i] = P_k[i]$ will contain the root of *i*.

Pointer jumping: analysis

```
begin
    for 1 ≤ i ≤ n pardo
        S[ i ] := P[ i ];
        while S[ i ] ≠ S[ S [ i ] ] do
            S[ i ] := S[ S [ i ] ];
end
```

implementable on CREW PRAM

Time of algorithm determined by number of iteration steps.
After each iteration step, the distance of node i to node S(i) doubles.
Therefore, we need ≤ log *h* iterations before S(i) = r.
So $T(n) = O(\log h)$.

Every iteration costs $O(n)$ operations.

So $W(n) = O(n \log h)$.

This means that the algorithm is not weakly optimal!

Pointer jumping: other example

Find an algorithm to determine the distances D[i] from node i to the root of the tree:

```
begin
  for 1 ≤ i ≤ n pardo
        S[i] := P[i];
        if i ≠ S[i] then D[i] := 1 else D[i] := 0;
        while S[i] ≠ S[S[i]] do
              D[i] := D[i] + D[S[i]];
              S[i] := S[S[i]];
end
```

$T(n) = O(\log h)$
$W(n) = O(n \log h)$

Exercise: use this algorithm to compute the height of a tree.

Special case: linked list

If each tree is just a path represented by a linked list and each node has a weight W[i], we can use pointer jumping to to solve the *parallel prefix* problem.

That is, by pointer jumping, we can compute the sum of the weights on the path from node i to its root in $O(\log h)$, $O(n \log h)$ where h is the length of the longest list.

Plan of Lecture 2

- Exercises
  - proof of correctness program cumulative frequencies
  - construction program for detecting weakly increasing runs
- Prefix computations and balanced trees
  - inner product
  - matrix multiplication
- Pointer jumping
  - searching for the root of a tree,
  - determining the distance to the root
- Divide & Conquer
  - finding the minimum 1-index in an array
  - finding the maximum of an array

Divide and Conquer

Basic idea

1. Split problem in nearly equal parts;
2. Solve sub problems concurrently, possibly recursively or iteratively,
3. Combine solutions of sub problems to solution of the whole problem

Examples of sequential algorithms

binary search;

mergesort;

quicksort

Example mergesort

```
mergesort (A , n)
  if n=1 return A[1]
  else
    B := mergesort(A[1…n/2], n/2);
    C := mergesort(A[n/2+1 .. n], n/2);
    return merge(B, n/2, C, n/2)
```

split and solve separately

combine

```
merge( X, m, Y, n)
   if m = 1 and n=1 then return min(X[1], Y[1]) ++ max(X[1],Y[1])
   else
      if X[1] ≤ Y[1] then return X[1]++merge(X[2..m],m-1,Y,n
      else
          return Y[1]++merge(X,m,Y[2..n-1], n-1)
```

Computing the maximum of an array

Last Lecture's Exercise

Using a CRCW PRAM, compute the maximum of n elements in an array A using an $(O(n^{1+c}), O(1))$ algorithm, where c is an arbitrary positive constant.

Remark:

We already know an $(O(n^{1+c}), O(1))$ algorithm for $c = 1$!

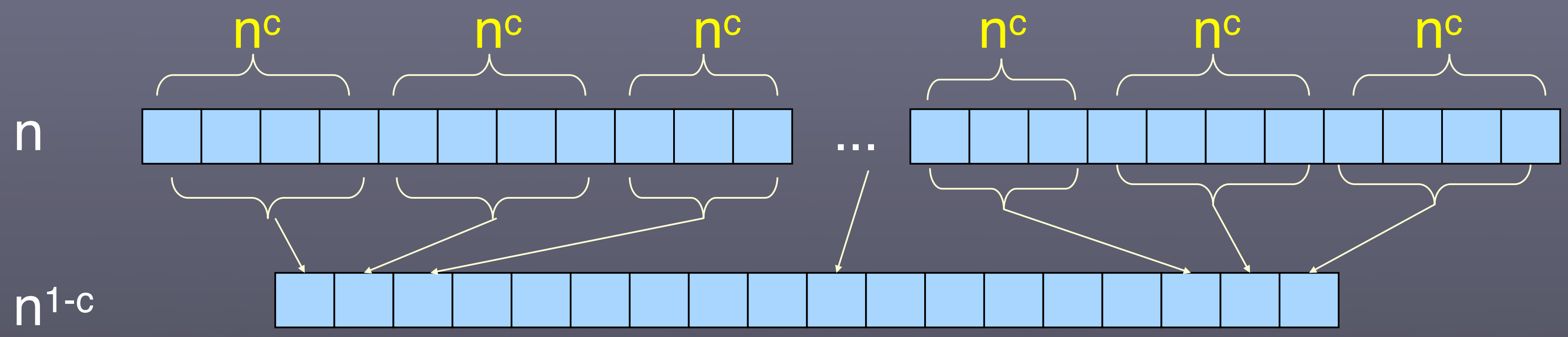
So it is sufficient to look at values $c < 1$.

