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

Durée: 13h30 – 16h30 **Documentation :** Documentation personnelle (papier) autorisée.

Nom: _____

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

1	2	3	4	5	Total
/10	/10	/10	/10	/10	/50
XXX	XXX	XXX			

L'examen comporte cinq (5) questions.
Certaines d'entre elles comportent des points bonus (maximum 8 points).

---

1. Manipulation de polynômes en OpenMP/C (10 pts)

On veut définir des opérations pour des polynômes en OpenMP/C. La signature des opérations du fichier `polynomes.h` et certaines macros et fonctions du fichier `polynomes.c` sont présentées dans l'extrait de code C 1.

```
////////////////////////////////////////////
// Fichier polynomes.h
////////////////////////////////////////////
typedef struct {
  int degre;
  int* coefficients;
} Polynome;

Polynome polynome( int degre, int coefficients[] );

Polynome fois( Polynome p1, Polynome p2 );

bool egaux( Polynome p1, Polynome p2 );


////////////////////////////////////////////
// Fichier polynomes.c (extraits)
////////////////////////////////////////////
#include "polynomes.h"

// Macros et fonctions auxiliaires.
#define MIN( x, y ) ((x) < (y) ? (x) : (y))
#define MAX( x, y ) ((x) > (y) ? (x) : (y))

static Polynome allouer( int degre )
{
  Polynome p;
  p.degre = degre;
  p.coefficients = (int*) malloc( (degre+1) * sizeof(int) );
  return( p );
}
```

Extrait de code C 1: Fichier `polynomes.h` et extraits du fichier `polynomes.c`.

Complétez la mise en oeuvre des opérations présentées à la page suivante de façon **à les rendre les plus parallèles et efficaces possible.**

Pour la distribution des itérations entre les *threads*, vous devez spécifier la valeur pour `schedule` qui vous semble la plus appropriée. Rappel : La syntaxe pour `schedule` est la suivante, où `chunk` est un entier (optionnel) supérieur ou égal à 1 :

```
#omp for ... schedule( static [,chunk] )
#omp for ... schedule( dynamic [,chunk] )
```

Vous pouvez aussi supposer que la variable d'environnement `OMP_NESTED` est égale à `TRUE`.

```
Polynome polynome( int degre, int coefficients[] )
{
  Polynome p = allouer(degre);



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

    p.coefficients[i] = coefficients[i];

  }

  return( p );
}


static int coefficient( int i, Polynome p1, Polynome p2 )
{
  int expMinR1 = MAX( 0, i-p2.degre );
  int expMaxR1 = MIN( i, p1.degre );

  int c = 0;



  for( int k = expMinR1; k <= expMaxR1; k++ ) {

    c += p1.coefficients[k] * p2.coefficients[i-k];

  }

  return( c );
}


Polynome fois( Polynome p1, Polynome p2 )
{
  Polynome p = allouer( p1.degre + p2.degre );




  for( int i = 0; i <= p.degre; i++ ) {

    p.coefficients[i] = coefficient( i, p1, p2 );

  }

  return( p );
}
```

2. Programme Java «mystere» (10 pts)

Le Programme Java 1 présente une méthode `mystere` utilisant des *threads*.

```
public static int mystere( final int elems[],
                           final int x,
                           final int nbThreads ) {
    final int res[] = new int[nbThreads];

    Thread ts[] = new Thread[nbThreads];
    for ( int i = 0; i < nbThreads; i++ ) {
        final int fi = i;
        ts[i] = new Thread( new Runnable() {
                public void run() {
                    int r = 0;
                    for( int j = fi; j < elems.length; j += nbThreads ) {
                        if ( elems[j] <= x ) {
                            r += elems[j];
                        }
                    }
                    res[fi] = r;
                }} );
        ts[i].start();
    }

    int r = 0;
    for ( int i = 0; i < nbThreads; i++ ) {
        try {
            ts[i].join();
            r += res[i];
        } catch( Exception ie ){}
    }

    return r;
}
```

Programme Java 1: Une méthode Java «`mystere`».

Soit le segment de code suivant qui comporte un appel à la méthode `mystere` :

```
int r = mystere( new int[]{ 20, 90, 20, 30, 50, 80, 40, 90, 50, 60},
                 43,
                 3  );

System.out.println( "r = " + r );
```

a. Qu'est-ce qui sera imprimé par le segment de code de la page précédente?

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

c. Expliquez brièvement de quelle façon (COMMENT?) fonctionne cette procédure. Plus spécifiquement, décrivez **quelle stratégie de programmation parallèle** (vue en cours) est utilisée?

