INF5171 — Programmation parallèle
Examen intra (Hiver 2011)

Durée: 18h00 – 21h00 **Documentation :** Documentation personnelle autorisée.

Nom: _____

Code permanent: | | | | | | | | | | | | |

1	2	3	4	5	Total
/10	/10	/10	/10	/20	/60

- L'examen comporte cinq (5) questions, comptant respectivement pour 10, 10, 10, 10 et 20 points, pour un total sur 60.
- Pour les problèmes demandant d'écrire du code MPD, vous devez utiliser une notation semblable à MPD, mais pas nécessairement identique. Entre autres, dans le cas des instructions `co`, vous pouvez utiliser une notation *moins stricte* que la notation acceptée par le compilateur. Par exemple, 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
```

  Dans le cadre de l'examen, *cette notation (*sans procédure auxiliaire*) sera acceptée* — en fait, *n'importe quelle* séquence de code dans un `co` sera acceptée, y compris des instructions `for` ou `if`. Toutefois, n'abusez pas de cette possibilité et, si cela vous semble utile, vous pouvez évidemment introduire une ou des procédures auxiliaires. (Pas besoin non plus de respecter l'ordre strict des déclarations — déclaration d'une procédure avant sa première utilisation.)
- Si nécessaire, vous pouvez supposer qu'il existe, en MPD, une instruction `critical` (semblable à celle d'OpenMP) qui assure que l'instruction qui suit est exécutée de façon exclusive et atomique, par exemple :

```
critical { x += 1; y += x; }
```

- Si nécessaire, vous pouvez utiliser, *sans les définir*, les procédures suivantes :

```
# Fonctions pour le calcul des bornes d'une tranche.
procedure inf( int i, int n, int nbProcs ) returns int r
procedure sup( int i, int n, int nbProcs ) returns int r

# Operations pour l'initialisation et manipulation du sac de taches.
procedure initSacDeTaches( int bi, int bs )
procedure obtenirTache( res int i, res int j ) returns bool disponible
```

1. Vrai ou faux (10 pts)

Pour chacune des affirmations suivantes, dites si elle est *vraie* ou *fausse* et justifiez brièvement votre réponse. Une réponse correcte mais sans justification ne vous donnera qu'une partie des points (0.5 sur 2.0).

a. Un programme est dit «SPMD» lorsque, à chaque instant, tous les processus concurrents formant le programme exécutent la même instruction. De tels programmes sont donc parfaits pour des machines SIMD plutôt que MIMD.

b. Des processus MPD activés de façon statique — avec des déclarations `process` — sont uniquement employés lorsqu'on utilise une stratégie d'allocation statique des tâches aux *threads*.

c. Lorsqu'on veut traiter une série d'éléments, par exemple, les diverses lignes d'un tableau, on utilise une distribution cyclique des lignes entre les *threads* dans le but d'éviter les interférences de lecture/écriture entre les *threads*, et donc accroitre ainsi le parallélisme.

d. Soit une machine possédant p processeurs. Si on mesure l'accélération relative d'un programme, alors la meilleure accélération qu'on obtient en pratique est au plus une accélération de p.

e. Dans un programme parallèle avec sac de tâches, le prochain *thread* qui devient libre va obtenir la plus ancienne tâche, i.e., celle ayant été ajoutée dans le sac il y a le plus longtemps.

2. Parallélisme récursif (10 pts)

Soit la constante et les types suivants qui permettent de définir des *arbres n-aires complets* :

```
const int NBE = ...;      # Nb d'enfants pour les NOEUDs internes.

type SorteNoeud = enum( FEUILLE, NOEUD );

type Arbre = ptr ArbreRec;
type ArbreRec = rec(
                  int valeur;
                  SorteNoeud sorte;
                  Arbre enfants[NBE];
                );

# INVARIANT
#   sorte == FEUILLE ⇒ enfants[i] == null (pour 1 ≤ i ≤ NBE)
#   sorte == NOEUD   ⇒ enfants[i] != null (pour 1 ≤ i ≤ NBE)
```

L'invariant précise ce qu'on entend par un **arbre n-aire complet** : soit on a une `FEUILLE`, auquel cas tous les pointeurs vers les enfants sont `null`, soit on a un `NOEUD` avec enfants, auquel cas tous les `NBE` sont présents, et *aucun des pointeurs n'est* `null`.

Soit la fonction suivante qui calcule la somme des `valeur`s associées aux différents noeuds de l'arbre — tant les noeuds internes que les feuilles, puisqu'on suppose que le champ `valeur` est correctement défini pour tous les noeuds :

```
procedure somme( Arbre a ) returns int laSomme
{
  laSomme = a^.valeur;   # Valable tant pour FEUILLE que pour NOEUD.

  if (a^.sorte == NOEUD) {
    for [i = 1 to NBE] {
      laSomme += somme( a^.enfants[i] );
    }
  }
}
```

Écrivez une version *parallèle* de cette fonction, utilisant du parallélisme récursif (avec la signature indiquée). Cette fonction récursive et parallèle doit générer des *threads*, mais en s'assurant *qu'il n'y ait jamais plus que* `nbThreads` *actifs à un instant donné.* On suppose que `nbThreads` est une variable globale, accessible à toutes les procédures et fonctions auxiliaires.

Indice : Introduisez une fonction auxiliaire qui reçoit en argument le nombre de *threads* encore disponibles, puis qui diminue cette valeur lors des appels récursifs.

