# Parallel programming
# *// programming*

S.I

*Françoise Baude*

Université Côte d'Azur

Polytech Nice Sophia

baude@unice.fr

web site: https://lms.univ-cotedazur.fr/course/view.php?id=17901

(self registration)

March 2023

## Chapter 3: Scan/Prefix

# PREFIX Parallel computation

- A very useful & general pattern named also SCAN
- Almost similar pattern known as SUFFIX
- Principle:
  - A collection of values (x1,x2,….xn)
    - In an array, or a list for instance
  - Compute (y1,y2,…,yn) such that
    yk = x1 *op* x2 *op*… *op* x(k-1) *op* xk
    (in suffix case: yk=xn *op* x(n-1) *op* … *op* xk)
  - OP is a binary associative operation
  - Popular case : OP is a sum
    - *prefix* *<u>sum</u>*
- Goal: Find a parallel time = O(log n) algorithm
  - Ideally with a total work = O(n)
  - Seq computation: O(n)

```
y[0]=x[0]
for i=1 to n-1 do
        y[i]=x[i] + y[i-1]
```

# Parallel prefix (v1)

- Based about a logarithmic depth binary tree traversal
- Along the recursive principle
  - OP(x1,x2,… x (n/2)) accumulates on (x (n/2) …, xn)
  - OP(x1,x2,… x (n/4)) accumulates on (x (n/4)+1 …, x(n/2))
  - OP(x (n/2)+1 …, x(3n/4) accumulates on (x (3n/4)+1 …, xn)

- Code EREW PRAM of Dekel et Sahni, Optimal
  - version of [Desprez], with OP=+
  - Collection size = 2^m, stored in the second half of array A. Result will be available in second half of array B
  - First halves of A & B used as intermediate storage
  - 3 phases:

1. « ascend » along the tree, from bottom to root

```
For l=m-1 to 0 do (in sequential)
    For j=2^l to 2^(l+1) -1 do_in parallel
        A[j]= A[2.j] + A[2.j +1]
    endFor
endFor
```

- Complexity: //time O(log n) = O(m) on EREW, with n/2 proc.
- Can be turned as work optimal

2. « descend » : for each node, accumulate (in B) value from left brother(in A) & value stored at father level (in B)

- If I'm a node at left hand side, just take the value accumulated at my father level (no left brother!)
- If I'm a node at right hand side, combine accumulated value at father and its left hand side node (=my left brother)

```
B[1]=0
For l=1 to m do (in sequential)
    For j=2^l to 2^(l+1) -1 do_in_parallel
        if EVEN(j) // j is a left hand side node,
            B[j] = B[j/2]  // j gets the value accumulated at j' father node
        if ODD(j) // j is a right hand side node
            B[j] = B[(j-1) / 2)] + A[j-1] // combine with + father & left brother values
    endFor
endFor
```

- Complexity: //time O(log n) = O(m) on EREW, with n/2 proc.
  - maxi. 2 concurrent Read access, so EREW is OK
    - eg j=8 and j=9, both access B[4] in read mode
- Can be turned as work optimal

