INF5170 — Programmation parallèle
Examen final (Hiver 2006)

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

Remarques :

- L'examen comporte six (6) questions, comptant respectivement pour 10, 10, 10, 10, 5 et 5 points, pour un total sur 50.
- Pour la première question, portant sur les moniteurs, vous pouvez utiliser la notation que vous désirez : notation du manuel d'Andrews (pseudo-MPD avec la construction `monitor`), MPD (avec les modules `Verrou` et `Cond` vus au devoir no. 3, que vous pouvez utiliser sans les définir), Java (avec ou sans les interfaces et classes de J2SE 5.0) ou même C (avec les opérations de la bibliothèque Posix).
- Pour les questions 3 et 4, portant sur les canaux de communication, vous pouvez utiliser la notation du manuel d'Andrews (pseudo-MPD) ou celle du langage MPD.

1. Opération collective de réduction (moniteur) (10 pts)

On veut définir un module permettant d'effectuer l'équivalent d'une opération collective de réduction en MPI. L'interface du module est présentée à la Figure 1. Pour simplifier, l'opération binaire à utiliser est fixée, à savoir la somme (addition).

Figure 1 Interface pour une opération collective de réduction.

```
resource ReduireSomme
  op reduire( int v ) returns int sommeTotale;
body ReduireSomme( int nbProcs )
  ...
end
```

Plus précisément, ce module fonctionne de la façon suivante :

- L'argument `nbProcs` du constructeur indique le nombre de processus qui seront impliqués dans l'utilisation de l'opération de réduction.
- Tant que tous les processus n'ont pas effectué leur appel à `reduire`, aucun résultat n'est produit. Donc, les `nbProcs-1` premiers processus qui appellent `reduire` bloquent en attente du résultat.
- Lorsque tous les `nbProcs` processus ont effectué leur appel à `reduire`, le résultat (`sommeTotale`) est retourné à chacun des processus, lesquels poursuivent ensuite leur exécution.

Le programme MPD 1 présente un petit programme utilisant ce module.
Le résultat produit par l'exécution de ce programme est présenté à la Figure 2.
En utilisation la notation de votre choix, écrivez un *moniteur*, utilisant la discipline *signaler-et-continuer*, qui permet de réaliser l'équivalent du module `ReduireSomme`.

```
resource TesterReduire()
  const int N = 5;

  import ReduireSomme;

  cap ReduireSomme rs = create ReduireSomme( N );

  process p [i = 1 to N] {
    for [k = 0 to 1] {
      int r = rs.reduire( 10**k * i );
      printf( "Processus %d: r = %d\n", i, r );
    }
  }
end
```

Programme MPD 1: Un petit programme utilisant le module de réduction pour une somme.

Figure 2 Trace possible d'exécution pour le programme MPD 1.

```
% a.out
Processus  5: r =  15
Processus  1: r =  15
Processus  2: r =  15
Processus  3: r =  15
Processus  4: r =  15
Processus  4: r = 150
Processus  5: r = 150
Processus  1: r = 150
Processus  2: r = 150
Processus  3: r = 150
```

2. Processus avec sémaphores et blocage (10 pts)

Soit le programme MPD suivant :

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

sem v1 = 1,
    v2 = 1;
```

```
process p1 {
  z += 2;
  P(v1);
  x += 2;
  P(v2);
  V(v1);
  y += 2;
  V(v2)
}
```

```
process p2 {
  P(v2)
  y += 1;
  P(v1);
  x += 1;
  V(v1);
  V(v2);
  z += 1;
}
```

a. L'exécution de ce programme peut conduire à un blocage (*deadlock*). Expliquez brièvement de quelle façon?

b. Quelles sont les valeurs finales possibles pour `x`, `y` et `z` dans l'état final où le programme est bloqué?

c. Quelles sont les valeurs finales possibles pour `x`, `y` et `z` si le programme se termine normalement sans blocage?

3. Barrière de synchronisation (canaux de communication) (10 pts)

On veut développer, à l'aide de canaux de communication, une *routine* qui permet de réaliser une *barrière de synchronisation réutilisable*, dont l'interface est la suivante :

```
procedure attendre( int i, int N )
# Permet au processus numéro i de signaler son arrivée à la barrière.
# Le processus attend ensuite jusqu'à ce que les N processus soient arrivés.
```

Cette barrière pourrait alors être utilisée tel qu'illustré dans le Programme MPD 2.