Solution: the idea

Step 1.

Divide the array of n elements in n^{1-c} blocks of n^c elements.
Compute the n^{1-c} maxima of these blocks in parallel using
the $(O(n^2), O(1))$ CRCW-algorithm for computing the maximum.

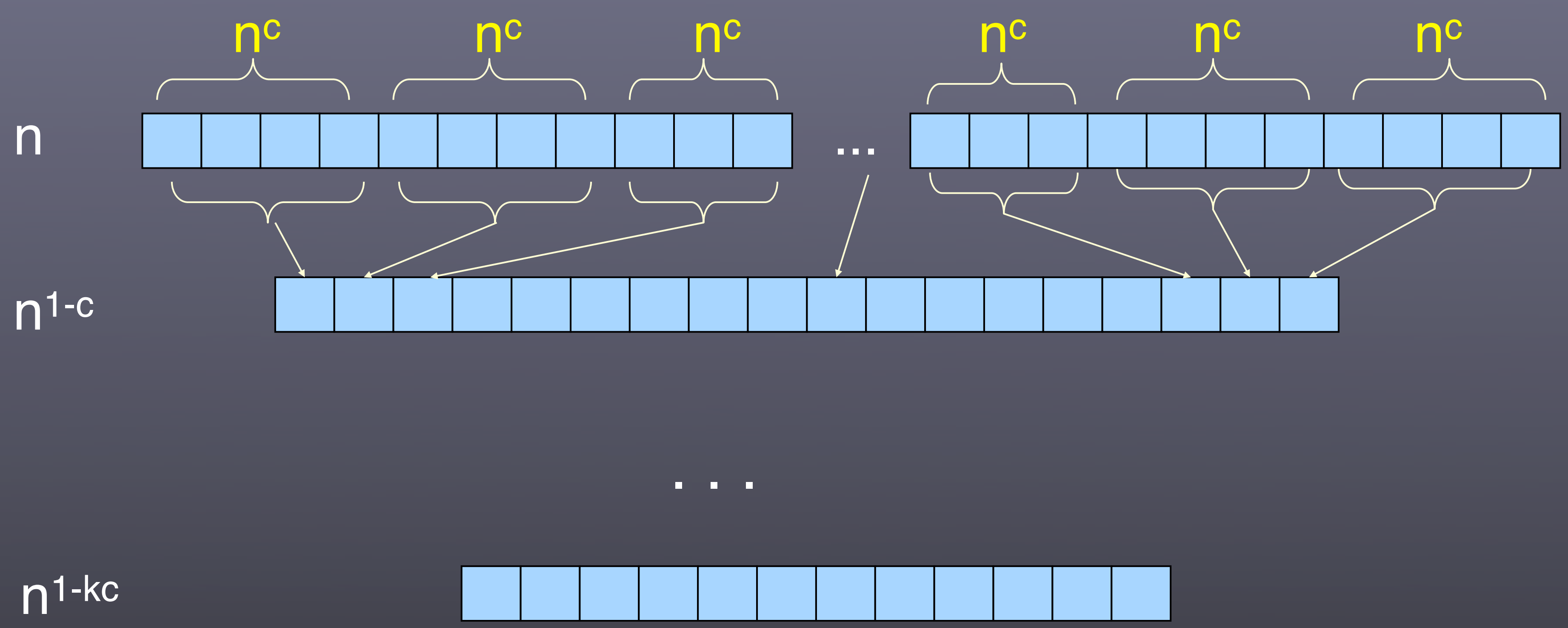
Solution: the idea



Step 1.

Divide the array of n elements in n^{1-c} blocks of n^c elements.
Compute the n^{1-c} maxima of these blocks in parallel using the $(O(n^2), O(1))$ CRCW-algorithm for computing the maximum.