3. Finalize the computation of the second half of B, by adding one by one the values stored in second half of A

- In // accumulate the own value to the incomplete prefix sum value

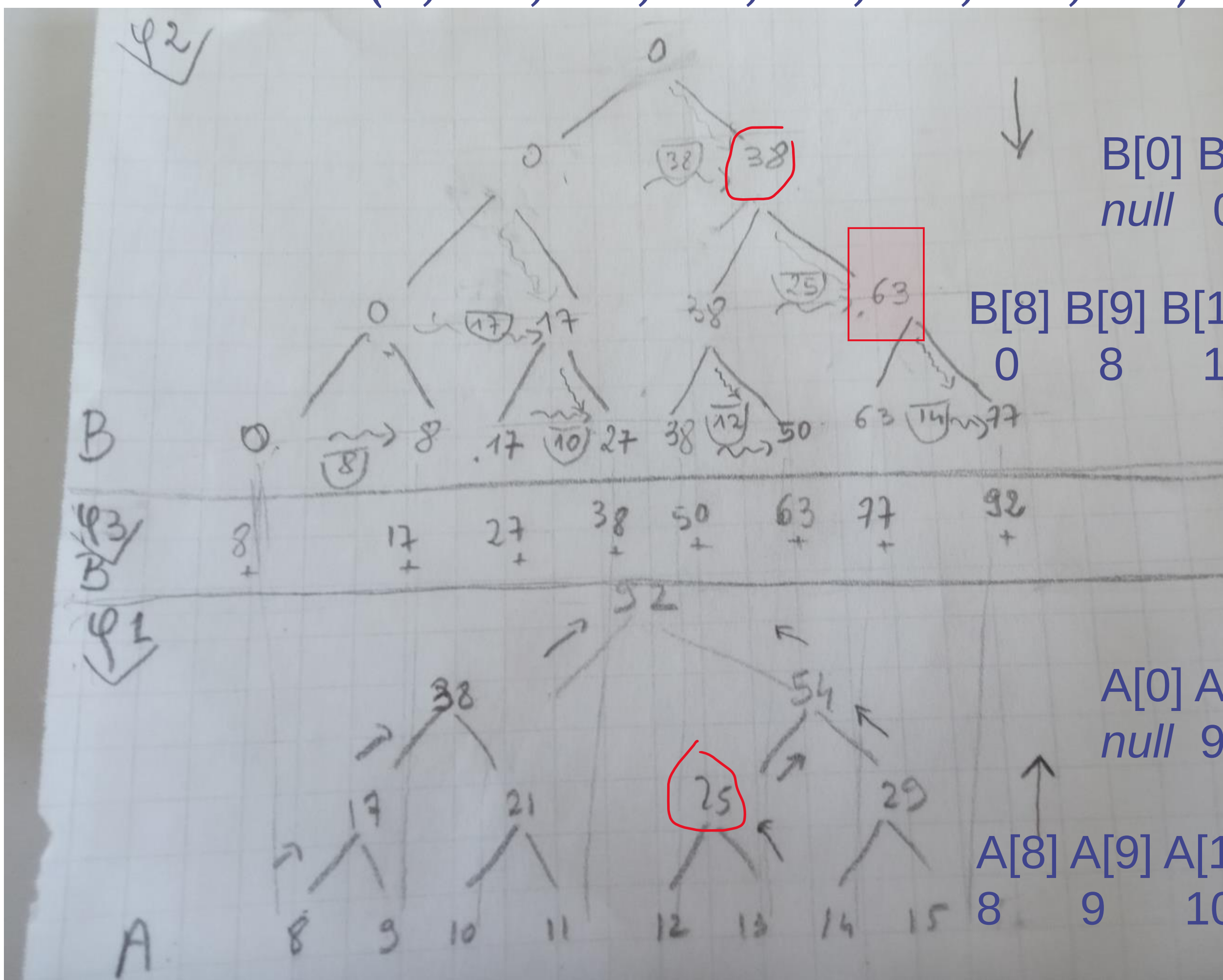
```
For j=2^m to 2^(m+1) - 1 do in parallel
        B[j]= B[j] + A[j]
endFor
```

- Final phase can be avoided if yk=x0 *op* x1 *op* … *op* x(k-1)
- Complexity: //time O(1) on an EREW, with n/2 proc.
- Can be turned as work optimal

TOTAL: O(log n) with O(n) procs on an EREW PRAM,
or O(log (n / log (n)) + log(n)) using O(n/log n) procs

# Simulation of v1 algo

- A=(8, 9, 10, 11, 12, 13, 14, 15)
- Prefix sum : final B[k]=A[0]+…+A[k] for each k
  B=(8, 17, 27, 38, 50, 63, 77, 92)



| B[0] | B[1] | B[2] | B[3] | B[4] | B[5] | B[6] | B[7] |
|---|---|---|---|---|---|---|---|
| *null* | 0 | 0 | 38 | 0 | 17 | 38 | 63 |

| B[8] | B[9] | B[10] | B[11] | B[12] | B[13] | B[14] | B[15] |
|---|---|---|---|---|---|---|---|
| 0 | 8 | 17 | 27 | 38 | 50 | 63 | 77 |

| A[0] | A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] |
|---|---|---|---|---|---|---|---|
| *null* | 92 | 38 | 54 | 17 | 21 | 25 | 29 |

| A[8] | A[9] | A[10] | A[11] | A[12] | A[13] | A[14] | A[15] |
|---|---|---|---|---|---|---|---|
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