3. Mise en oeuvre parallèle de l'opération `detect` (Java) (10 pts)

On a vu en cours comment une approche «*Map/Reduce*» pouvait être réalisée en Java, et ce pour une classe `MRArray<T>` permettant de manipuler des collections d'éléments définies avec un tableau simple, classe qui mettait en oeuvre l'interface `CollectionMapReduce<T>`.

On veut ajouter une autre opération à l'interface `CollectionMapReduce` :

```
/** detect: Retourne un element de la collection qui satisfait le predicat.
  * Retourne null si aucun element ne le satisfait.
  *
  * @param p Predicat de selection
  */
 T detect( Fonction<T,Boolean> p );
```

Rappelons la définition de l'interface `Fonction` :

```
interface Fonction<T,R> {
    R appliquer( T e );
}
```

Complétez le code Java (page suivante) pour mettre en oeuvre, **de façon parallèle**, la fonction `detect`.

Contrainte de mise en oeuvre : Cette mise en oeuvre parallèle doit être une version à *granularité (très!) fine* — donc *pas besoin de travailler avec des tranches de tableau.*

Remarque : La fonction `detect` peut très bien retourner le premier élément (dans l'ordre des indices) qui satisfait la condition décrite par le prédicat `p`. Une fois cette valeur trouvée, on peut alors terminer l'exécution des *threads* encore actifs en utilisant la méthode `shutdownNow()` — inutile de poursuivre leur exécution, puisqu'une valeur a été trouvée!

```
import java.util.concurrent.*;

public class MRArray<T> implements CollectionMapReduce<T> {
    private T[] elems;

    MRArray( T elems[] ) {
        this.elems = elems;
    }

    @SuppressWarnings("unchecked")
    public T detect( final Fonction<T,Boolean> p ) {









    }
```

4. Sémaphores (MPD) (10 pts)

Soit le programme MPD suivant, qui utilise des sémaphores.

```
resource R()
  int x = 10;

  sem s1 = 1,  s2 = 0,  s3 = 0;

  process proc1 { P(s2); P(s3); P(s1); x = 2 * x; V(s1); V(s2); V(s3) }

  process proc2 { P(s1); x = x + 1; V(s1); V(s2); }

  process proc3 { P(s1); x = x - 1; V(s1); V(s3); }

  process proc4 { P(s2); P(s3); P(s1); x = x + 3; V(s1); V(s2); V(s3) }

  final         { printf( "x = %d\n", x ); }
end
```

a. Pour chacun des sémaphores `s1`, `s2` et `s3`, indiquez de quel type de sémaphore il s'agit (binaire/généralisé, exclusion mutuelle/signalisation?).

- `s1` :
- `s2` :
- `s3` :

b. Indiquez quelles sont les sorties qui peuvent être imprimées par ce programme? Si un interblocage (*deadlock*) est possible, indiquez-le aussi.

c. Est-ce que ce programme fonctionnerait quand même correctement si, dans le processus `proc4`, on intervertissait les deux appels à `P` — c'est-à-dire «`P(s2); P(s3)`» devient «`P(s3); P(s2)`»? Justifiez *brièvement* votre réponse.

Bonus (5 pts) Réécrivez le code MPD de la page précédente de façon à ce qu'il ait un comportement équivalent, mais ce **sans utiliser de sémaphores**. Plus précisément, vous devez remplacer la clause `final` par un `process proc5` et vous devez utiliser des instructions `< await(...) ...>`. Vous pouvez évidemment introduire des variables auxiliaires **autres que des sémaphores**.

Revoici le code du programme initial (pour éviter de retourner à la page précédente) :

```
int x = 10;

sem s1 = 1, s2 = 0, s3 = 0;

process proc1 { P(s2); P(s3); P(s1); x = 2 * x; V(s1); V(s2); V(s3) }

process proc2 { P(s1); x = x + 1; V(s1); V(s2); }

process proc3 { P(s1); x = x - 1; V(s1); V(s3); }

process proc4 { P(s2); P(s3); P(s1); x = x + 3; V(s1); V(s2); V(s3) }

final         { printf( "x = %d\n", x ); }
```

5. Opération pour recueillir les éléments d'un tableau (pseudo-MPD) (10 pts)

Les langages fondés sur le modèle SPMD (*Single Program Multiple Data*) fournissent de nombreuses opérations *collectives* de communication et synchronisation : barrière, réduction, diffusion, etc. Ces instructions...