```
int N = ...     # Nombre de processus.

# Déclarations des divers canaux requis pour la mise en oeuvre.
...

procedure attendre( int i, int N )
{ ... }

process p[i = 0 to N-1]
{
  # Faire quelque chose...
  printf( "Proc %d debute\n", i );
  attendre(i, N);

  # Faire autre chose...
  printf( "Proc %d fait autre chose\n", i );
  attendre(i, N);

  # Terminer.
  printf( "Proc %d termine\n", i );
}
```

Programme MPD 2: Programme MPD avec `N` processus utilisant la barrière `attendre`.

Écrivez le code de la procédure `attendre`, et ce à l'aide de l'une ou l'autre des méthodes suivantes basées sur l'utilisation de canaux de communication (donc une parmi les trois) :

a. Solution centralisée, de type maître–esclaves.

b. Solution symétrique (SPMD) où tous les processus exécutent *exactement* le même code.

c. Solution quasi-symétrique avec un anneau de processus, où les processus communiquent avec au plus deux voisins adjacents.

4. `MCellule` (canaux de communication) (10 pts)

Une M-structure, comme une I-structure (vue au devoir no. 3), est une forme de *tableau* composé d'un certain nombre (fixe) de `MCellules`, où chaque `MCellule` assure une synchronisation implicite entre producteurs et consommateurs de cette cellule. Une `MCellule` diffère toutefois d'une `ICellule` en ce qu'elle possède le comportement suivant :

- Initialement, une `MCellule` est *vide*.
- Une `MCellule` ne peut être lue que lorsqu'elle est *pleine*. La lecture du contenu d'une `MCellule` a pour effet additionnel *de vider la* `MCellule`.
- Une `MCellule` peut être écrite plus d'une fois et pour qu'une écriture soit valide, la `MCellule` doit être vide. C'est une *erreur* d'écrire dans une `MCellule` pleine.

Soit alors les types présentés à la Figure 3.

Figure 3 Types, avec canaux de communication, pour des `MCellules` réalisées avec un *moniteur actif*.

```
optype CanalReponseOp( int );
type   CanalReponse = cap CanalReponseOp;

type SorteOp = enum( ECRIRE, LIRE );

type ArgumentsLire   = rec( CanalReponse cr );
type ArgumentsEcrire = rec( int v );

type Arguments = union( ArgumentsLire argsLire; ArgumentsEcrire argsEcrire );

optype CanalRequeteOp( SorteOp, Arguments );
type   CanalRequete = cap CanalRequeteOp;
```

Définissez un moniteur actif — donc un processus recevant et traitant des requêtes et produisant des réponses par l'intermédiaire de canaux de communication — qui permet de gérer une `MCellule`. La procédure utilisée pour créer le processus (actif) gérant les requêtes d'accès à la `MCellule` aura simplement l'interface suivante :

```
procedure gererUneMCellule( CanalRequete requete )
```

Un exemple d'utilisation d'une telle `MCellule` est présenté dans le Programme MPD 3 (page suivante).

Remarques, contraintes et indices :

- Contrairement à l'exercice 2 du devoir no. 3, vous devez traiter une *unique* `MCellule`, et non un `M-Structure` complète (qui est simplement un tableau de `MCellules`).
- Pour simplifier le problème, lorsqu'une écriture est effectuée alors que la `MCellule` est déjà pleine, alors l'exécution du programme doit simplement être *avortée* (avec `stop`).
- Pour simplifier, vous pouvez aussi supposer que le processus qui gère la `MCellule` fonctionne de façon continue, sans fin (donc pas besoin de traiter la terminaison ou de libérer la `MCellule`).
- Pour la mise en oeuvre de la procédure `gererUneMCellule`, vous ne pouvez utiliser que des canaux de communication, avec les opérations `send` et `receive` associées. *Vous ne pouvez pas utiliser de sémaphores ou de variables de condition.*
- Indice/suggestion pour «conserver» les requêtes de lecture qui doivent être mises en attente : canal!

