Parallel Algorithms and Parallel computers (ii)

IN4 026

Lecture 2

Cees Witteveen
Parallel and Distributed Systems Group
Faculty Electrical Engineering, Mathematics and Computer Science

---

Basic techniques and Examples

- Balanced trees
  - cumulative frequencies (recursive & iterative)
  - inner product, matrix multiplication
- Pointer jumping
  - searching for the root of a tree,
  - determining the distance to root
- Divide & conquer
  - finding the minimum 1-index in an array
  - finding the maximum of an array

exercises of last week

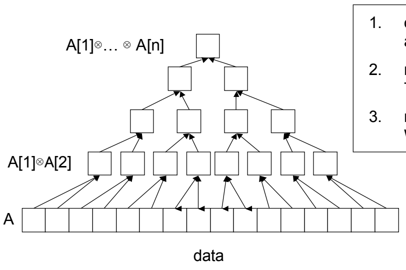
---

1. Balanced Trees

---

balanced trees

- principle
  construct a (virtual) *balanced binary tree* to process input elements and traverse the tree to perform operations on the nodes.



1. operations at each level are performed concurrently
2. number of time steps is T(n) = O(height) = O(log n)
3. number of operations is W(n) = O(n)

tree results are well-defined for binary left-associative operators ⊗

balanced trees: example

- Given an array A[1. . n] of frequencies, compute array C[1 . . n] of cumulative frequencies, i.e. $C[\,i\,] = \Sigma_{j \le i} A[\,j\,]$.

proof of correctness: see lecture

```
cumfreq ( A , n ):
begin
    1. if n=1 then  { C[ 1 ]:= A[ 1 ];  exit }                    T = O(1)
    2. for 1 ≤ i ≤ n/2  pardo                                      T = O(1)
               B [ i ] := A [ 2i-1] + A[ 2i ]
    3. Z := cumfreq ( B , n/2 )                                    O(T(n/2))
    4. for 1 ≤ i ≤ n   pardo
           { i = 0 mod 2  => C [ i ]  = Z[ i/2 ],                  T = O(1)
             i = 1          => C [ 1 ] = A[1],
             else           => C [ i ]   = Z[ (i-1)/2 ] + A(i) }
end
```

cumulative freq: WT-analysis

- T(n) = O(1) + O(1) + T(n/2) + O(1) = T(n/2) + O(1)
  = O(log n)
- W(n) = O(1) + O(n)+ W(n/2) + O(n) = W(n/2) + O(n)
  = O(n)
- Conclusions:
  - algorithm is (weakly) optimal
  - algorithm is cost-optimal for p = O(n/log n) on a p-PRAM

Cumulative frequencies: iterative

input: array A[1 . . n] $n = 2^k$
output: array $C_{\log n \times n}$ with $C[\,0\,,\,j\,] = \Sigma_{j \le i} A[\,j\,]$.

This algorithm is the unfolding of the previously given recursive specification of the balanced tree technique

```
cumfreq_iter( A , n ):
begin
  1. for 1 ≤ j ≤ n pardo B[0,j] : = A[j]

  2. for h = 1 to  log n do
         for 1 ≤  j ≤  n/2^h pardo
           B[h,j] := B[h - 1, 2j -1] + B[h - 1, 2j]

  3. for  h = log n  to  0  do
          for 1 ≤  j ≤  n/2^h pardo
            { j  even => C[ h,j ]  = C[ h+1, j/2 ]
              j = 1     => C[ h,1]  = B[ h ,1 ]
              else      => C[ h,j ]  = C[ h+1, (j-1)/2] + B[h,j] }
end
```

General Comments

- Given an array A[1. . n] and any associative operator ∗, the balanced tree scheme can be used to compute the array
  C[1 . . n] of "prefix sums" where
  C[i] = A[1] ∗ A[2] ∗ … ∗A[i].
- The same technique can be used for
  - broadcasting a value to all memories of processors
  - compacting a labeled array
  - inner product computations

