INF7440 — Conception et analyse d'algorithmes
Examen final (Hiver 2005)

Durée: 120 minutes **Documentation autorisée:** Documentation personnelle.

Remarques :

- L'examen comporte trois (3) questions, mais vous pouvez en faire seulement deux (2) parmi ces trois. La question no. 1 est obligatoire et compte pour 35 points. Vous devez ensuite répondre à l'une des deux questions parmi les questions 2 et 3, chacune comptant pour 15 points. Si vous répondez aux deux questions au choix, un bonus (maximum 50 % des points) pourra vous être accordé, pour une note maximale de 57,5 sur 50.

- 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 celle acceptée par le compilateur, par exemple, instructions `co` contenant des segments arbitraires de code.

  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 ou fonction auxiliaire et à l'utiliser dans la partie droite de l'affectation. Dans le cadre de l'examen, la 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., $n2^k$, $\frac{n}{\lg\ n} = 2^l$, etc. Toutefois, si vous posez une telle hypothèse, vous devez l'indiquer clairement et explicitement

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

- Les algorithmes que vous écrivez *n'ont pas besoin d'être optimaux* (en travail ou en coût). Toutefois, si on vous demande d'écrire un algorithme parallèle, cet algorithme devra évidemment être *asymptotiquement plus rapide* que l'algorithme séquentiel correspondant.

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

1. Algorithmes parallèles pour matrices (obligatoire) (35 points)

Une certain nombre d'opérations de manipulation de matrices vous sont présentées à la page suivante. Vous devez développer deux (2) versions, *parallèles*, d'une fonction `transposee`, dont l'interface est la suivante :

```
op transposee( ptr Matrice p ) returns ptr Matrice r
```

Rappelons la définition de la transposée d'une matrice. Soit A une matrice $n \times n$:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n-1} & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n-1} & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn-1} & a_{nn} \end{pmatrix}$$

Alors la *transposée* de A — `transposee(`A`)` — est la matrice suivante :

$$\texttt{transposee}(A) = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{n-11} & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n-12} & a_{n2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{1n-1} & a_{2n-1} & \dots & a_{n-1n-1} & a_{nn-1} \\ a_{1n} & a_{2n} & \dots & a_{n-1n} & a_{nn} \end{pmatrix}$$

Plus précisément, vous devez définir deux versions de la fonction `transposee`, chacune étant le plus parallèle possible (pour obtenir un temps d'exécution le plus rapide possible) :

a. Une première version qui utilise du *parallélisme itératif à granularité fine*.

b. Une deuxième version qui utilise du *parallélisme récursif*. Plus précisément, soit A une matrice $n \times n$. Soit alors les quatre (4) sous-matrices suivantes de A, chacune de taille $\frac{n}{2} \times \frac{n}{2}$:

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)$$

Alors la fonction `transposee` possède la propriété suivante :

$$\texttt{transposee}\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \texttt{transposee}(A_{11}) & \texttt{transposee}(A_{21}) \\ \hline \texttt{transposee}(A_{12}) & \texttt{transposee}(A_{22}) \end{array}\right)$$

Notez bien que, dans les deux cas, vous avez accès à la représentation interne des matrices, c'est-à-dire que, comme pour le devoir #3, vous devez écrire la mise en oeuvre (corps de la ressource) de l'opération indiquée. Vous devez aussi supposer que les opérations fournies sont, elles aussi, les plus parallèles possibles : elles s'exécutent donc en temps $\Theta(1)$ (avec n^2 processeurs).

