SE – PRAM (corrigé)

ING2 – GSI/MI – Algorithmique parallèle

Année 2016–2017

PRAM

(1) Write a program for a CREW PRAM that find whether some element e is in an array of n elements, $e_{i=1,n}$, all distinct. What is its complexity? □

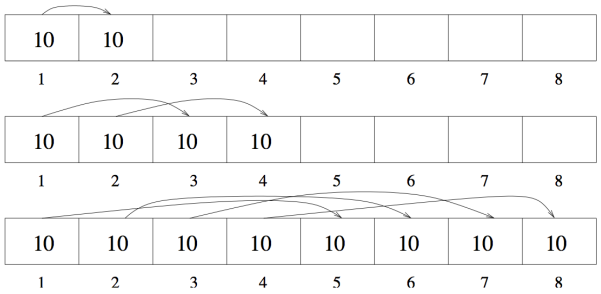
```
Initial condition: list of n elements stored in E[0 ... n], element e
Final condition: isIn=TRUE if e is in E[0 ... n]
Global variables: boolean isIn
begin
spawn(P1...PN)
forall 0 <i <n
do
    if( e == E[i] )  isIn=TRUE
endforall
```

Because all elements are distinct only one processor will write, the write is exclusive, The time complexity is O(1), there is only one step.

(2) Can the same program work in a EREW PRAM. Why? What would be its complexity? □

Because in a EREW PRAM you can not read concurrently the same variable you can not do it the same. If it where done sequentially its complexity would be O(n), with the broadcast of the next exercise it would be O(log(n)). This is similar to what would happen in a distributed memory machine, were processors can not see each other memories.

(3) Write a program for a EREW PRAM that broadcasts a variable to all processors, that is, ensures all processors have a local copy of a variable. What is its complexity? □

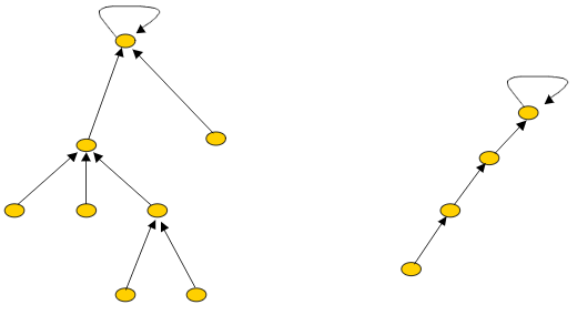


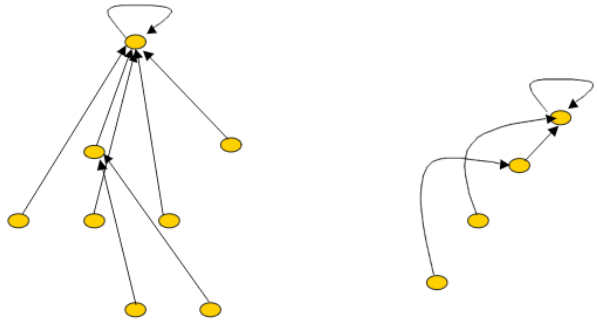
We are doubling in each step how many copies we are making, so there are log(n) steps, and at every step we activate double the processors. This version assumes there are 2^i P, if there were less we would have to check the limits. This is done in the next exercises.

```
Initial condition: Processor P0 has element e stored in R
Final condition: All processors have element e stored in A[i]
begin
for 0 <= i < log(n)                    \\There are log(n) steps
do
    forall 0 <= j < 2^i                 \\Each time we activate the double of processors
    do
         A[j+2^i] = A[j]
    endforall
endfor
```

Its time complexity is O(log(n)), the number of steps.