Balanced Trees and Matrix operations

Topics

- inner products and balanced trees
- matrix vector product: a WT-analysis
- matrix multiplication

Balanced trees and inner products

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

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

  The inner product can be computed by an
  (O(n), O(log n))-algorithm using a simplified balanced tree method

```
input:   U[1..n], V[1..n]  where $n = 2^k$;  output: $U^TV$
begin
1. for $1 \le i \le n$ pardo
         C[i] = U[i] x V[i]
2. for h=1 to log n do
         for $1 \le k \le n/2^h$ 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 x B
begin
   1. for $1 \le i,k \le n$ pardo
         C'[i,k] = A[i,k] x B[k]
   2. for h=1 to log n do
         for $1 \le i \le n$, $1 \le k \le n/2^h$ pardo
                C'[i,k] = C'[i,2k-1] + C'[i,2k]
   3. for $1 \le i \le n$ pardo
                C[i] = C'[i,1]
end
```

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

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

Step 3: T(n) = O(1), W(n) = O(n)

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:        $C_{nxn} = A \times B$
begin
  1. for $1 \le i,j,k \le n$ pardo                      T(n) = O(1), W(n) = O(n³)
       $C'[i,j,k] = A[i,k] \times B[k,j]$
  2. for h=1 to log n do                           T(n) = O(log n),W(n) = O(n³)
       for $1 \le i,j \le n$, $1 \le k \le n/2^h$ pardo
            $C'[i,j,k] = C'[i,j,2k-1] + C'[i,j,2k]$
  3. for $1 \le i,j \le n$ pardo                         T(n) = O(1), W(n) = O(n²)
            $C[i,j] = C'[i,j,1]$
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

Relations with Grama

Consult Grama et al, Chapter 8 for details concerning the influence of communication and task allocation on the performance.

Compare the results presented here with the results obtained in section 8.1.1 and 8.1.2 of Grama.

Note that for a row-wise 1-D partitioning applied to matrix-vector multiplication and matrix multiplication, the balanced tree method cannot profit from concurrency.

Note that in general Grama et al do not make a distinction between the architecture-free properties of the algorithm and the implementation details.

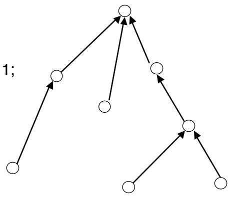
2. Pointer Jumping

Pointer jumping

- pointer jumping is a technique suitable for fast access in pointer accessible *rooted-tree* or *-forest* like data structures

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

- a forest is a set of trees



Pointer jumping: example

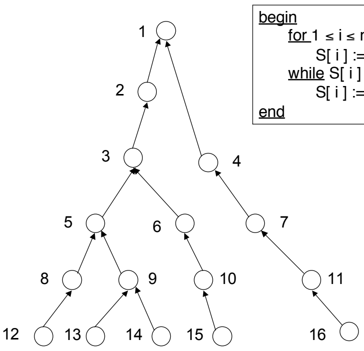
- Given:
  - a forest F = (V, E) where V ={1, . . n}. F is represented as an array P[1 . . . n] with P[i]= j iff (i , j) ∈ E, i.e. j is parent of i in a tree of F.

- Question:
  - for every j , 1 ≤ j ≤ n , find the root S[j] in the tree containing j.

