INF5170 — Programmation parallèle
Examen intra (Hiver 2006)

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

Remarques :

- L'examen comporte cinq (5) questions, comptant respectivement pour 5, 5, 20, 10 et 10 points, pour un total sur 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
```

  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. Donc, *n'importe quelle* séquence de code dans un `co` sera acceptée, y compris des instructions complexes.
- En MPD, on a les équivalences suivantes : `int(true) == 1` ; `int(false) == 0`.
- Pour les questions sur les polynômes, *vous pouvez utiliser, sans les définir, les opérations (*`op`/`proc`*) ainsi que les procédures et fonctions auxiliaires (*`procedure`*) définies dans le devoir #1.*

---

1. Trace d'exécution et instructions atomiques en pseudo MPD (5 pts)

Indiquez ce qui sera imprimé par le segment de code suivant, écrit en pseudo MPD (MPD avec instructions `await`). Si plusieurs réponses sont possibles, indiquez-les toutes, y compris, si cela est possible, l'absence de réponse, c'est-à-dire les cas où rien n'est imprimé à cause d'un *deadlock*.

```
int x = 0, y = 1;

co  < if   ( x == 0 ) { x = x + 1; } >
//  < await( x == 0 ) { y = 2; } >
//  < x = x - y; >
oc

printf( "x = %d, y = %d\n", x, y );
```

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

On a vu durant le cours l'utilisation de la pseudo-instruction `await` pour assurer l'accès à une section critique par un groupe de processus. La solution vue en classe avait l'allure suivante, où `n` indique le nombre de processus qui vont compétitionner pour accéder à la section critique :

```
bool lock = false;

process processusAccesSC [i = 1 to n] {
  while (true) {
    < await(!lock) lock = true; >
    ... section critique ...
    lock = false;
    ... section non critique ...
  }
}
```

Ce protocole d'entrée et de sortie de la section critique à l'aide d'un verrou booléen assure qu'un seul et unique processus à la fois pourra être en train d'exécuter le code de la *section critique*.

Supposons que l'on veuille plutôt faire en sorte que plusieurs processus différents puissent être dans la section critique, en autant qu'il n'y ait jamais plus que `k` processus (donc le nombre de processus dans la *section critique* doit être inférieur ou égal à `k`, où $\mathtt{k} \leq \mathtt{n}$). De quelle façon faudrait-il modifier le code présenté ci-haut pour permettre cela?

Vous pouvez/devez utiliser des instructions en pseudo MPD, c'est-à-dire l'instruction `await` et les crochets "`<...>`", mais en autant que l'utilisation soit malgré tout de *granularité fine*. Évidemment, vous n'êtes pas obligé d'utiliser `lock` et vous pouvez introduire d'autres variables, de types autre que `bool`.

3. Parallélisme itératif à granularité fine et grossière (20 pts)

Soit $p(x)$ un polynôme de degré n :

$$p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_{n-1}x^{n-1} + a_nx^n$$

La dérivée $p'(x)$ de $p(x)$ par rapport à x est alors le polynôme suivant :

$$p'(x) = a_1 + 2a_2x + 3a_3x^2 + \ldots + (n-1)a_{n-1}x^{n-2} + na_nx^{n-1}$$

Note : Dans le cas particulier où $p(x) = a_0$, alors $p'(x) = 0$.

Soit alors l'interface suivante d'une fonction `deriver` :