4 Write a program that finds for each node in a DAG, its root, following for each node its successors (vertex closer to the root). □





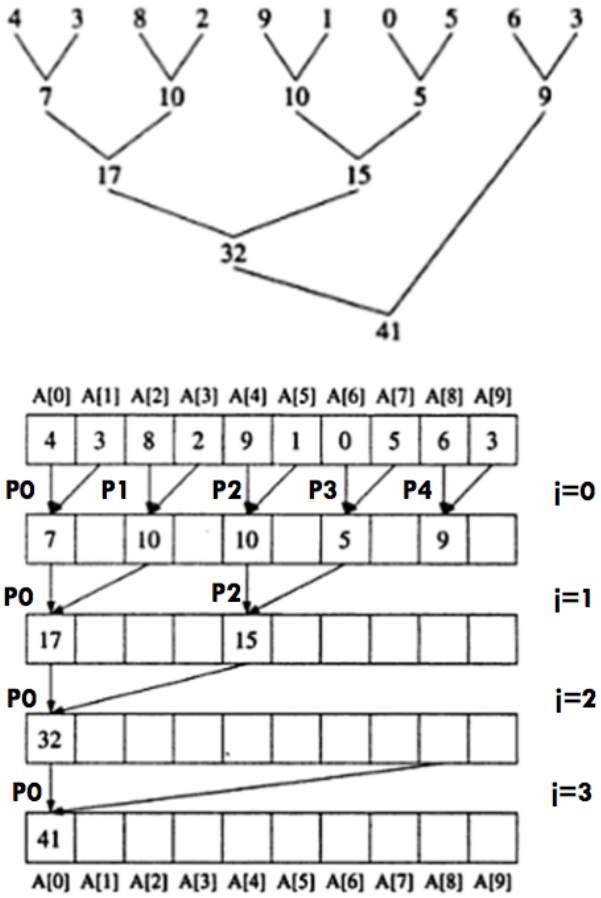
(From wikipedia) Following a path in a graph is an inherently serial operation, but pointer jumping reduces the total amount of work by following all paths simultaneously and sharing results among dependent operations. Pointer jumping iterates and finds a parent (a vertex closer to the tree root) each time. By following parents computed for other vertices, the traversal down each path can be doubled every iteration, which means that the tree roots can be found in logarithmic time.

Pointer doubling operates on an array successor with an entry for every vertex in the graph. Each $root[i]$ is initialized with itself if that vertex is t a root (parent NULL, or itself in the figure). At each iteration, each parent is updated to its parent's parent. The root is found when the parents's parent points to the end. This destroys the tree, a copy could be made to keep it.

```
Initial condition: An array parent representing a forest of trees. parent[i] is the par
Final condition: An array containing the root ancestor for every vertex
begin
forall processor i, in parallel
    do if parent[i] = NULL
    then root[i]=i
while there exist a node i such that parent[i]!=NULL
   do forall processor i, in parallel
       if parent[i] !=NULL
       then root[i]  = root[parent[i]]
       parent[i] = parent[parent[i]]
```

Its complexity is O(log(n))

(5) Write a program for a EREW PRAM that find computes the sum of all elements in an array A of size n. This is called a reduction. What´s its complexity? □



The processors in a PRAM algorithm manipulate data stored in global registers. In this example we start by spawning the processors we need and the activating them if needed in that iteration. For adding n numbers we spawn $n/2$ processors, as seen in the tree. Each addition corresponds to:

$A[2i] + A[2i + 2^j]$, because we are spanning only $n/2$ processors

Note, the processor which is active has an i such that:

$imod2^j = 0$ (ie. keep only those processors active, less every iteration).

