INF5171 — Programmation concurrente et parallèle
Examen intra (Automne 2012)

Durée: 9h30 – 11h00 **Documentation :** Documentation personnelle autorisée.

Nom: _____

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

1	2	3	Bonus	Total
/5	/10	/15	/ 5	/30

- L'examen comporte trois (3) questions, comptant respectivement pour 5, 10 et 15 points, pour un total sur 30. L'examen comporte aussi une *question bonus*, optionnelle, qui sera corrigée de façon plus stricte. (Il est donc possible d'obtenir une note > 30!)
- Pour les problèmes demandant d'écrire du code MPD, vous pouvez utiliser, pour les instructions `co`, une notation *moins stricte* que celle 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
```

  Pour 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 `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 — i.e., déclaration avant la première utilisation.)
- Si nécessaire, vous pouvez utiliser, *sans les définir*, les fonctions suivantes :

```
# Fonctions pour le calcul des bornes d'une tranche.
#  k: Numero du thread
#  n: Nombre <<d'elements>> a repartir entre les threads
#  nbThreads: Nombre de threads entre lesquels repartir les <<elements>>

procedure inf( int k, int n, int nbThreads ) returns int r;
procedure sup( int k, int n, int nbThreads ) returns int r;
```

1. Vrai ou faux (5 pts)

Pour chacune des affirmations suivantes, dites si elle est *vraie* ou *fausse*.

a. La variable d'environnement `MPD_PARALLEL` permet de sélectionner le nombre de processeurs qui seront utilisés pour exécuter un programme MPD.

b. Lorsqu'on utilise l'approche diviser-pour-régner récursive, si on veut pouvoir exploiter le parallélisme, il faut nécessairement un langage qui permet de créer et activer un nombre arbitraire de *threads*, et ce de façon dynamique (i.e., durant l'exécution).

c. Lorsqu'on veut traiter les diverses lignes d'une matrice, si le travail requis par chaque ligne est équivalent, alors on peut utiliser une allocation statique des lignes aux différents *threads*, et l'allocation peut se faire par groupe de lignes adjacentes ou de façon cyclique.

d. Sur une machine à p processeurs, la meilleure stratégie pour obtenir un programme avec la meilleure accélération absolue est de décomposer le problème en exactement p sous-problèmes indépendants.

e. L'algorithme suivant, exprimé en pseudo-MPD, permet de faire la somme de deux vecteurs `a` et `b` (de taille `n` > 2) et de mettre le résultat dans `c` (taille `n`) :

```
co [i = 1 to n by 2]  c[i] = a[i] + b[i]
// [j = 2 to n by 2]  c[j] = a[j] + b[j]
oc
```

Le coût de cet algorithme est $O(n \times n) = O(n^2)$.

2. Sac de tâches (10 pts)

Nous avons vu en cours (exercice) un exemple de programme parallèle utilisant un sac de tâches où de nouvelles tâches étaient générées et ajoutées *dynamiquement* (en cours d'exécution) au sac de tâches.

Supposons que pour la manipulation d'un tel sac de tâches, on ait la constante et les opérations présentées dans l'Extrait de code MPD 1. On suppose que l'information définissant une tâche est constituée de deux entiers positifs, sauf pour la valeur spéciale `FIN_DES_TACHES` qui indique *qu'il ne reste plus aucune tâche*. Supposons aussi que chacune des opérations s'exécute de façon atomique appropriée.

Soit la procédure `executerTaches` présentée dans l'Extrait de code MPD 2, qui utilise la routine `FA` vue en cours (*Fetch-and-Add*), ainsi que les procédures `trier` et `mystere`.

```
# Indicateur pour signaler qu'il ne reste plus de taches a executer.
const int FIN_DES_TACHES = -1;

# Initialisation du sac de taches.
procedure initSac()
{ ... }

# Ajout d'une tache dans le sac.
procedure ajouterTache( int i, int j )
{ ... }

# Retrait d'une tache.
#
# Si aucune tache n'est disponible, la fonction bloque, i.e., l'appellant
# est suspendu (mis en attente), et ce jusqu'a ce qu'une tache soit ajoutee.
#
# La procedure s'assure que le dernier thread actif ajoute dans le
# sac les valeurs FIN_DES_TACHES appropriees pour reactiver les threads
# en attente et assurer la terminaison correcte du programme.
procedure obtenirTache( res int i, res int j )
{ ... }
```

Extrait de code MPD 1: Constante et procédures pour un sac de tâches dynamique avec retrait et ajout de tâches.