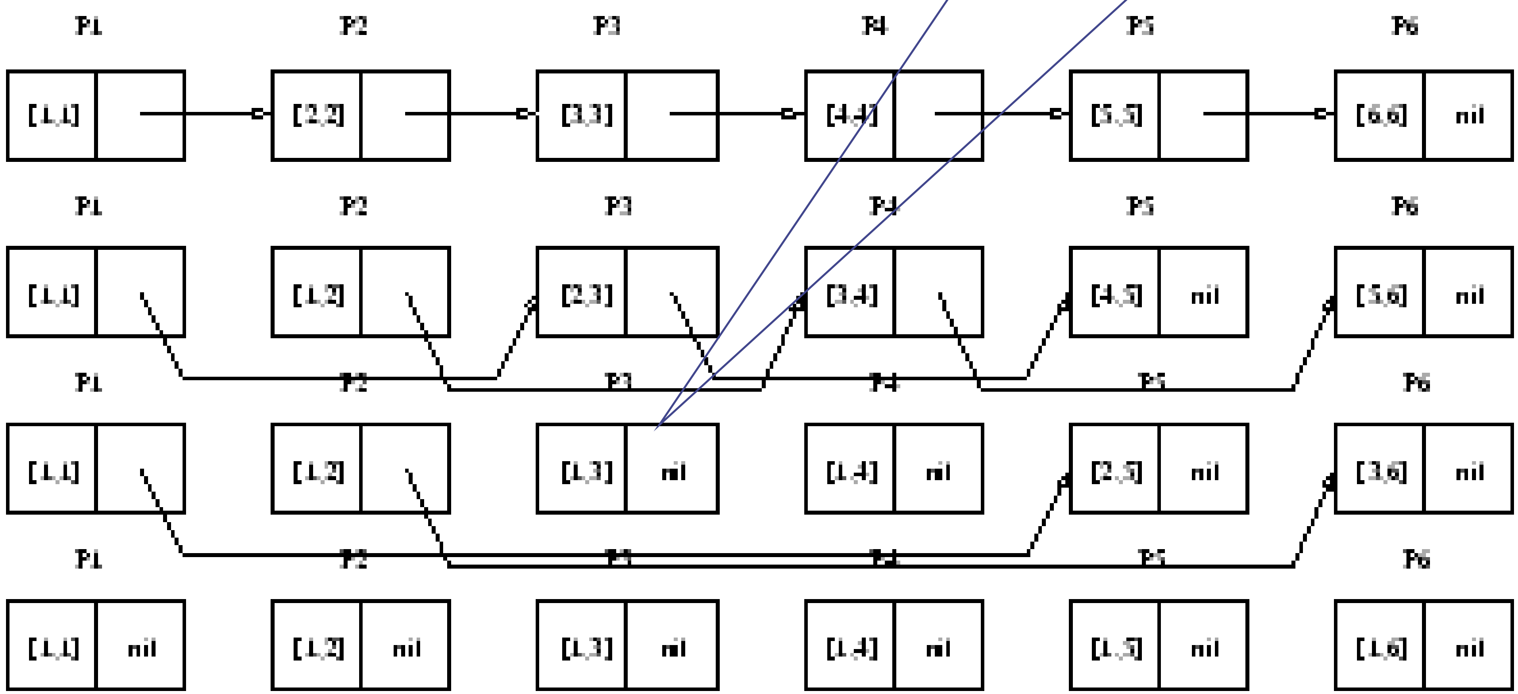
# Parallel prefix (v2)

- Based on a collection : linked list
  - -> none of elements knows its position in the list
  - Visit the list by following pointers between elements: pointer jumping
  - Principle of recursive doubling:
    - For all proc., the distance to which information are propagated is doubling at each parallel step
    - So, the parallel time needed is in O(log of list length) so that all the information gets propagated