Also check that the array does not go out of bound:

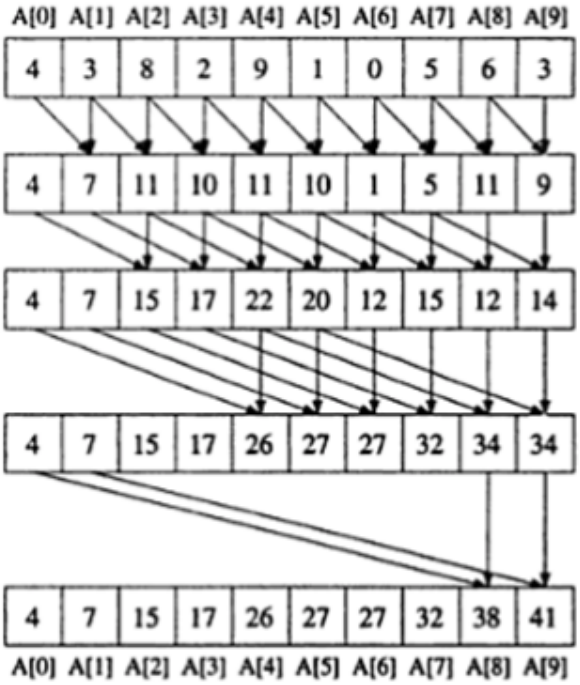
$2i + 2^j < n$ The SPAWN routine requires log(n/2) doubling steps. The sequential for loop executes log(n) times. Note that if n is not divisible by 2 its log(n) will not be an integer (for example log(9) = 3.-), so the < ensures that extra iteration is made. Each iteration takes constant time. Hence overall time complexity is $\Theta(log(n))$ given $n/2$ processors.

```
Initial condition: list of n>= 1 elements stored in A[0 ... n-1]
Final condition: Sum of elements stored in A[0]
```

```
Global variables: n, A[0...n-1],j
begin
forall 0<=i<= (n/2)-1 do in parallel                    //We need n/2 processors
    for 0<=j<  log(n) do                                //We need log(n) iterations
        if (i modulo 2^j = 0 AND 2i + 2^j < n) then     //Activate P and check limits
             A[2i] = A[2i] + A[2i + 2^j]
        endif
    endfor
enforall
```

Its complexity is O(log(n))

(6) Write a program for a EREW PRAM that find computes for each A[i] in an array A[0..n] the value A[i]=A[0]+A[1]+A[2]+...+A[i] (This is called a prefix sum). What´s its complexity? □



There are n-1 processors activated, ass you see in the figure.
Each one accesses $A[i]$, then accesses $A[i-2^j]$, where j is the depth (j varies from 0 to $log(n)$.
At every step we are using all processors larger than 2^j. Of course, the bounds need to be checked, but because the limits of the array are the same as the limits of the processors (n) and we are

looking "left" towards 0 it is not necessary.
Works for any operation involving Running time is t(n) = $\Theta(logn)$. Cost is $c(n) = p(n) \times t(n) = \Theta(nlg(n))$.
Note not cost optimal, as RAM takes $\Theta(n)$ Can be made optimal.

(Wikipedia) Prefix sums are trivial to compute in sequential models of computation, by using the formula $y_i = y_i - 1 + x_i$ to compute each output value in sequence order. However, despite their ease of computation, prefix sums are a useful primitive in certain algorithms such as counting sort, and they form the basis of the scan higher-order function in functional programming languages. Prefix sums have also been much studied in parallel algorithms, both as a test problem to be solved and as a useful primitive to be used as a subroutine in other parallel algorithms. In the parallel random access machine model of computing, prefix sums can be used to simulate parallel algorithms that assume the ability for multiple processors to access the same memory cell at the same time, on parallel machines that forbid simultaneous access. By means of a sorting network, a set of parallel memory access requests can be ordered into a sequence such that accesses to the same cell are contiguous within the sequence; scan operations can then be used to determine which of the accesses succeed in writing to their requested cells, and to distribute the results of memory read operations to multiple processors that request the same result. Parallel prefix (using multiplication as the underlying associative operation) can also be used to build fast algorithms for parallel polynomial interpolation. There are improvements to be made to save memory.

```
Initial condition: list of n>= 1 elements stored in A[0 ... n-1]
Final condition: Each elements A[i] contains A[i]=A[0]+A[1]+A[2]+...+A[i]
Global variables: n, A[0...n-1],j
begin
forall 1<=i< n do in parallel            \\We use n-1 processors, starting in 1
    for 0<=j<  log(n) do                 \\log(n) steps
        if i >2^j then                   \\processors larger than 2^j are involved
             A[i] = A[i] + A[i - 2^j]     \\We look left on the array
        endif
    endfor
enforall
```

