

P-RAM

1 Sélection dans une liste

▷ **Question 1** Soit L une liste contenant n objets coloriés soit en bleu, soit en rouge. Concevoir un algorithme EREW efficace qui sépare les éléments bleus des éléments rouges (c'est-à-dire qui construit une nouvelle liste ne contenant que les éléments bleus).

2 Recherche des racines dans une forêt

On donne ici un autre exemple de problème pour la séparation des modèles EREW et CREW. Soit \mathcal{F} une forêt d'arbres binaires. Chaque nœud i d'un arbre est associé à un processeur $P(i)$ et possède un pointeur vers son père $pere(i)$. On va chercher des algorithmes EREW et CREW pour que chaque nœud connaisse la racine de son arbre (notée $racine(i)$), et ainsi prouver l'intérêt des lectures concurrentes.

▷ **Question 2** Donner un algorithme P-RAM CREW pour que chaque nœud détermine $racine(i)$. Démontrer que l'algorithme proposé n'utilise que des lectures concurrentes et déterminer sa complexité.

3 Procédure mystère

On définit les deux opérateurs suivants pour un tableau $A = [a_0, a_1, \dots, a_{n-1}]$ de n entiers :

– $PRESCAN(A)$ renvoie le tableau $[0, a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots, a_0 + a_1 + \dots + a_{n-2}]$

– $SCAN(A)$ renvoie le tableau $[a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots, a_0 + a_1 + \dots + a_{n-1}]$

Nous avons vu en cours comment réaliser ces opérateurs en temps $O(\log n)$ sur une P-RAM EREW.

On considère la procédure SPLIT suivante :

```

SPLIT( $A, Flags$ )
   $I_{down} \leftarrow PRESCAN(not(Flags))$ 
   $I_{up} \leftarrow n - REVERSE(SCAN(REVERSE(Flags)))$ 
  Pour  $i = 1$  to  $n$  en parallèle
    Si  $Flags(i)$  Alors
       $Index[i] \leftarrow I_{up}[i]$ 
    Sinon
       $Index[i] \leftarrow I_{down}[i]$ 
   $Result \leftarrow PERMUTE(A, Index)$ 
  Renvoyer  $Result$ 

```

Les noms des différentes fonctions sont relativement intuitifs ; en particulier, REVERSE renverse le tableau, et PERMUTE($A, Index$) réordonne le tableau A selon la permutation $Index$.

▷ **Question 3** On considère l'entrée suivante :

$$\begin{array}{rcl}
 A & = & [5 \ 7 \ 3 \ 1 \ 4 \ 2 \ 7 \ 2] \\
 Flags & = & [1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0]
 \end{array}$$

Appliquez la procédure SPLIT à cette entrée. Intuisez par l'exemple ce que fait la procédure SPLIT.

▷ **Question 4** Prouvez votre intuition. Quel est le coût la procédure SPLIT ?

On considère la procédure MYSTÈRE suivante :

```
MYSTÈRE( $A$ ,  $Number\_of\_Bits$ )
Pour  $i = 0$  to  $Number\_of\_bits - 1$ 
     $bit(i) \leftarrow$  tableau indiquant si le  $i$ -ème bit
    des éléments de  $A$  est à 1
     $A \leftarrow$  SPLIT( $A, bit(i)$ )
```

- ▷ **Question 5** *Faire tourner la procédure sur $A = [5, 7, 3, 1, 4, 2, 7, 2]$ avec $Number_of_Bits = 3$.*
- ▷ **Question 6** *Que fait la procédure MYSTÈRE ?*
- ▷ **Question 7** *Avec des entrées de taille $O(\log n)$ bits, quelle est la complexité avec n processeurs ? Et avec seulement p processeurs ? Quelles sont les valeurs de p les plus intéressantes ?*

4 Réponses aux exercices

▷ Question 1, page 1

On utilise la technique de saut de pointeur vu précédemment pour concevoir l'algorithme en-dessous. Chaque processeur détermine le processeur bleu qui le suit dans la liste en temps $O(\log n)$.

Cet algorithme est bien EREW, mais cela demande une petite explication. Il fonctionne comme l'algorithme du saut de pointeur en parallèle sur plusieurs listes dont le nœud terminal est bleu ou *NIL*, chaque processeur conservant un pointeur sur la fin de sa liste, donc sur le prochain élément bleu. Chaque saut de pointeur se fait sur des listes indépendantes et n'interfère pas avec les autres. À la fin de l'algorithme, la liste des éléments bleus commence avec le premier élément de la liste initiale si ce dernier est bleu et avec son successeur bleu sinon :

```

EXTRAIT-BLEUS()
1:  Pour tout  $i$  en parallèle :
2:    Si  $suivant(i) = NIL$  Or  $couleur(suivant(i)) = bleu$  Alors
3:       $fini(i) \leftarrow Vrai$ 
4:       $bleu(i) \leftarrow suivant(i)$ 
5:    Tant que il existe un nœud  $i$  tel que  $fini(i) = Faux$  :
6:      Pour tout  $i$  en parallèle :
7:        Si  $fini(i) = Faux$  Alors
8:           $fini(i) \leftarrow fini(suivant(i))$ 
9:          Si  $fini(i) = True$  Alors  $bleu(i) \leftarrow bleu(suivant(i))$ 
10:          $suivant(i) \leftarrow suivant(suivant(i))$ 

```

▷ Question 2, page 1

L'algorithme naturel pour chercher les racines utilise la technique de saut de pointeur, un nœud et ses ascendants partageant le chemin conduisant à la racine :

```

CALCUL_RACINE()
1:  Pour tout  $i$  en parallèle :
2:    Si  $pere(i) = NIL$  Alors  $racine(i) = i$ 
3:    Tant que il existe un nœud  $i$  tel que  $pere(i) \neq NIL$  :
4:      Pour tout  $i$  en parallèle :
5:        Si  $pere(i) \neq NIL$  Alors
6:          Si  $pere(pere(i)) = NIL$  Alors  $racine(i) = racine(pere(i))$ 
7:           $pere(i) = pere(pere(i))$ .

