INF7440 — Conception et analyse d'algorithmes
Examen final (Automne 2003)

Durée: 18h00 – 21h00 (trois heures) **Documentation autorisée:** Documentation personnelle.

Remarques :

- L'examen comporte trois (3) questions obligatoires, comptant respectivement pour 20, 10 et 10 points (total de 40 points).

  L'examen comporte aussi un certain nombre de *questions bonus*, optionnelles, qui seront corrigées de façon plus stricte. (Il est donc possible d'obtenir une note supérieure à 40.)

- Vous devez écrire vos algorithmes, surtout lorsqu'il s'agit de segment de code avec parallélisme, en utilisant une notation semblable à MPD. Toutefois, dans le cas des instructions `co`, vous pouvez utiliser une notation *moins stricte* que la notation MPD acceptée par le compilateur. Plus précisément, le compilateur MPD *n'accepterait pas* l'instruction `co` suivante, parce que le corps de l'instruction `co` n'est pas une invocation :

  ```
  co [i = 1 to n]
    x[i] = y[i] + z[i];
  oc
  ```

  Le compilateur obligerait plutôt à introduire une procédure auxiliaire et à l'utiliser dans la partie droite de l'affectation :
  `procedure sum2( int x, int y ) returns int r { r = x + y; }`.

  Dans le cadre de l'examen, la première notation (sans procédure auxiliaire) sera acceptée. En fait, *n'importe quel* bout de code dans un `co` sera accepté.

- Pour simplifier la présentation et l'analyse des algorithmes que vous écrirez, vous pourrez supposer, lorsque cela sera utile, que la taille du problème est un nombre d'une forme appropriée, par ex., $n = 2^k$, $\frac{n}{\lg\ n} = 2^l$, etc.

- À moins d'indication contraire, vous pouvez utiliser, sans la définir, la procédure parallèle suivante, de temps $\Theta(\lg\ n)$ et de coût optimal $\Theta(n)$ (qui utilise donc $\frac{n}{\lg\ n}$ processeurs) :

  ```
  optype OpBinaire = (int, int) returns int;

  procedure calculerPrefixes( int x[*], res int prefixes[*], int n, cap OpBinaire op0 )
  ```

- À moins d'indication contraire (par ex., le bonus à la question 1.b.), les algorithmes que vous écrivez *n'ont pas besoin d'être optimaux* (en coût).

- Pour justifier vos analyses de complexité, vous pouvez, le cas échéant, annoter chacune des parties de votre algorithme avec sa complexité associée puis ensuite synthétiser le tout.

- En MPD, on a les équivalences suivantes : `int(true) == 1` ; `int(false) == 0`.

1. Nombre d'occurrences d'un élément (20 pts)

Soit `A` un tableau (vecteur) de `n` nombres entiers. Soit `inf` et `sup` deux entiers arbitraires. On veut concevoir une procédure qui permet déterminer le nombre d'éléments de `A` qui sont compris (inclusivement) entre `inf` et `sup` :

```
procedure trouverOccurrences( int A[*], int n, int inf, int sup )
          returns int nb
# POSTCONDITION
#   nb = NUMBER( 1 <= i <= n SUCH THAT inf <= A[i] <= sup :: i )
```

Exemples, pour `A = (10, 20, 30, 10)` :

- `trouverOccurrences( A, 4, 20, 10 )` $\Rightarrow 0$
- `trouverOccurrences( A, 4, 10, 10 )` $\Rightarrow 2$
- `trouverOccurrences( A, 4, 10, 30 )` $\Rightarrow 4$

a. Version parallèle récursive (11 points) :

(i) Écrivez une version *parallèle* et récursive de la procédure `trouverOccurrences` basée sur une stratégie diviser-pour-régner "trichotomique" ($\Rightarrow$ décomposition en trois (3) sous-problèmes de taille équivalente).

(ii) Donnez les équations de récurrence décrivant le temps d'exécution de cette procédure. (N'oubliez pas de spécifier les conditions appropriées sur `n`.) Quel est alors le temps d'exécution résultant (complexité asymptotique)?

(iii) Combien de processeurs sont requis (nombre exact)?

(iv) Quel est le coût de votre algorithme? Est-il optimal (en coût)?

(v) Le *travail* effectué par votre algorithme est-il inférieur, égal ou supérieur au coût? Justifiez brièvement votre réponse.

b. Version parallèle non récursive (9 points) :

(i) Écrivez une version *parallèle* mais non récursive de la procédure `trouverOccurrences`. Cette version parallèle doit évidemment être *asymptotiquement plus rapide* qu'une version de cette procédure qui serait séquentielle.