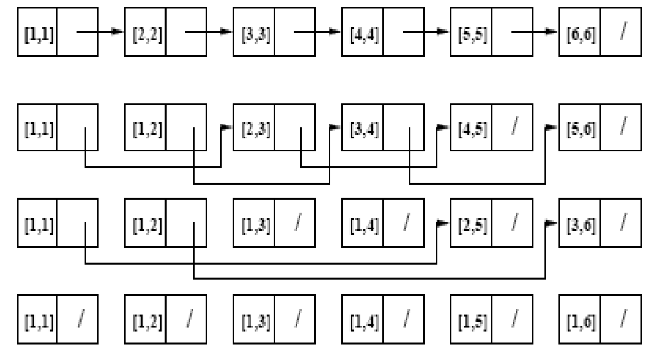
# Principle of version 2

- Code EREW PRAM in O(logn) time, not work optimal

This value is the accumulated result of values of element 1 to 3 included



# Code for v2




$[i,j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$ pour $i \leq j$

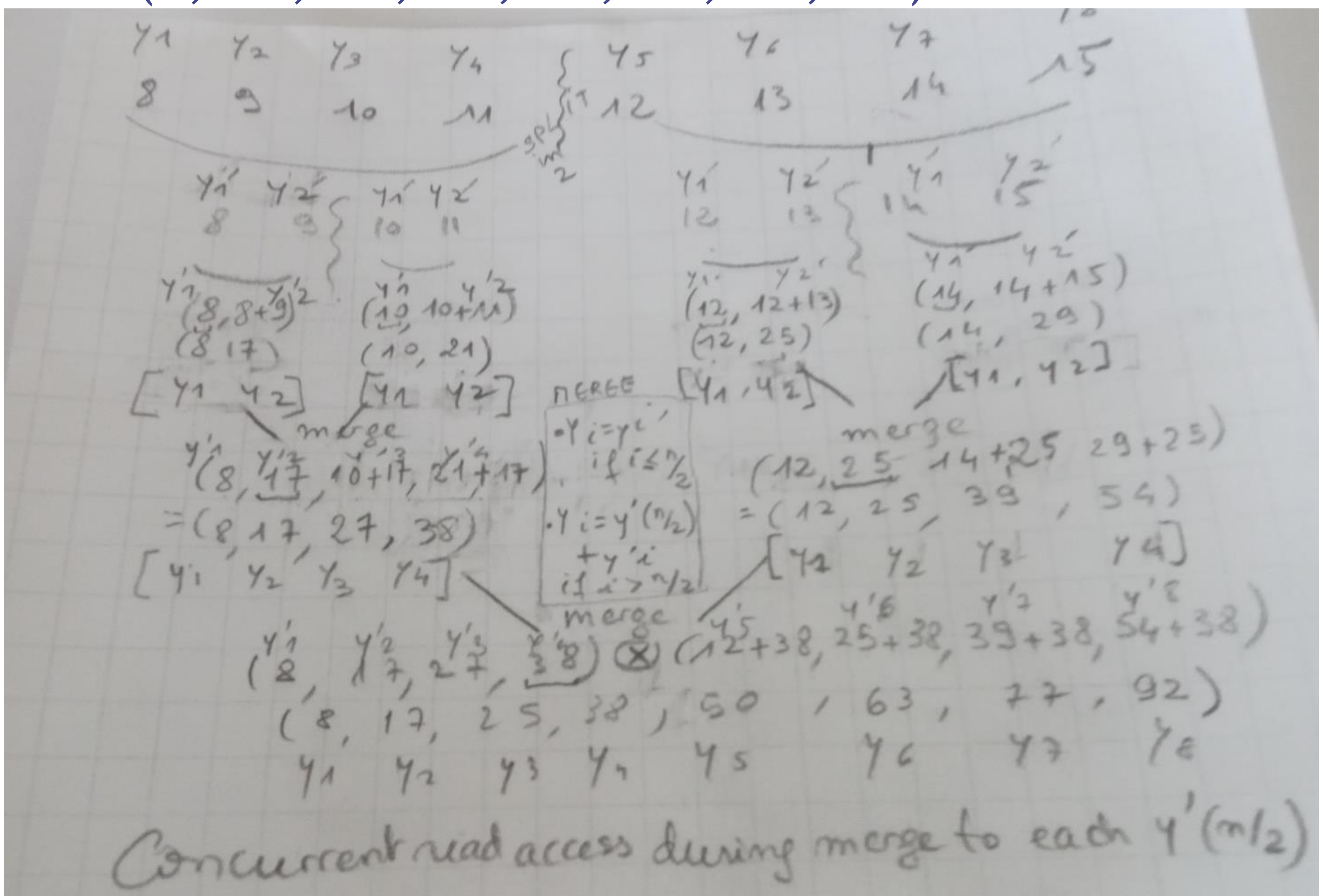
**for** each processor i in // **do**
  $y[i] \leftarrow x[i]$
  **while** ($\exists$ objet $i$ t.q. $next[i] \neq$ NIL) **do**
    **for** each processor i in // **do**
      **if** $next[i] \neq$ NIL **then**
        $y[next[i]] \leftarrow y[i] \otimes y[next[i]]$
        $next[i] \leftarrow next[next[i]]$

# Parallel prefix (v3)

- Based upon a « simple » divide & conquer parallelized approach
- Along the recurrence principle
  - yi = y'i  if i<= n/2
  - yi = y'(n/2) *OP* y'i  if i> n/2

# Simulation of v3 algo

- A=(8, 9,  10, 11, 12, 13, 14, 15)
- Prefix sum : final B[k]=A[1]+…+A[k] for each k
  B=(8, 17, 27, 38, 50, 63, 77, 92)



# Remarks about v3

- Requires a CREW PRAM
  - At each merging of 2 sub problems:
    - Concurrent read of value y'(n/2) by all the procs. in charge of indices > n/2
      - Duplicate in a subsidiary array of length (n/2), the value y'(n/2) as many times it needs to be read « concurrently »
      - Cf TD1

- Parallel time complexity : O(logn)
- At each parallel step, at most n/2 procs. needed
  - Possible to reduce this amount by a O(logn) factor