```

Cet algorithme est bien de type CREW puisque les seules écritures effectuées par le processeur i concernent des données qui lui sont propres ($racine(i)$ et $pere(i)$). Par contre, cet algorithme n'est pas EREW puisque plusieurs processeurs peuvent avoir le même père (surtout à la fin!) et donc peuvent accéder simultanément à la même donnée en lisant $pere(i)$ par exemple.

L'analyse de complexité effectuée pour le calcul de la distance à la fin d'une liste s'applique : tous les nœuds des arbres de la forêt connaissent leur racine en temps $O(\log d)$, où d est la profondeur maximale des arbres.

▷ Question 3, page 1

$$\begin{array}{rcl}
 A & = & [\ 5 \ 7 \ 3 \ 1 \ 4 \ 2 \ 7 \ 2 \] \\
 Flags & = & [\ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \] \\
 Idown & = & [\ 0 \ 0 \ 0 \ 0 \ \boxed{0} \ \boxed{1} \ 2 \ \boxed{2} \] \\
 Iup & = & [\ \boxed{3} \ \boxed{4} \ \boxed{5} \ \boxed{6} \ 7 \ 7 \ \boxed{7} \ 8 \] \\
 Index & = & [\ 3 \ 4 \ 5 \ 6 \ 0 \ 1 \ 7 \ 2 \] \\
 Result & = & [\ 4 \ 2 \ 2 \ 5 \ 7 \ 3 \ 1 \ 7 \]
 \end{array}$$

L'analyse de l'exemple montre bien l'effet de la procédure *SPLIT* : les éléments du vecteur A dont la composante de $Flags$ vaut 0 sont regroupés au début du vecteur $Result$, en conservant leur ordre initial. De même, les éléments du vecteur A dont la composante de $Flags$ vaut 1 sont regroupés à la fin du vecteur $Result$, en conservant leur ordre initial.

▷ **Question 4, page 1**

Prouver d'abord que le tableau *Index* contient tous les éléments de 0 à $n - 1$, et que tous les petits éléments sont dûs à *Idown* et les grands à *Iup*. Ceci prouve que les éléments pour lesquels *Flags* est à 0 sont mis au début de *Result* et les autres à la fin : on a bien un split. Reste à prouver que l'ordre des éléments pour un même *Flag* est bien conservé. C'est facile pour **PRESCAN**, et pareil pour l'autre si on comprend que le **REVERSE(SCAN(REVERSE(*Flags*)))** effectue simplement un **SCAN** à partir de la fin du tableau *Flags*. (Note : les booléens sont considérés comme des entiers pour les scans : $1 + 1 = 2 \dots$)

Avec $O(n)$ processeurs, les opérations **SCAN** et **PRESCAN** ont un coût $O(\log n)$, et les autres opérations ont un coût constant, donc la procédure **SPLIT** toute entière a un coût $O(\log n)$.

▷ **Question 5, page 2**

On commence avec l'exemple pour voir ce que fait cette mystérieuse procédure :

$$\begin{array}{lcl}
 A & = & [\ 5 \ 7 \ 3 \ 1 \ 4 \ 2 \ 7 \ 2 \] \\
 bit(0) & = & [\ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \] \\
 A \leftarrow \text{SPLIT}(A, bit(0)) & = & [\ 4 \ 2 \ 2 \ 5 \ 7 \ 3 \ 1 \ 7 \] \\
 bit(1) & = & [\ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \] \\
 A \leftarrow \text{SPLIT}(A, bit(1)) & = & [\ 4 \ 5 \ 1 \ 2 \ 2 \ 7 \ 3 \ 7 \] \\
 bit(2) & = & [\ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \] \\
 A \leftarrow \text{SPLIT}(A, bit(2)) & = & [\ 1 \ 2 \ 2 \ 3 \ 4 \ 5 \ 7 \ 7 \]
 \end{array}$$

▷ **Question 6, page 2**

La procédure a bien l'air de trier le tableau *A*. En fait, c'est une mise en œuvre du tri par base, ou *radix-sort* : on commence avec le bit le moins significatif, et la procédure **SPLIT** partitionne le tableau initial en deux paquets selon la valeur de ce bit. La procédure de tri fonctionne parce que chaque appel à **SPLIT** trie les éléments selon la valeur du bit courant, tout en maintenant l'ordre relatif aux bits précédents : d'où la nécessité d'aller du bit le moins significatif au bit le plus significatif.

▷ **Question 7, page 2**

Il y a $O(\log n)$ itérations, d'où un tri de coût $O(\log^2 n)$ avec $O(n)$ processeurs. Si on utilise seulement p processeurs, le coût de **SPLIT** devient $O(\frac{n}{p} + \log n)$ (théorème de Brent), et celui du tri $O((\frac{n}{p} + \log n) \log n) = O(\frac{n}{p} \log n + \log^2 n)$. Donc pour $p = n$ processeurs, la complexité principale est le $\log^2 n$, et cela le reste tant que $p \geq \frac{n}{\log n}$. En dessous, on retrouve une composante linéaire avec $\frac{n}{p} + \log n$ et la complexité totale tend vers $O(n \log n)$ ce qui est rassurant (meilleure complexité du tri sur un processeur). À l'égalité, on a la même complexité mais avec beaucoup moins de n processeurs.