```
resource ProgrammeAvecMCellules()

  # ... Les types indiqués précédemment...
  ...


  #
  # La procédure que vous devez mettre en oeuvre.
  procedure gererUneMCellule ( CanalRequete requete )
  { ... }


  # Exemple d'utilisation.
  # Création d'une (unique) MCellule.
  op requeteMC1( SorteOp, Arguments );
  fork gererUneMCellule( requeteMC1 );

  # Utilisation de la MCellule.
  Arguments args;

  args.argsEcrire = ArgumentsEcrire( 10 );
  send requeteMC1( ECRIRE, args );

  op reponse( int );
  args.argsLire = ArgumentsLire( reponse );
  send requeteMC1( LIRE, args );
  int v;
  receive reponse( v );
  printf( "v = %d (= 10?)\n", v );
end
```

Programme MPD 3: Exemple (simple) d'utilisation d'une `MCellule`.

5. Programme MPI «mystère» (5 pts)

Que fait le Programme MPI 1 (p. 8), écrit dans la version C de MPI?

6. Programme Java «mystère» (5 pts)

Le Programme Java 1 (p. 9) présente un programme utilisant certaines fonctionnalités associées à la bibliothèque `java.util.concurrent`. Que fait ce programme?

```
int main( int argc, char *argv[] ) {
  int nbProcs, numProc;
  int nbElems, *elems;
  int x;
  int i;

  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &nbProcs );
  MPI_Comm_rank( MPI_COMM_WORLD, &numProc );

  if ( numProc == 0 ) {
    nbElems = atoi( argv[1] );
    assert( nbElems % nbProcs == 0 );
    elems = (int*) malloc( nbElems );
    for( i = 0; i < nbElems; i++ ) {
      elems[i] = lireUnEntierSurStdin();
    }
  }

  MPI_Bcast( &nbElems, 1, MPI_INT, 0, MPI_COMM_WORLD );

  int mesNbElems = nbElems / nbProcs;
  int *mesElems = malloc(mesNbElems);
  MPI_Scatter( elems, mesNbElems, MPI_INT, mesElems, mesNbElems, MPI_INT,
               0, MPI_COMM_WORLD );

  if ( numProc == 0 ) {
    x = atoi( argv[2] );
  }
  MPI_Bcast( &x, 1, MPI_INT, 0, MPI_COMM_WORLD );

  int r = 0;
  for( i = 0; i < mesNbElems; i++ ) {
    if ( mesElems[i] == x ) {
      r += 1;
    }
  }

  int rGlobal;
  MPI_Reduce( &r, &rGlobal, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );
  if ( numProc == 0 ) {
    printf( "La reponse est %d\n", rGlobal );
  }

  MPI_Finalize();
}
```

Programme MPI 1: Un programme MPI «mystère».

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

class Mystere implements Callable<Integer> {
    static private ExecutorService pool;
    private int i, j;

    Mystere( int i, int j ) {
      this.i = i; this.j = j;
    }

    public Integer call() {
        if (j - i <= 10) {
            int r = 1;
            for( int k = i; k <= j; k++ ) {
                r *= k;
            }
            return( r );
        } else {
            int m = (i + j) / 2;
            Future<Integer> f1 = pool.submit( new Mystere(i,   m) );
            Future<Integer> f2 = pool.submit( new Mystere(m+1, j) );
            try {
               return( f1.get() * f2.get() );
            } catch(Exception ie) {
               return( Integer.MIN_VALUE );
            }
        }
    }

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println( "Usage:\n   java Mystere n" );
            System.exit(-1);
        }
        int n = Integer.parseInt(args[0]);

        pool = new ThreadPoolExecutor( Integer.MAX_VALUE, Integer.MAX_VALUE,
                                       50000L, TimeUnit.MILLISECONDS,
                                       new LinkedBlockingQueue<Runnable>());

        Future<Integer> f = pool.submit( new Mystere(1, n) );
        try {
            System.out.println( f.get() );
        } catch(Exception ie) {}

        pool.shutdown();
    }
}
```

Programme Java 1: Un programme Java «mystère».