(ii) Quel est le temps d'exécution (asymptotique)?

(iii) Combien de processeurs sont requis (nombre exact)?

(iv) Quel est le coût de votre algorithme? Est-il optimal (en coût)?

- **Bonus (2 pts)** : Un *bonus* sera accordé si votre algorithme non récursif est *de coût optimal*.

2. Tri par fusion à la PRAM (10 pts)

On a vu au chapitre 2 ("Diviser-pour-régner") la méthode récursive de "tri par fusion" pour effectuer le tri d'une séquence d'éléments. Cette méthode de tri repose sur l'utilisation d'une procédure `fusionner` qui permet de *combiner* deux sous-séquences *déjà triées* en une seule séquence triée. La spécification de cette procédure est la suivante :

```
procedure fusionner( ref int A[*], int inf, int mid, int sup )
# PRECONDITION
#   inf <= mid <= sup,
#   ALL( inf   <= i < mid :: A[i] <= A[i+1] ),
#   ALL( mid+1 <= i < sup :: A[i] <= A[i+1] )
# POSTCONDITION
#   A[inf:sup] est une permutation de A'[inf:sup]
#   ALL( inf <= i < sup    :: A[i] <= A[i+1] )
```

Quelques exemples d'appel :

```
fusionner( [287, 306, 642, 604, 73, 610, 618, 712], 3, 3, 4 )
  => A[3:4] = [604, 642]    (le reste de A est inchangé)

fusionner( [287, 306, 604, 642, 73, 610, 618, 712], 5, 5, 6 )
  => A[5:6] = [73, 610]     (le reste de A est inchangé)

fusionner( [287, 306, 604, 642, 73, 610, 618, 712], 1, 4, 8 )
  => A[1:8] = [73, 287, 306, 604, 610, 618, 642, 712]
```

a. En *utilisant* la procédure `fusionner` (donc pas besoin de définir `fusionner`), concevez une procédure `trier`, dont l'en-tête est indiqué plus bas, qui permet de trier une séquence de nombres, et ce de façon *parallèlle* à la PRAM, donc sans récursion.

```
procedure trier( ref int A[*], int n )
# PRECONDITION
#   n >= 1
# POSTCONDITION
#   A est une permutation de A',
#   ALL( 1 <= i < n :: A[i] <= A[i+1] )
```

b. Quel est le temps d'exécution (asymptotique) de votre algorithme?

c. Combien de processeurs sont requis (nombre exact) par votre algorithme?

d. Quel est le coût de votre algorithme? Est-il optimal (en coût)?

3. Partition d'une séquence d'entiers (10 pts)

On veut *partitionner* une séquence de n entiers (positifs) en k sous-séquences de façon à ce que les sommes des éléments des diverses sous-séquences soient les plus égales possible entre elles. Ici, par partitionner en entend découper en k sous-séquences d'éléments adjacents. Par exemple, la séquence `[10, 12, 23, 10]` de $n = 4$ éléments peut être décomposée en $k = 2$ sous-séquences de plusieurs façons différentes :

- `[]` et `[10, 12, 23, 10]`
- `[10]` et `[12, 23, 10]`
- `[10, 12]` et `[23, 10]`
- `[10, 12, 23]` et `[10]`
- `[10, 12, 23, 10]` et `[]`

Plus précisément, le coût d'une sous-séquence est déterminé par la *somme* des éléments de cette sous-séquence. Le coût global d'une partition (groupe de sous-séquences) est alors le *maximum* des coûts des diverses sous-séquences associées à cette partition. Évidemment, on cherche à *minimiser* ce coût.

Pour simplifier le problème, on cherche simplement à déterminer *le coût* d'une partition optimale, et non le contenu exact de cette partition.

L'en-tête d'une procédure solutionnant ce problème est le suivant :

```
procedure partitionner( int nbs[*], int n, int k ) returns int cout
# PRECONDITION
#   n >= 1
#   k >= 1
#   ALL( 1 <= i <= n :: nbs[i] > 0 )
```

Quelques exemples d'appels à cette procédure produiraient les résultats suivants :

- `partitionner( [1, 2, 3, 2], 4, 1 )` ⇒ coût = 8

  Une partition en une seule sous-séquence doit nécessairement inclure tous les éléments.
- `partitionner( [1, 2, 3, 2], 4, 2 )` ⇒ coût = 5

  La partition optimale est celle pour les sous-séquences suivantes : `[1, 2]` et `[3, 2]` (coût = 5).