```
procedure somme( Arbre a ) returns int laSomme
```

3. Manipulation de fractions à grande précision (10 pts)

Soit une fonction `nbOccurrences` qui détermine le nombre d'occurrences d'un chiffre *décimal* `c` (entre 0 et 9) dans la représentation décimale normalisée d'une fraction `f` :

```
op nbOccurrences( Fraction f, int c ) returns int nb
# PRECONDITION
#   0 ≤ c ≤ 9
#   B = 100
# POSTCONDITION
#   nb = nombre d'occurrences du chiffre décimal c dans normaliser(f)
```

Plus précisément, cette fonction retourne le nombre de fois où le chiffre `c` apparaitrait dans la représentation textuelle (normalisée) de `f` si on produisait cette représentation, mais sans la produire explicitement. Pour réaliser cette fonction, *il ne s'agit donc pas* de faire appel à la fonction `toString` et ensuite de faire du *pattern-matching* ; il faut plutôt analyser la représentation interne de `f`, que vous pouvez évidemment normaliser.

Voici un cas de test simple pour cette fonction :

```
procedure testerDiviser4a()
{ nommerCasDeTest( "testerDiviser4a" );

  assertIntEquals( 0, nbOccurrences( NaN, 3 ) );
  assertIntEquals( 2, nbOccurrences( fraction( "1 / 223" ), 2 ) );
  assertIntEquals( 6, nbOccurrences( fraction( "1122334455 / 22334455220" ), 2 ) );
  assertIntEquals( 4, nbOccurrences( fraction( "100/ 2003" ), 0 ) );
}
```

Écrivez une version *parallèle à granularité très fine* (la plus parallèle possible) de cette fonction.

Suggestions :

- Pour simplifier, vous pouvez supposer que le base `B` est fixe et est égale à 100 — donc pas besoin de traiter une base arbitraire. En fait, vous pouvez utiliser la fonction auxiliaire suivante :

  ```
  procedure nbOccsChiffre( int chiffreB, int c ) returns int nb
  # PRECONDITION: B = 100; 0 ≤ chiffreB < B; 0 ≤ c ≤ 9
  {
    nb  = int( chiffreB % 10 == c );   # Retourne 1 si vrai, 0 sinon.
    nb += int( chiffreB / 10 == c );
  }
  ```

- Utilisez du pseudo-MPD, notamment `critical`!
- Attention au *cas spécial* illustré par la dernière assertion du cas de test!
- N'oubliez pas de `normaliser` au préalable.

```
proc nbOccurrences( f, c ) returns nb
```

4. Programmes OpenMP/C (10 pts)

a. On veut trouver une position où un élément `valCherchee` apparaît dans un tableau `elems` (avec `n` éléments). L'élément peut apparaître plusieurs fois, mais une seule des positions est retournée. Si l'élément n'est pas présent, la position retournée est `-1`.

Pour chacun des extraits de code ci-bas, indiquez si la procédure produirait ou non le bon résultat et justifiez brièvement votre réponse — notamment, même si la réponse produite est bonne, est-ce une bonne stratégie?

Le contexte englobant chacun des extraits de code est le suivant :

```
void trouverPos( int elems[], int n, int valCherchee, int *pos )
{
  *pos = -1;
  ... <extrait de code indiqué plus bas>...
}
```

(i)
```
#pragma omp parallel
{
  for( int i = 0; i < n; i++ ) {
    if ( elems[i] == valCherchee )
      *pos = i;
  }
}
```

(ii)
```
#pragma omp parallel for
for( int i = 0; i < n; i++ ) {
  if ( elems[i] == valCherchee )
    #pragma omp critical
    *pos = i;
}
```

(iii)
```
#pragma omp parallel for
for( int i = 0; i < n; i++ ) {
  if ( elems[i] == valCherchee )
    *pos = i;
}
```

b. Soit le fragment de code suivant, qui permet de calculer l'ensemble de Mandelbrot sur une matrice `N` par `N`, où la fonction `couleur` est la même que dans le labo #3 :

```
void calculerCouleurs( Couleur couleurs[N,N], int maxIterations )
{

  for( int i = 0; i < N; i++ ) {

    for( int j = 0; j < N; j++ ) {

      couleurs[i,j] = couleur( borneInf+i*delta, borneInf+j*delta, maxIterations );