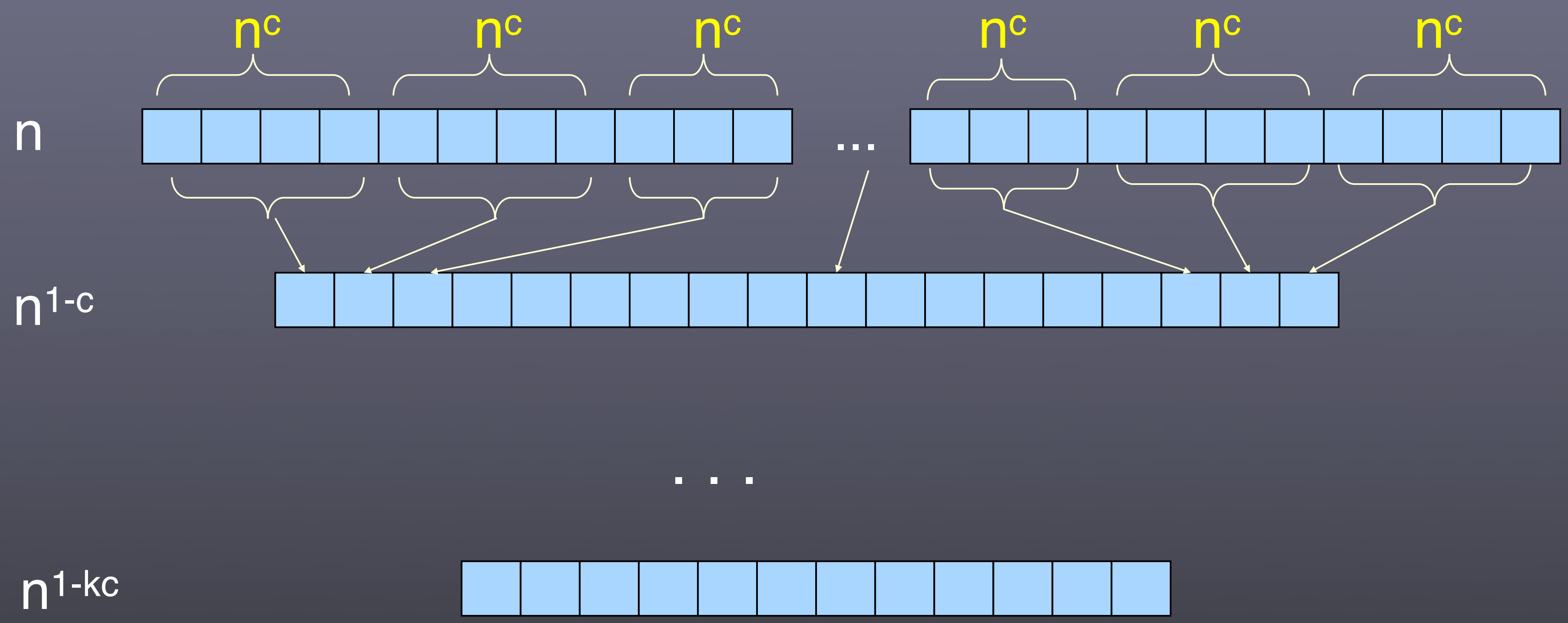
Solution: the idea



Step 2.

Iteratively applying the procedure in Step 1, compute the maxima on the resulting arrays with n^{1-c}, $n^{1-c}/n^c = n^{1-2c}$, $n^{1-2c}/n^c = n^{1-3c}$, …, n^{1-kc} elements containing maxima until $k = \lceil(1-c)/2c\rceil$