  Les autres partitions possibles sont les suivantes et sont toutes de coût supérieur :

  `[]` et `[1, 2, 3, 2]` (coût = 8) — `[1]` et `[2, 3, 2]` (coût = 7)
  `[1, 2, 3]` et `[2]` (coût = 6) — `[1, 2, 3, 2]` et `[]` (coût = 8)
- `partitionner( [1, 2, 3, 2], 4, 3 )` ⇒ coût = 3
- `partitionner( [1, 2, 3, 2], 4, 4 )` ⇒ coût = 3

Un algorithme récursif pour résoudre ce problème est présenté à la figure 1.

```
procedure somme( int nbs[*], int i, int j ) returns int s
{
   s = 0;
   for [k = i to j] {
     s += nbs[k];
   }
}

procedure partitionner( int nbs[*], int n, int k ) returns int cout
{
  if (n == 1) {
    cout = nbs[1];
  } else if (k == 1) {
    cout = somme( nbs, 1, n );
  } else {
    cout = high(int);
    for [i = 1 to n] {
      int gauche = partitionner( nbs, i, k-1 );
      int droite = somme( nbs, i+1, n );
      cout = min( cout, max(gauche, droite) )
    }
  }
}
```

Figure 1: Algorithme récursif pour partitionner une séquence

a. Écrivez une procédure `partitionner` séquentielle et *non récursive* qui est *asymptotiquement plus efficace* que l'algorithme de la figure 1.

   Note : Si vous ne réussissez pas à écrire un algorithme *non récursif*, vous pouvez écrire un algorithme récursif en autant qu'il soit *asymptotiquement plus efficace* que celui présenté plus haut. Toutefois, dans ce cas, vous n'aurez qu'une partie des points.

b. Quelle est la complexité asymptotique de votre algorithme? (Une borne supérieure O sera acceptée, en autant que cette borne soit suffisamment précise.)

   Note : N'oubliez pas qu'une fonction de complexité peut dépendre de *plusieurs* arguments. N'oubliez pas aussi de tenir compte de l'appel à `somme`.

- **Bonus (4 pts)** De quelle façon pourrait-on *paralléliser* l'algorithme non récursif?

  (Vous n'êtes pas obligé de donner le code : indiquez simplement, en quelques mots, de quelle façon on pourrait procéder. Mais notez bien que la bonne réponse ne consiste pas simplement à dire que toutes les boucles `for` devraient être remplacées par des `co`.)

  Que deviendrait alors le temps d'exécution?

4. Algorithme *branch-and-bound* (question optionnelle) (Bonus 5 pts)

Soit un type `Arbre` binaire défini par les constantes et types suivants, où l'invariant indique que tous les champs `valeur` sont non négatifs et qu'un noeud interne possède toujours deux (2) enfants alors qu'une feuille n'en possède aucun :

```
const int FEUILLE = 0;
const int NOEUD   = 1;

type Arbre = ptr ArbreRec;
type ArbreRec = rec( int sorte; Arbre gauche, droit; int valeur; );
# INVARIANT
#   valeur >= 0
#   sorte = FEUILLE or sorte = NOEUD
#   sorte = FEUILLE => gauche  = null & droit  = null
#   sorte = NOEUD   => gauche ~= null & droit ~= null
```

La procédure suivante permet d'explorer un arbre `a` et de déterminer le coût associé au chemin formé par la séquence de noeuds allant de la racine de l'arbre vers les feuilles, noeuds qui doivent satisfaire la condition spécifiée par `cond` (de type `Condition`) :

```
optype Condition = (Arbre) returns bool;

procedure traverserArbre( Arbre a, int cumul, cap Condition cond )
{
  if (cond(a)) {
    if (a^.sorte == FEUILLE) {
      printf( "Cout du chemin = %d\n", cumul+a^.valeur );
    } else {
      traverserArbre( a^.gauche, cumul+a^.valeur, cond );
      traverserArbre( a^.droit,  cumul+a^.valeur, cond );
    }
 }
}
```

Par exemple, l'appel suivant permettrait d'imprimer le coût de chacun des chemins allant de la `racine` vers des feuilles et pour lesquels les noeuds ont un champ `valeur` qui est pair :

```
procedure estPair( Arbre a ) returns bool r
{ r = (a^.valeur % 2) == 0; }

traverserArbre(racine, 0, estPair);
```

a. Modifiez la procédure `traverserArbre` pour faire en sorte que *seul le chemin de coût minimal* soit imprimé, et ce en utilisant une stratégie de type *branch-and-bound*. En d'autres mots, vous devez éviter d'explorer les sous-arbres pour lesquels le chemin déjà exploré est assuré de conduire à un résultat qui ne serait pas optimal.

b. Serait-il possible d'introduire une forme de *best-first search* dans votre algorithme? Si oui, expliquez brièvement de quelle façon?