    }

  }

}
```

Quelle(s) directive(s) OpenMP faudrait-il indiquer pour obtenir le meilleur temps d'exécution possible pour ce programme? Justifiez brièvement votre réponse.

5. Procédures de traitement d'images de bas niveau (20 pts)

Description du problème

On veut faire du traitement d'images, *(très!) simple*, de bas niveau. On considére qu'une image est un tableau à deux dimensions, de taille `NBL` par `NBC` (nb. lignes par nb. colonnes), formé de pixels et où chaque pixel est un simple entier positif indiquant un *niveau de gris* :

```
# Définition d'un Pixel noir-gris-blanc
type Pixel = int;       # ALL( p: Pixel :: NOIR <= p <= BLANC )

const Pixel NOIR  = 0;
const Pixel BLANC = 255;

type Image = [NBL, NBC] Pixel;
```

Étant donné une image, on veut définir deux opérations :

a. Une opération de *seuillage* (*thresholding*) : La procédure parcourt chacun des pixels de l'image et si la valeur du pixel est (strictement) inférieure à un certain `seuil`, alors le pixel devient `NOIR`.

b. Une opération de *lissage* (*smoothing*) : La procédure parcourt chacun des pixels *intérieurs* de l'image et si *le nombre de voisins immédiats* du pixel qui sont (strictement) inférieurs à un certain `seuil` est *plus grand ou égal* à `nbVoisins`, alors le pixel devient `NOIR`. Les pixels sur le pourtour de l'image (i.e., `img[1,*]`, `img[NBL,*]`, `img[*,1]` et `img[*,NBC]`) sont laissés *inchangés*.

Dans les deux cas, la matrice 2D de l'image est modifiée par la procédure. On peut donc spécifier les signatures suivantes pour ces deux opérations, où dans chaque `POSTCONDITION`, `img` dénote le contenu de la matrice *avant* l'appel alors que `img'` dénote le contenu de l'image *après* l'appel, et lorsque rien n'est spécifié alors c'est que le pixel est inchangé :

```
op seuillage( ref Image img, Pixel seuil );
# POSTCONDITION
#   POUR 1 ≤ i ≤ NBL, 1 ≤ j ≤ NBC ::
#     img[i,j] < seuil ⇒ img'[i,j] = NOIR

op lissage( ref Image img, Pixel seuil, int nbVoisins );
# POSTCONDITION
#   POUR 1 < i < NBL, 1 < j < NBC ::
#     (nb de voisins immédiats de img[i,j] < seuil) ≥ nbVoisins  ⇒ img'[i,j] = NOIR
```

Les *voisins immédiats* d'un point intérieur d'index `[i,j]` sont les huit (8) points d'index suivants : `[i-1,j-1]`, `[i-1,j]`, `[i-1,j+1]`, `[i,j-1]`, `[i,j+1]`, `[i+1,j-1]`, `[i+1,j]` et `[i+1,j+1]`.

Voici deux cas de tests illustrant l'effet de ces opérations, sur des images avec 4 lignes et 5 colonnes :

```
img1 = ((1, 1, 0, 1, 3), (5, 0, 8, 3, 2), (2, 0, 2, 7, 1), (0, 2, 1, 0, 3));
img2 = ((0, 0, 0, 0, 3), (5, 0, 8, 3, 0), (0, 0, 0, 7, 0), (0, 0, 0, 0, 3));

seuillage( img1, 3 );

assertImageEquals( img2, img1 );


img1 = ((1, 1, 0, 0, 0),
        (8, 1, 1, 2, 2),
        (3, 2, 1, 1, 3),
        (6, 2, 2, 1, 7));

img2 = ((1, 1, 0, 0, 0),
        (8, 0, 0, 0, 2),
        (3, 2, 0, 1, 3),
        (6, 2, 2, 1, 7));

lissage( img1, 2, 4 );

assertImageEquals( img2, img1 );
```

Ce que vous devez faire

a. Écrivez, en MPD/pseudo-MPD, la procédure `seuillage` en utilisant du *parallélisme de résultat de granularité grossière.*

   Plus précisément, votre procédure devra respecter les contraintes suivantes :

   - La répartition des éléments entre les divers *threads* doit se faire *par colonnes*, et non par lignes — en d'autres mots, tous les éléments d'une colonne donnée doivent être traités par un même *thread.*
   - La répartition des diverses colonnes à traiter doit se faire *de façon cyclique* entre les divers *threads.*

   Vous pouvez supposer que le nombre de *threads* à créer est spécifié par la variable globale `nbThreads`.

b. Écrivez, en MPD/pseudo-MPD, la procédure `lissage` en utilisant du *parallélisme de résultat de granularité fine*, où chaque colonne représente une tâche indépendante et où chaque *thread* traite une et une seule tâche.

   Remarque : Attentions aux interférences entre les *threads.*

```
proc seuillage( img, seuil )
```

```
proc lissage( img, seuil, nbVoisins )
```