```
op deriver( ptr Polynome p ) returns ptr Polynome pPrime;
# POSTCONDITION
#   pPrime représente la dérivée de p, telle que décrite plus haut.
```

a. Parallélisme à granularité fine :

   (a) Écrivez une version parallèle de la fonction `deriver`, utilisant du parallélisme itératif à *granularité fine basé sur le parallélisme de résultat.*

   (b) Donnez le temps d'exécution asymptotique ainsi que le coût de votre algorithme. Est-ce un algorithme de coût optimal?

b. Même question que la précédente (code et analyse asymptotique), mais cette fois pour du parallélisme de résultat itératif et *à granularité grossière utilisant au plus* `NB_PROCS` *processus.*

   Note : Pour simplifier le code, vous pouvez supposer que la taille du polynôme à dériver est supérieure à `NB_PROCS`. Vous pouvez aussi poser d'autres hypothèses, en autant que vous les indiquiez *de façon explicite.*

4. Parallélisme récursif (10 pts)

Soit $p(x)$ un polynôme de degré $n-1$, donc un polynôme ayant exactement n coefficients — pour simplifier, on suppose que n est une puissance de 2 :

$$p(x) = a_0+a_1x+a_2x^2+a_3x^3+\ldots+a_{\frac{n}{2}-1}x^{\frac{n}{2}-1}+a_{\frac{n}{2}}x^{\frac{n}{2}}+a_{\frac{n}{2}+1}x^{\frac{n}{2}+1}+\ldots+a_{n-2}x^{n-2}+a_{n-1}x^{n-1}$$

Une façon équivalente de représenter ce polynôme est la suivante, obtenue en factorisant l'élément $x^{\frac{n}{2}}$ de la deuxième moitié :

$$p(x) = \left[a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots + a_{\frac{n}{2}-1}x^{\frac{n}{2}-1}\right] + x^{\frac{n}{2}}\left[a_{\frac{n}{2}} + a_{\frac{n}{2}+1}x + \ldots + a_{n-1}x^{\frac{n}{2}-1}\right]$$

Ce même principe pourrait être appliqué récursivement aux deux moitiés résultantes (qui ont chacune $n/2$ coefficients, aussi une puissance de 2).

La valeur d'un polynôme $p(x)$ pour une valeur v est obtenue en substituant v pour x et en évaluant les diverses expressions arithmétiques. Par exemple, soit $p_0(x) = 2+3x+5x^2+4x^3$. Alors on aura :

$p_0(0) = 2$	$p_0(2) = 2 + 6 + 20 + 32 = 60$
$p_0(1) = 14$	$p_0(3) = 2 + 9 + 45 + 108 = 164$

Soit alors l'interface suivante d'une fonction `evaluer` :

```
op evaluer( ptr Polynome p, int v ) returns int r
# POSTCONDITION
#   r = valeur du polynôme p obtenue pour x = v, i.e., p(v)
```

a. On veut obtenir une version *parallèle récursive* de cette fonction qui se base sur le principe de division dichotomique récursive (et équilibrée) d'un polynôme indiqué plus haut (en supposant que le nombre de coefficients est une puissance de 2). Complétez la fonction auxiliaire `evaluerRec` utilisée comme suit par la mise en oeuvre (`proc`) de l'opération `evaluer` :

```
procedure evaluerRec( Polynome p, int x, int n ) returns int r
{ ... }

proc evaluer( p, x ) returns r
{  r = evaluerRec( p^, x, ub(p^) );  }
```

Indices :

- En MPD, on calcule v^k avec l'expression `v**k`.
- Utilisez des *tranches de tableau* comme argument à la fonction récursive `evaluerRec`. Rappel : le type `Polynome` est défini par «`type Polynome = [*]int;`».

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

Pour votre analyse, vous pouvez considérer l'opération d'exponentiation «`**`» comme une opération primitive, donc s'exécutant en temps $O(1)$.

5. Calcul parallèle des préfixes (10 pts)

Soit la procédure `calculerPrefixes` qui permet de calculer de façon parallèle les divers préfixes d'un vecteur `x` d'éléments relativement à un opérateur `op0` :

```
procedure op0( int x, int y ) returns int r
{ r = ...opération binaire associative sur x et y ...; }

procedure calculerPrefixes( int x[*], res int prefixes[*], int n )
# PRECONDITION
#   n ≥ 1 & n est une puissance de 2
# POSTCONDITION
#   prefixes[1] = x[1]
#   prefixes[2] = op0( x[1], x[2] )
#   prefixes[3] = op0( op0( x[1], x[2] ), x[3] )
#   etc.
```

Supposons qu'on ait à notre disposition une version de cette procédure qui soit de *coût optimal*, donc ayant les caractéristiques suivantes :

- Temps : $O(\lg\ n)$
- Nombre de processeurs : $\frac{n}{\lg\ n}$

Soit la fonction `procMystere` présentée plus bas, ainsi que la fonction `op0`, laquelle est associée à l'opérateur binaire `max`.

a. Indiquez ce que fait la fonction `procMystere`.

b. Donnez son temps d'exécution asymptotique.

c. Donnez son coût (asymptotique) et indiquez si c'est optimal ou non. Si ce n'est pas optimal, décrivez *brièvement* ce qu'il faudrait faire pour obtenir un algorithme de coût optimal.

d. Supposons que l'on remplace `max` par `min` dans `op0`. Que ferait alors la fonction `procMystere`?

```
procedure op0( int x, int y ) returns int z
{ z = max( x, y ); }

procedure procMystere( int x[*], int n, int v ) returns bool r
# PRECONDITION
#   n est une puissance de 2.
{
   int t[n];
   co [i = 1 to n]
     t[i] = int( x[i] == v );   # 1 si x[i] == v, 0 sinon.
   oc

   int prefixes[n];
   calculerPrefixes( t, prefixes, n );
   r = (prefixes[n] == 1);
}
```