- servent de point de synchronisation — un *thread* qui exécute l'opération bloque jusqu'à ce que tous les autres *threads* aient aussi exécuté cette opération ;
- servent aux divers *threads à échanger de l'information.*

Une opération typique est `gather` — qu'on peut traduire par «recueillir» — où l'un des *threads*, la `destination`, cherche à recueillir les éléments d'un tableau lesquels sont répartis entre divers *threads.*

Pour simplifier, on va supposer que le nombre d'éléments à recueillir, donc la taille du tableau, est égal au nombre de *threads* (y compris la destination). Donc, chaque *thread* fournit *un unique élément*, *y compris le* thread `destination`. On va aussi supposer que chaque *thread* est identifié par un numéro *unique* compris entre 1 et `nbThreads` et que les bornes du tableau à recueillir sont elles aussi comprises entre ces bornes. Après l'appel à `gather`, le *thread* `destination` aura donc reçu dans le tableau l'élément de chaque *thread* dont l'index correspond au numéro du *thread.*

La signature d'un **moniteur** pseudo-MPD exportant une telle opération est comme suit:

```
monitor Gather
  op gather( int numThread, int dest, int valeur, ptr [*]int a )
body Gather( int nbThreads )
```

Plus précisément, cette ressource/opération s'utilise de la façon suivante :

- L'argument `nbThreads` du constructeur indique le nombre de *threads* qui seront impliqués dans l'utilisation de l'opération `gather`.
- Un appel à `gather` se fait en spécifiant quatre (4) arguments :
  - `numThread` : Le numéro du *thread* appelant.
  - `destination` : Le numéro du *thread* qui recueille les divers éléments et les met dans son tableau.
  - `valeur` : La valeur à transmettre.
  - `a` : Un *pointeur* vers le tableau qui contiendra les divers éléments, tableau significatif uniquement pour le *thread* `destination`. Les *threads* autres que `destination` doivent simplement utiliser la valeur `null`.
- Tant que tous les *threads* n'ont pas effectué l'appel à `gather`, aucun résultat n'est retourné au *thread* destinataire *et tous les* threads *appelants sont bloqués.* Donc, les `nbThreads-1` premiers *threads* qui appellent `gather` bloquent.
- Lorsque les `nbThreads` *threads* ont effectué leur appel à `gather`, les éléments reçus sont copiés dans le tableau `a` par le *thread* `destination`. (Les diverses valeurs reçues doivent évidemment être copiées dans un tableau temporaire lors de l'arrivée d'un *thread.*) Ensuite, tous les *threads* peuvent poursuivent leur exécution.

Le programme MPD 1 présente un exemple d'utilisation. Le résultat produit par l'exécution de ce programme avec `NB_THREADS=10` et `DEST=8` est présenté à la Figure 1.

```
procedure imprimer( int a[*] ) { ... }

const int NB_THREADS = 10;
const int DEST = 8;

cap Gather g = create Gather( NB_THREADS );

process p[numThread = 1 to NB_THREADS] {
  int a[NB_THREADS];

  if ( numThread == DEST ) {
    g.gather( numThread, DEST, 10*numThread, @a );
    printf( "Thread %d: a = ", numThread ); imprimer( a );
  } else {
    g.gather( numThread, DEST, 10*numThread, null );
  }
}

# Remarque: L'operateur '@' permet d'obtenir l'adresse d'une variable.
# Cet operateur retourne donc une valeur de type ptr!
```

Programme MPD 1: Un petit programme (pseudo-MPD) utilisant le moniteur permettant de recueillir les éléments d'un tableau d'entiers avec `gather`.

```
% a.out
Thread 8: a = [ 10 20 30 40 50 60 70 80 90 100  ]
```

Figure 1: Trace d'exécution pour le programme MPD 1.

Écrivez (en pseudo-MPD) un **moniteur** permettant de réaliser une telle opération `gather`, et ce en utilisant la propriété d'exclusion mutuelle **implicite** d'un tel monitor et une ou plusieurs **variables de condition** (de type `cond`).

Attention : Pour simplifier le problème, il n'est pas nécessaire que cette opération soit réutilisable — donc c'est correct si c'est une *one-shot barrier*, ne pouvant pas être utilisée de façon répétitive dans une boucle.

Bonus (3 pts) : Un bonus sera accordé si une erreur d'exécution (par ex., avec `assert`) est signalée si tous les *threads* n'indiquent pas la même valeur pour `destination` et si les numéros de *thread* ne sont pas uniques.

```
monitor Gather
  op gather( int numThread, int dest, int valeur, ptr [*]int a )
body Gather( int nbThreads )









  proc gather( numThread, dest, valeur, a )
  {









    }
end
```