INF5170 — Programmation parallèle
Examen intra (Automne 2008)

Durée: 13h30 – 16h30 (3 heures) **Documentation autorisée:** Toute documentation personnelle.

Remarques :

- L'examen comporte cinq (5) questions, comptant respectivement pour 8, 5, 15, 12 et 10 points, pour un total sur 50.

  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 > 50.)

- Vous devez écrire vos algorithmes en utilisant une notation la plus semblable possible 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
  ```

  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.

- Sauf pour la question 5, vous pouvez utiliser, sans les définir, les procédures et fonctions suivantes tirées du devoir 1 :

  ```
  # 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. Trace d'exécution et instructions atomiques en pseudo MPD (8 pts)

Soit le segment de code suivant :

```
int x = 0;

co  < await( x != 0 ) { x = x - 2; } >
//  < await( x != 0 ) { x = x - 3; } >
//  < await( x == 0 ) { x = x + 5; } >
oc
```

[4] a) Est-ce que ce programme termine? Si oui, quelles sont alors les valeurs possibles pour `x`? Si non, pourquoi?

[4] b) Même question, mais cette fois en remplaçant chaque mot «`await`» par le mot «`if`» — en conservant les crochets d'instruction atomique.

2. Instructions atomiques en pseudo MPD et section critique (5 pts)

On a vu dans le cours l'utilisation de la pseudo-instruction `await` pour assurer l'accès à une section critique par un groupe de processus, et ce à l'aide d'une variable booléenne `lock`.

Les processus suivants (Extrait de code MPD 1) définissent une variante de l'accès à une section critique, où l'un des processus, `unProc0`, joue un rôle différent des autres processus, et où les autres processus (`unProc[i]`) ont aussi un comportement différent de celui vu dans le protocole de base. Donc, le protocole d'entrée/sortie dans la section critique diffère de celui vu en cours, et est spécifique à chaque type de processus.

```
const int VAL = ...;    # VAL est toujours positif, i.e., VAL > 0.

bool  x0 = false;
int   y  = 0;

process unProc0 {
  while (true) {
    < await( y == 0 ) { x0 = true; } >
    ... section critique ...
    x0 = false;
    ... section non critique ...
  }
}

process unProc[i = 1 to n] {
  while (true) {
    < await( ~x0 & y < VAL ) { y = y + 1; } >
    ... section critique ...
    < y = y - 1; >
    ... section non critique ...
  }
}
```

Extrait de code MPD 1: Accès à une section critique par un groupe de processus, dont un qui joue un rôle spécial.

Décrivez ce que permet exactement ce protocole d'accès à la section critique. En quoi cela diffère-t-il du protocole de base simple vu au chapitre 3? Quelle est la différence entre le comportement des deux types de processus? Que se passe-t-il dans le cas où `VAL < n`? Lorsque `VAL >= n`?

Bonus (5 pts) Une propriété important d'un protocole d'accès à une section critique est de permettre *l'entrée éventuelle* dans la section critique de n'importe quel processus qui le désire — en d'autres mots, un processus qui tente d'entrer la section critique devrait éventuellement pouvoir y parvenir. Peut-on imaginer une situation où ce ne serait pas le cas pour ce protocole, i.e., où un processus tenterait d'entrer dans la section critique mais pourrait ne pas y arriver? Si oui, quelle est cette situation? Si non, pourquoi est-ce impossible?

3. Calcul parallèle d'une fonction chaotique (15 pts)

Une *fonction chaotique* est une fonction simple et déterministe, généralement calculée par récurrence, qui peut produire des résultats très différents même pour des points rapprochés les uns des autres — on parle alors de «sensibilité aux conditions initiales» (terme introduit par la *théorie du chaos*).[1]

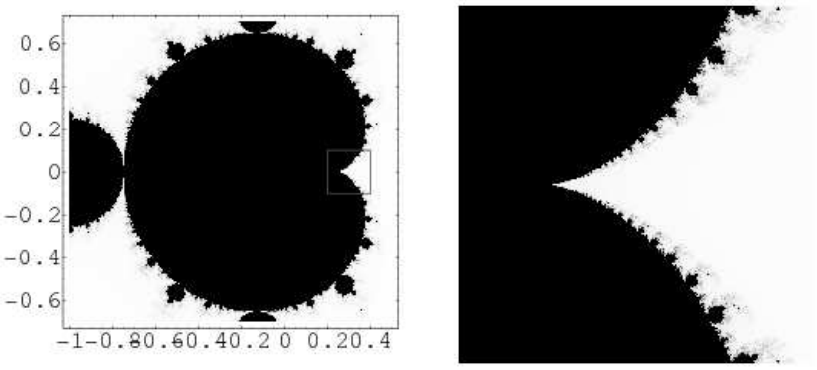

Figure 1: Une représentation graphique de l'ensemble de Mandelbrot.