```
procedure trier( ref int a[*], int n )
{ ... Methode de tri simple... }

procedure FA( ref int x, int incr ) returns int r
# FA = FetchAndAdd (atomique)
# EFFET: < r = x; x += incr; >
{ ... }

procedure executerTaches( ref int a[*], int b, ref int c[*], ref int d )
{
  const int S = ...;
  assert( S >= 1, "Valeur invalide pour S!?" );

  int i, j;
  obtenirTache( i, j )
  while( i != FIN_DES_TACHES ) {
    if ( j - i < S ) {
      for [k = i to j st a[k] == b] {
        int l = FA(d, 1);
        c[l+1] = k;
      }
      obtenirTache( i, j );
    } else {
       int mid = (i + j) / 2;
       ajouterTache( mid+1, j );
       j = mid;
    }
  }
}

procedure mystere( int a[*], int b, res int c[*], res int d )
{
  initSac();
  ajouterTache( 1, ub(a) );

  d = 0;
  co [i = 1 to nbThreads]
    executerTaches( a, b, c, d )
  oc

  trier( c, d );
}
```

Extrait de code MPD 2: Procédures et fonction `trier`, `FA`, `executerTaches` et `mystere`.

Soit l'appel suivant à la procédure `mystere` :

```
int x[10] = (2, 3, 4, 5, 2, 2, 4, 5, 34, 2);
int y[10];
int z;

mystere( x, 2, y, z );
```

a. Qu'est-ce qui sera calculé par cet appel à `mystere` — c'est-à-dire quelle sera la valeur des variables *modifiées* par `mystere`?

b. De façon plus générale, décrivez ce que fait la procédure `mystere` (QUOI?)?

c. Expliquez brièvement de quelle façon (COMMENT?) fonctionne cette procédure. En quoi est-ce semblable, ou différent, d'autres approches de création de parallélisme vues en cours?

d. Que se passe-t-il si `S > ub(a)`?

3. Procédure de traitement d'images de bas niveau (15 pts)

Description du problème

On veut faire du traitement d'images comme dans le laboratoire #3. Plus précisément, étant donné une image, on veut définir une opération `histogramme` qui détermine *le nombre de points de chacune des couleurs*. Voir l'Extrait de code MPD 3 pour les types et la signature des opérations publiques, dont les opérations `histogramme`, `nbLignes` et `nbColonnes`.

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

  # Couleurs extremes pour pixels noir-gris-blanc.
  const Pixel NOIR  = 0;
  const Pixel BLANC = 255;


  ################################################################
  # Definition d'un type image = pixmap.
  ################################################################
  type Image = ptr [*,*] Pixel;

  op image( int points[*,*] ) returns Image img;

  op nbLignes  ( Image img ) returns int;
  op nbColonnes( Image img ) returns int;


  ################################################################
  # Definition d'un type et d'une operation pour histogrammes.
  ################################################################
  type Histogramme = [NOIR:BLANC] int;

  op histogramme( Image img ) returns Histogramme h;

body Images()
  ...
end
```

Extrait de code MPD 3: Constante, types et opération pour histogramme d'images.

L'extrait de code MPD 4 donne un exemple d'utilisation, spécifié à l'aide d'un cas de test MPDUnit.

```
procedure testerHistogramme1()
{ nommerCasDeTest( "testerHistogramme1" );

  Image img = image( ((0, 1, 0, 1, 3),
                      (5, 0, 0, 3, 2),
                      (2, 0, 2, 5, 1),
                      (0, 2, 1, 0, 3)) );

  Histogramme h = histogramme( img );

  assertIntEquals( 7, h[0] );
  assertIntEquals( 4, h[1] );
  assertIntEquals( 4, h[2] );
  assertIntEquals( 3, h[3] );
  assertIntEquals( 0, h[4] );
  assertIntEquals( 2, h[5] );
  for [i = 6 to BLANC] {
    assertIntEquals( 0, h[i] );
  }
}
```

Extrait de code MPD 4: Cas de test pour l'opération `histogramme`.

Ce que vous devez faire

Écrivez (page suivante), en MPD/pseudo-MPD, la fonction `histogramme` en utilisant du *parallélisme de données de granularité grossière*.

Plus précisément, votre procédure devra respecter les contraintes suivantes :

- Vous devez compléter et utiliser la fonction auxiliaire `histoPartiel` (séquentielle) qui vous est fournie.
- Le nombre de *threads* à créer est spécifié par la variable globale `nbThreads`. Ces *threads* sont créés de façon dynamique, donc avec une instruction `co`.
- La répartition des pixels entre les divers *threads* doit se faire *par groupes de lignes adjacentes*.
- Vous ne pouvez utiliser ni l'instruction `FA`, ni des sémaphores.
- Pour le nombre de lignes/colonnes, utilisez les opérations publiques fournies par le module `Image`.
- Si approprié (?!), introduisez une ou des routines (fonction ou procédure) auxiliaires.

Bonus (5 pts) [Répondez au verso de la présente page] Écrivez, en MPD/pseudo-MPD, une version de la fonction `histogramme` qui utilise du *parallélisme de résultat à granularité (très) fine* — toujours sans utiliser `FA`.

```
# Cette fonction construit un histogramme, partiel, pour la partie de l'image
# comprise entre les lignes l1 a l2 et les colonnes c1 à c2.
procedure histoPartiel( Image img, int l1, int l2, int c1, int c2 )
          returns Histogramme h
{
  h = ([BLANC-NOIR+1] 0);

  for [i = l1 to l2, j = c1 to c2] {
    ### A COMPLETER!




  }
}

proc histogramme( img ) returns h
### A COMPLETER!
```