Pour chacun des algorithmes, vous devez présenter une analyse (asymptotique) : temps d'exécution, nombre de processeurs, travail et coût. Dans chaque cas, vous devez aussi indiquer si l'algorithme est optimal en travail ou en coût. Dans le cas de l'algorithme récursif, vous devez évidemment présenter les équations de récurrence appropriées (qui caractérisent le temps d'exécution de l'algorithme ainsi que le travail effectué) et la solution asymptotique résultante.

```
# Déclaration du type Matrice.
type Matrice = [*][*]int;


op mkMatrice( int n, int cs[*][*] )
              returns ptr Matrice r
# RÔLE
#    Construit une matrice n × n à partir d'un tableau de coefficients.
# PRECONDITION
#    n est une puissance de 2.
#    cs est une matrice n×n.
# POSTCONDITION
#    Les coefficients de la matrice r sont ceux spécifiés par le tableau cs.
# ANALYSE
#    S'exécute en temps Θ(1) avec un nombre de processeurs égal au nombre
#    d'éléments de la matrice.


op mkMatrice22( ptr Matrice m11, ptr Matrice m12,
                ptr Matrice m21, ptr Matrice m22 )
                returns ptr Matrice r
# RÔLE
#    Construit une matrice n × n à partir des quatre (4) sous-matrices n/2 × n/2 indiquées.
# PRECONDITION
#    Les matrices (carrées) m11, m12, m21 et m22 sont toutes de même taille.
# POSTCONDITION
#    r contient les sous-matrices indiquées~:
#      r = ( (m11, m12), (m21, m22) )
# ANALYSE
#    S'exécute en temps Θ(1) avec un nombre de processeurs égal au nombre d'éléments.


op sousMatrice( ptr Matrice p, int l1, int l2, int c1, int c2 )
                returns ptr Matrice r
# RÔLE
#    Extrait de la matrice p une sous-matrice spécifiée par les bornes indiquées.
# PRECONDITION
#    Les index indiqués dénotent une sous-matrice carrée, i.e., c2-c1 = l2-l1
# POSTCONDITION
#    r = p[l1:l2, c1:c2]
# ANALYSE
#    S'exécute en temps Θ(1) avec (l2-l1+1)×(c2-c1+1) processeurs.
```

Figure 1: Opérations de manipulations de matrices

2. Algorithme PRAM (choix) (15 points)

Soit l'algorithme suivant, un algorithme de style PRAM écrit en pseudo-MPD (des instructions ordinaires plutôt que des appels de sous-routine sont utilisées dans les `co`) :

```
procedure proc( int A[*], int n ) returns int nb
# PRECONDITION
#   SOME( k: nat SUCH THAT k >= 0 :: n = 2**k )
{
  int T[n];
  co [i = 1 to n]
    T[i] = 0;
  oc

  for [i = 1 to lg(n, 2)] {
    int dist = 2**(i-1);
    co [j = dist to n by 2*dist]
      T[j+dist] = T[j] + T[j+dist] + int(A[j] > A[j+1]);
    oc
  }
  nb = T[n];
}
```

a. Que fait cet algorithme? Expliquez brièvement son fonctionnement.

b. Quelles sont les caractéristiques de cet algorithme (temps d'exécution, nombre de processeurs, coût et travail)?

c. Quel type de modèle PRAM est utilisé (EREW, CREW, CRCW)? S'il s'agit d'un algorithme CW, quel type de résolution de conflit doit être utilisé?

d. Est-ce que cet algorithme PRAM est aussi valide dans le modèle MPD d'exécution (avec entrelacement arbitraire de l'exécution des diverses instances d'une instruction `co`)?

3. Algorithmes d'exploration (choix) (15 points)

Vous disposez de n objets, numérotés de 1 à n, que vous voulez vendre à exactement n acheteurs, numérotés de 1 à n. Chaque acheteur est disposé à payer un certain prix pour chacun des objets, prix qui peut varier selon les acheteurs.

a. Supposons que chaque acheteur puisse vous acheter *exactement un objet*. Un premier problème consiste donc à attribuer à chaque acheteur exactement un objet de façon à maximiser la somme totale que vous recevrez, des n acheteurs, pour vos n objets. Décrivez l'arbre des solutions pour un tel problème et analysez le plus précisément possible sa taille, pour un n donné.

b. Supposons maintenant que chaque acheteur puisse vous acheter *un nombre quelconque d'objets* (aucun objet, un objet ou plusieurs objets). Un deuxième problème consiste alors à attribuer à chaque acheteur un certain nombre d'objets de façon à maximiser la somme totale que vous recevrez, des n acheteurs, pour vos n objets.

   Expliquez comment et pourquoi ce deuxième problème, dans lequel vous pouvez vendre autant d'objets que vous le désirez à un acheteur, peut être utilisé *pour le calcul d'une borne* dans un algorithme de type *branch-and-bound* pour résoudre le premier problème décrit au point a.

c. Faites fonctionner un algorithme de type *branch-and-bound* avec parcours en *profondeur* et basé sur la borne décrite au point b. sur les données suivantes ($n = 4$) :

$i \backslash j$	*Objet* 1	*Objet* 2	*Objet* 3	*Objet* 4
Acheteur 1	15	11	8	16
Acheteur 2	14	11	10	5
Acheteur 3	5	3	7	15
Acheteur 4	8	12	20	10

   Dessinez la partie de l'arbre des solutions que l'algorithme va parcourir, en indiquant pour chaque sommet la borne calculée en ce sommet, la meilleur solution connue en arrivant sur ce sommet, et la meilleure solution connue en quittant ce sommet. Indiquez aussi clairement l'ordre de visite des divers sommets.