Solution: the idea

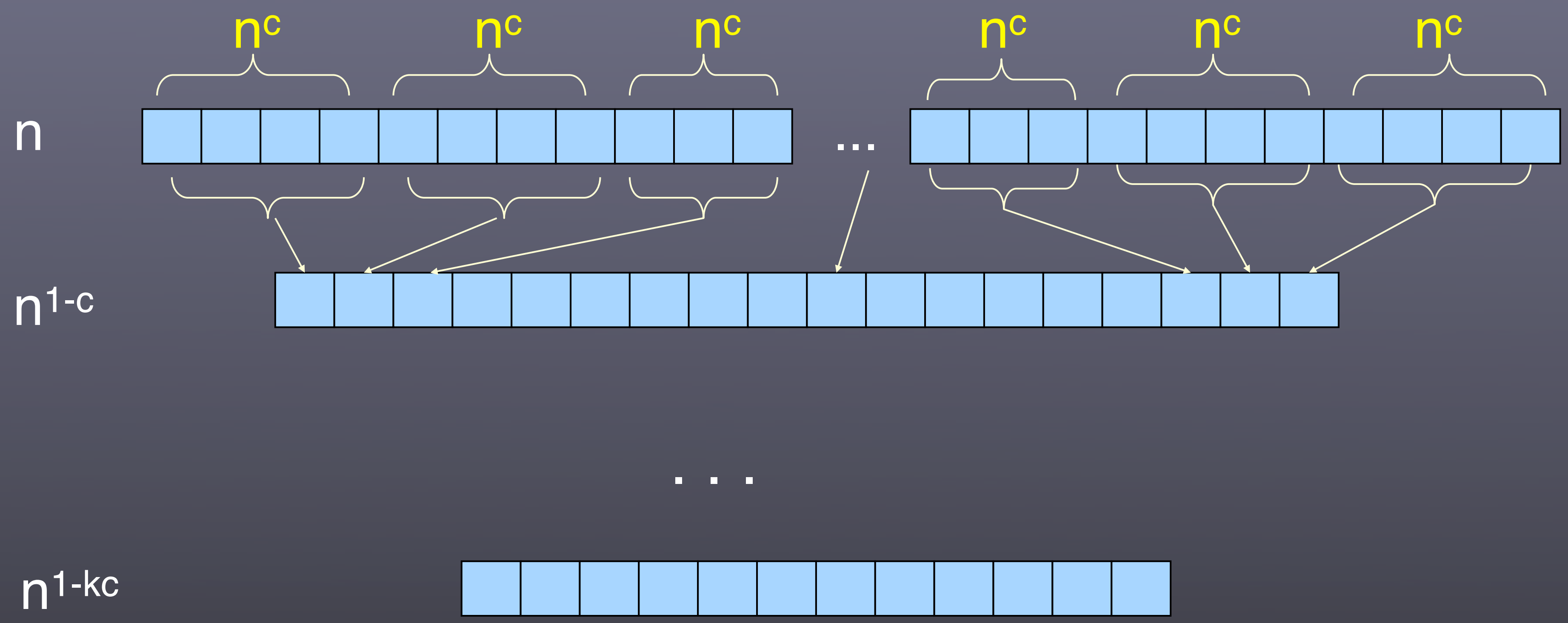


Step 2.

Iteratively applying the procedure in Step 1, compute the maxima on the resulting arrays with n^{1-c}, $n^{1-c}/n^c = n^{1-2c}$, $n^{1-2c}/n^c = n^{1-3c}$, …, n^{1-kc} elements containing maxima until $k = \lceil (1-c)/2c \rceil$