Pointer jumping: example

S

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

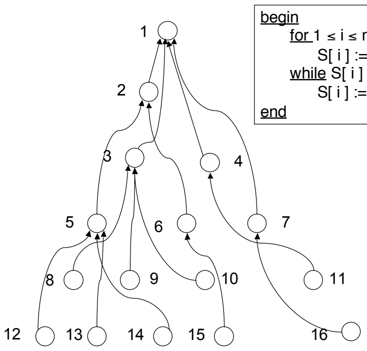


```
begin
      for 1 ≤ i ≤ n pardo
            S[ i ] := P[i];
      while S[ i ] ≠ S[ S [ i ] ] do
            S[ i ] := S[ S [ i ] ];
end
```

Pointer jumping: example

S

1	1	1
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

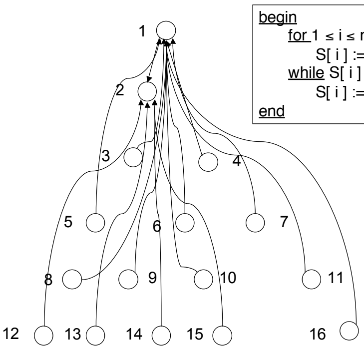


```
begin
      for 1 ≤ i ≤ n pardo
            S[ i ] := P[i];
      while S[ i ] ≠ S[ S [ i ] ] do
            S[ i ] := S[ S [ i ] ];
end
```

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

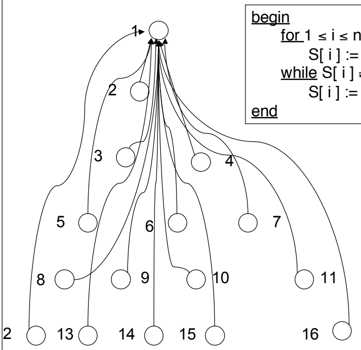


```
begin
    for 1 ≤ i ≤ n pardo
        S[ i ] := P[i];
    while S[ i ] ≠ S[ S [ i ] ] do
        S[ i ] := S[ S [ i ] ];
end
```

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: algorithm

– Assume that for every root r we have P[r] = r.

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

```
begin                                   P_0 := S;  j := 0
    for 1 ≤ i ≤ n pardo
        S[ i ] := P[ i ];
        while S[ i ] ≠ S[ S [ i ] ] do
            S[ i ] := S[ S [ i ] ];
end                                     j := j+1; P_j := S;
```

Correctness of the algorithm: 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))* . 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, at the end, 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
```

The algorithm is not weakly optimal!

- after each iteration the distance of node i to the node S(i) doubles
  => we need ≤ log h iterations before S(i) = r
  => T(n) = O(log h)
- every iteration costs O(n) operations
  => W(n) = O(n log h)

POINTER JUMPING (2)

- An algorithm to determine distances D[i] from node i to the root of the tree:

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

```
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
```

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

---

3. Divide and Conquer

Divide and Conquer

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

sequential examples:
binary search;
quicksort

Divide and Conquer: example

Problem: min-1 index

- Given
  a boolean array A[1..n]
- Question:
  find an O(n,1) algorithm on a CRCW-PRAM
  to compute the smallest value k such that A[k] = 1.

Method

- First we present an $(O(n^2),O(1))$- algorithm to solve the min-1 index problem by concurrent application of a find-min algorithm to compute the minimum value in an integer array.
- Then we discuss an $(O(n),O(1))$- algorithm for a simpler problem : find-1index: given an array A, does there exist a value k such that A[k]=1.
- Finally we combine both algorithms find-min and find-1index to an $(O(n),O(1))$-algorithm using the divide-and-conquer approach.

Phase (1): find-min

- An $O(n^2)$, $O(1)$-algorithm to determine the minimal value in an *integer* array A[1..n]:

```
findmin(A , n)
begin
    1. for 1 ≤ i, j ≤ n  pardo                          T=O(1),  W = O(n²)
        if A[i] ≤ A[j] then B[i,j] := 1 else B[i,j] := 0

    2. for 1 ≤ i ≤ n pardo                              T=O(1),  W = O(n²)
        M[i] := 1;
        for  1 ≤ j ≤ n pardo if B[i,j]= 0 then M[i]:= 0;
        if M[i] then min := A[i]
    3. return min;                                      T=O(1),  W = O(1)
end
```

Phase (2): first version min-index

input :	boolean arrayA[1..n]
output:	index of first 1 in A, else n +1