Un exemple d'une *fonction chaotique* est celle qui permet de produire l'ensemble de Mandelbrot, illustré graphiquement à la figure 1 — il s'agit d'une *fractale* : la figure de droite, qui est un agrandissement de la petite partie encadrée de la figure de gauche, montre une propriété typique des fractales, à savoir, *l'auto-similarité* (on retrouve la figure de départ lorsqu'on agrandit un morceau).

Une particularité de la fonction permettant de calculer l'ensemble de Mandelbrot est que le temps requis pour déterminer si un point fait partie de l'ensemble (point coloré en noir) varie d'un point à un autre, et ce même pour des points très près les uns des autres.

L'Extrait de code MPD 2 présente divers constantes, types et fonctions pouvant être utilisés pour une version simplifiée du problème du calcul de l'ensemble de Mandelbrot. Plus spécifiquement, la fonction `couleur` permet de déterminer, pour un point `pt` (type `Point`) donné, la couleur (`BLANC` ou `NOIR`) associée à ce point. Ainsi, la couleur est `BLANC` dès qu'on détecte que `longueur(xk, yk) >= DISTANCE`. Par contre, la couleur du point est `NOIR` si après avoir effectué un certain nombre d'itérations — spécifié par `maxIterations` — la valeur `longueur(xk, yk)` est toujours restée inférieure à `DISTANCE`.

[1] `http://fr.wikipedia.org/wiki/Theorie_du_chaos`

```
# Taille de la grille de points a traiter: une constante, pour simplifier
# le code et eviter de manipuler des tableaux alloues dynamiquement.
const int  N = ...


# Quelques types, pour simplifier l'interface des fonctions.
type Couleur  = enum( BLANC, NOIR );
type Couleurs = [N,N]Couleur;

type Point    = rec( real x; real y );
type Points   = [N,N]Point;


# Fonction auxiliaire pour calculer la longueur du vecteur (x, y).
procedure longueur( real x, real y ) returns real d
{ d = sqrt( x**2 + y**2 ) }


# Valeur specifique a la region qu'on veut explorer: une constante,
# pour simplifier le code.
const real DISTANCE = ...

# Fonction pour determiner la couleur d'un point pt.
procedure couleur( Point pt, int maxIterations ) returns Couleur c
{
  real xk = pt.x;
  real yk = pt.y;

  int nbIterations = 0;
  while( longueur(xk, yk) < DISTANCE & nbIterations < maxIterations ) {
    real xk1 = xk * xk - yk * yk + pt.x;
    real yk1 =  2 * xk * yk      + pt.y;

    xk = xk1;
    yk = yk1;
    nbIterations += 1
  }
  if ( nbIterations == maxIterations ) {
    c = NOIR;
  } else {
    c = BLANC;
  }
}
```

Extrait de code MPD 2: Constantes, types et fonctions pour le calcul de la couleur d'un point de l'ensemble de Mandelbrot.

On veut écrire, en MPD, une procédure parallèle qui permet de déterminer la `couleur` d'un ensemble de points spécifiés par un tableau de type `Points`. Plus précisément, la procédure possède l'interface suivante :

```
procedure calculerCouleurs( Points pts, res Couleurs couls, int maxIterations )
# PRECONDITION
#   maxIterations > 0
#   ub(pts, 1) = ub(couls, 1) = N
#   ub(pts, 2) = ub(couls, 2) = N
```

Cette fonction applique la fonction `couleur` (cf. Extrait de code MPD 2) sur chacun des points du tableau `pts` et affecte la couleur résultante à l'élément correspondant du tableau `couls`.

[10] a) Écrivez une version *parallèle* de cette procédure utilisant l'approche qui, selon vous, devrait produire *les meilleures performances* sur une machine multi-processeurs telle que la machine `arabica`.

[5] b) Justifiez brièvement l'approche que vous avez choisie, et ce en la comparant avec *au moins deux autres approches possibles* pour mettre en oeuvre, de façon parallèle, cette procédure.

4. Produit vectoriel de deux vecteurs (12 pts)

Soit deux vecteurs $u = [u_1, \ldots, u_n]$ et $v = [v_1, \ldots, v_m]$. Le *produit vectoriel* de ces deux vecteurs, noté $u \times v$, est une *matrice* r de taille $n \times m$ où $r_{i,j} = u_i * v_j$.