NB: if $k = \lceil (1-c)/2c \rceil$ then
$(n^{1-kc})^2 = n^{2-2kc} \leq n^{1+c}$

Solution: the idea

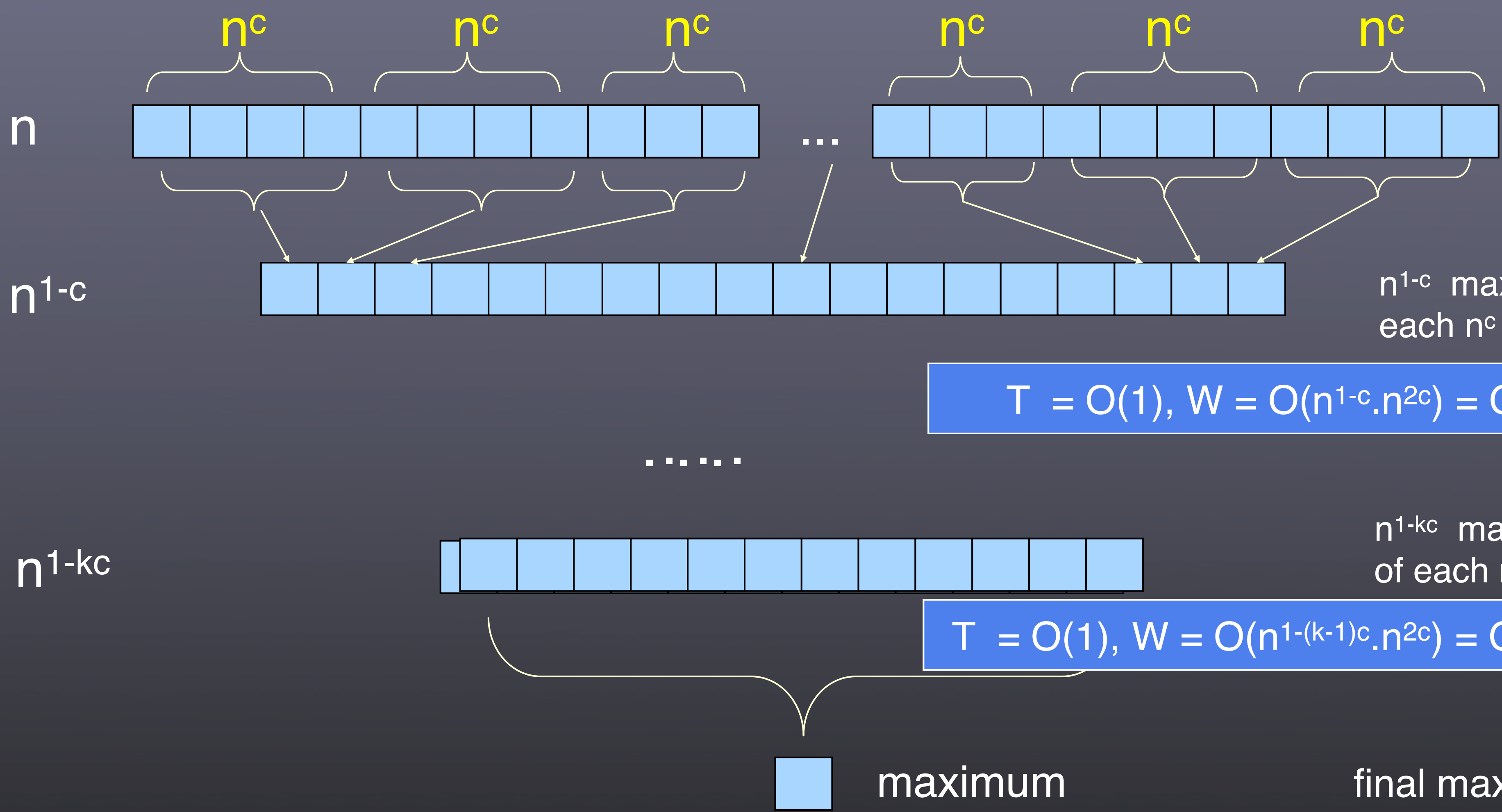


Step 3.

Compute the final maximum for the array containing $\leq n^{1-kc}$ maxima using the $(O(n^2), O(1))$ - algorithm.

Notice that k is a constant depending upon c !

Solution: analysis



n^{1-c} maxima of each n^c -block

T = O(1), W = $O(n^{1-c}.n^{2c}) = O(n^{1+c})$

n^{1-kc} maxima of each n^c blok

T = O(1), W = $O(n^{1-(k-1)c}.n^{2c}) = O(n^{1+c})$

final maximum

T = O(1), W = $O(n^{(1-kc)2}) = O(n^{1+c})$

Total: $T(n) = (k+1) \times O(1) = O(1)$; $W(n) = (k+1) \times O(n^{1+c}) = O(n^{1+c})$

Summary of the d&c algorithm

We may assume $c < 1$ (why?).

1. Partition(iteratively) the array in n^{1-c} parts of size n^c.
   Apply with the quadratic work-constant time the CRCW PRAM algorithm on every subarray of size n^c and store the result in an array of maxima of size n^{1-c}.

2. Apply step 1 iteratively on the resulting array. The k-th iteration is applied on an array of size $n^{1-(k-1)c}$ en results in an array of size n^{1-kc}. Every iteration costs $T = O(1)$ time and $W = n^{1-kc} \times O(n^{2c}) = O(n^{1+(2-k)c}) \leq O(n^{1+c})$ work.

3. After $k \geq \lceil 1/2c - 0.5 \rceil$ iterations we have $(n^{1-kc})^2 \leq n^{1+c}$.
   Hereafter the CRCW-PRAM algorithm can be finally applied on the resulting array with n^{1-kc} maxima to compute the global maximum. The costs are: $T = O(1)$, $W = O(n^{1+c})$.

   The total costs are: $T(n) = (k+1)\, O(1) = O(1)$,
   $W(n) = (k+1)\, O(n^{1+c}) = O(n^{1+c})$.

New exercise:

Problem: min-1 index

- Given
  a boolean array A[1..n]

- Question
  find an $W(n)= O(n)$, $T(n) = O(1)$ algorithm on a CRCW-PRAM to compute the first 1 in A,
  that is, compute *the smallest index* k such that $A[k] = 1$.

See you next week