```
begin
    var B[1..n] of int
    1. for 1 ≤ i ≤ n pardo
           if A[i] = 1 then B[i] = i
           else B[i] = n+1;
    2. return findmin(B,n)
end
```

Note that this is an $O(n^2)$, $O(1)$ algorithm !

Phase (3): find1-index

- We turn to a related simpler problem:

find-1index:
given a boolean array A[1..n],
is there an index k such that A[k] = 1?

- An $(O(n),O(1))$ CRCW-PRAM algorithm to solve this problem

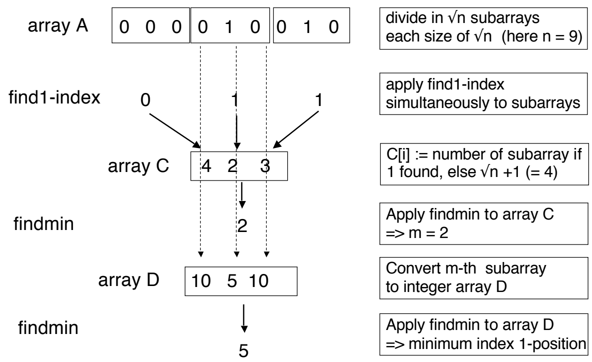
```
find-1index(A)
 begin
    output := 0;
    for 1 ≤ j ≤ n pardo
        if A[j] = 1 then output := 1;
    return output
 end
```

Phase(4) : divide and conquer idea

Combine both algorithms to an (O(n),1) -algorithm using divide and conquer as follows

1. divide array A into $\sqrt{n}$ subarrays with length $\sqrt{n}$
2. apply algorithm find-1index to these subarrays in parallel; this enables us to determine in which of the $\sqrt{n}$-arrays a 1 occurs with cost $T(n) = O(1)$ and $W(n) = O(n)$
3. use the results obtained to construct an array $C[1.. \sqrt{n}]$ such that $C[i] = 1$ iff the i-th subarray contains a 1; costs $T = O(1)$, $W = O(\sqrt{n})$
4. To find the first subarray containing a 1, apply findmin to $C[1 .. \sqrt{n}]$ costs: $T = O(1)$, $W = O(n)$. If findmin returns m, then we look into the subarray $A[(m-1)x\sqrt{n} +1, . . ., mx\sqrt{n}]$.
5. We create an array $D[1 .. \sqrt{n}]$ with $D[j] = (m-1)x\sqrt{n} + j$ if $A[(m-1)x\sqrt{n} + j] = 1$ and $D [j] = n +1$ else; we find the first 1 of A by by applying findmin to D in $T = O(1)$ and $W = O(n)$

Example of application



array A: 0 0 0 | 0 1 0 | 0 1 0

divide in $\sqrt{n}$ subarrays each size of $\sqrt{n}$ (here n = 9)

find1-index: 0 1 1

apply find1-index simultaneously to subarrays

array C: 4 2 3

C[i] := number of subarray if 1 found, else $\sqrt{n}$ +1 (= 4)

findmin: 2

Apply findmin to array C => m = 2

array D: 10 5 10

Convert m-th subarray to integer array D

findmin: 5

Apply findmin to array D => minimum index 1-position

Finding the maximum

Problem:
Given an array A[1..n] such that
for every j=1,...,n, $1 \leq A[j] \leq n$,
find an algorithm to determine $\max_i\{ A[i] \}$ in
$W = O(n)$, $T = O(1)$.

- Solution

```
begin
 1. for 1 ≤ j ≤ n pardo B[A[j]] := 1
 2. max:= findmaxindex(B,n)
 end
```

Exercise

- Given an integer array A[1..n], compute the maximum of n elements using an $(n^{1+c}, O(1))$ algorithm, where c is an arbitrary (small) positive constant.

Please prepare solutions to this exercise for next week lecture