L'interface d'une procédure MPD pour calculer un tel produit vectoriel peut être spécifiée comme suit :

```
procedure produitVectoriel( int u[*], int n, int v[*], int m, res int r[*,*] )
# PRECONDITION
#   n = ub(u) & n > 0
#   m = ub(v) & m > 0
#   n = ub(r,1) & m = ub(r,2)
# PRECONDITION (liee a la mise en oeuvre)
#   ?          (A completer: voir plus bas)
# POSTCONDITION
#   r[i,j] = u[i] * v[j]       1 <= i <= n, 1 <= j <= m
```

[10] a) Écrivez le code MPD pour cette procédure en utilisant du *parallélisme de résultat de granularité grossière*.

Votre procédure devra respecter les *contraintes de mise en oeuvre 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 doit se faire *de façon cyclique* entre les divers *threads*.

Vous pouvez supposer que le nombre de *threads* à créer est spécifié par une constante `NB_THREADS`. Pour simplifier les calculs, vous pouvez aussi supposer que `n` et `m` sont d'une forme appropriée — par exemple, des multiples d'une certaine forme de `NB_THREADS`. Attention : Vous devez indiquer explicitement, dans une `PRECONDITION` de votre procédure, la ou les conditions que vous imposez sur `n` et/ou `m`.

[2] b) Est-ce que l'utilisation d'une répartition par tranche d'éléments *adjacents*, plutôt qu'une répartition *cyclique*, produirait un programme plus performant sur une machine multi-processeurs telle qu'`arabica`? Justifiez brièvement votre réponse.

Bonus (4 pts) Une autre répartition possible des éléments du résultat entre les divers *threads* est une allocation *par bloc* plutôt que par tranche. Alors qu'une tranche au sens vu précédemment contient tous les éléments d'une ou plusieurs lignes (ou colonnes), un bloc contient une série d'éléments adjacents, mais pas nécessairement toute la ligne ou la colonne. Par exemple, une matrice `a[10,8]` pourrait être décomposée en quatre (4) blocs : `a[1:5,1:4]`, `a[1:5,5:8]`, `a[6:10,1:4]`, `a[6:10,5:8]`. Si la procédure que vous définissez pour la sous-question a) utilise une approche de répartition par bloc, des points supplémentaires vous seront accordés.

5. Sac de tâches (10 pts)

Dans les programmes parallèles utilisant des sacs de tâches que nous avons vus en cours, le sac de tâches était initialisé au début de l'exécution du programme de façon à contenir, dès le départ, toutes les tâches possibles. En d'autres mots, une fois le sac de tâches créé, aucune nouvelle tâche n'était ajoutée en cours d'exécution. Or, dans de nombreux problèmes utilisant une approche avec sac de tâches, il est souvent utile de pouvoir ajouter de nouvelles tâches *dynamiquement* (en cours d'exécution).

Supposons que pour la manipulation du sac de tâches, on ait la constante et les opérations présentées dans l'Extrait de code MPD 3, où on suppose que l'information définissant une tâche est constituée de deux valeurs entières positives — sauf pour la valeur spéciale `FIN_DES_TACHES` — et où chacune des opérations s'exécute de façon atomique appropriée.

Soit la procédure **`executerTaches`**, écrite en *pseudo-MPD*, présentée dans l'Extrait de code MPD 4.

Soit le programme suivant, où `N` est une puissance de 2 et `K` un entier positif inférieur à `N` :

```
# Constantes.
const int N = ...;  # ASSERT( N > 0 & N est une puissance de 2 )
const int K = ...;  # ASSERT( K < N )

# Allocation et initialisation de a.
int a[N]
for [i = 1 to N] {
  a[i] = i;
}

# Utilisation du sac de taches.
initSac();
ajouterTache( 1, N );
int resultat = 0;
co [i = 1 to NB_THREADS]
  executerTaches( a, resultat, K )
oc

# Impression du resultat.
printf( "resultat = %d\n", resultat );
```

[5] a) Qu'est-ce qui sera calculé et imprimé par ce programme avec `a` tel que défini?

[5] b) Expliquez brièvement de quelle façon fonctionne cette procédure. En quoi est-ce semblable, ou différent, d'autres approches de création de parallélisme vue en cours?

```
# Indicateur special 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 au sac.
#
# La procedure s'assure que le dernier thread actif ajoute dans le
# sac des 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 3: Constante et procédures pour sac de tâches.

```
procedure executerTaches( int a[*], ref int resultat, int K )
{
  int i, j;
  obtenirTache( i, j )
  while( i != FIN_DES_TACHES & j != FIN_DES_TACHES ) {
    if ( j - i + 1 <= K ) {
      int r = 0
      for [k = i to j] {
        r += a[k]
      }
      < resultat = resultat + r; >  # Instruction pseudo-MPD
      obtenirTache( i, j );
    } else {
       int mid = (i + j) / 2;
       ajouterTache( i, mid );
       i = mid + 1;
       # j est inchange
    }
  }
}
```

Extrait de code MPD 4: Procédure `executerTaches` en pseudo-MPD.