Its time complexity is O(log(n))

Erathostenes

(7) Write a sequential algorithm to find the first n prime numbers. You will give the temporal and spatial complexity. □

1- Create a list of unmarked natural numbers $2, 3, ?, n$

2 - $k = 2$

3 - Repeat

(a) Mark all multiples of k between k^2 and n

(b) Let k = smallest unmarked number > k

until $k^2 > n$

4. The unmarked numbers are primes

NAIVE:

```
Primes[2..n]=[TRUE]
k=2                                  \\We start with 2
while k < n do                       \\Until we reach n
    next = 0
    for 2 <i < n do                   \\Loop all numbers
       if (i modulo k) == 0 then      \\Is divisible
             Primes[i] == FALSE
       else if next == 0             \\The first non divisible will be a prime
             next = i
       endif
    endfor
    k = next                          \\Get ready for next
    next = 0
endwhile
```

There are better versions, we do not care:

```
Input: an integer n > 1

Let A be an array of Boolean values, indexed by integers 2 to n,
initially all set to true.
```

```
for i = 2, 3, 4, ..., not exceeding sqrt(n):
  if A[i] is true:
    for j = i^2, i^2+i, i^2+2i, i2+3i, ..., not exceeding n :
      A[j] := false

Output: all i such that A[i] is true.
```

It includes a common optimization, which is to start enumerating the multiples of each prime i from i^2. The time complexity of this algorithm is O(n log log n) (check in google $\Theta(n+n/2+n/3+n/5+n/7...)$

.

Spatial complexity $\Theta(n)$

(8) Repeat the previous exercise by writing a parallel algorithm for priority CRCW PRAM. How many steps does it take? How many processors are needed? You will give the temporal and spatial complexities. □

If we just simple make the loop a for all works, we check all number at the same time. In a priority CRCW only the lowest P will write the next prime

```
Primes[2..n]=[TRUE]
k=2                                    \\We start with 2
while k < n do                         \\Until we reach n
    next = 0
    forall 2 <i < n do                 \\Do all numbers at the same time
       if (i modulo k) == 0 then        \\Is divisible
             Primes[i] == FALSE
       else if next == 0                \\The first non divisible will be a prime
             next = i
       endif
    endforall
    k = next                           \\Get ready for next
    next = 0
endwhile
```

More formal, we can find the next prime in every iteration. This would be what has to be done when we do not have n processors. With just p processors $p < n$ the number would have to be split

among them, each would check with the same prime all divisibles, then they would communicate their lower prime candidate and decide the next candidate among all possibilities.

```
Primes[2..n]=[TRUE]
while k < n do                          \\Until we reach n
    forall 2 <i < n do                  \\Check the candidate, communication
       if(Primes[i]==TRUE)
          k=i                          \\only one writes, priority CRCW
    endforall
    forall 2 <i < n do                  \\Do all numbers at the same time
       if (i modulo k) == 0 then         \\Is divisible
             Primes[i] == FALSE
       endif
    endforall
endwhile
```

Total sieves needed sqrt(n)(ln(sqrt(n)) Temporal complexity $\Theta(nlog(n))$
$\Theta(nlog(n))/p + sqrt(n)(ln(sqrt(n)) + \Theta(log(n))$

Spatial complexity $\Theta(n)$

If broadcast is needed, with p processors : $+\Theta(log(n))$ Same time complexity as before, but faster because we have p processors:
$\Theta(n/ploglog(n))$