INF5171 — Programmation concurrente et parallèle
Examen final (Automne 2015)

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

Nom: _____

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

1	2	3	4	5	6	Total
/10	/10	/5	/15	/10	/ 5	/50
					Bonus	

- L'examen comporte cinq (5) questions obligatoires et une (1) question bonus (la question no. 6, avec un programme style SIMD).
- Si nécessaire, vous pouvez utiliser, **sans les définir**, les fonctions suivantes, exprimées ici en Ruby mais qui pourraient être utilisées dans n'importe quel autre langage :

```
# Fonctions pour le calcul des bornes d'une tranche (bloc) d'elements.
def inf( k, n, nb_threads ) ...
def sup( k, n, nb_threads ) ...
```

- Lorsque vous écrivez du code Java, pour simplifier, **vous pouvez ignorer les exceptions lors d'appels à `join()`, `get()`, `await()`**, etc. — donc pas besoin de `try`.
- La déclaration «`function<bool (int)> pred`» dans l'exemple C++ (p. 7) indique un prédicat sur un `int`, donc une fonction qui reçoit un argument `int` et qui retourne `true` si l'argument satisfait une certaine condition, `false` sinon.

  Idem pour le type `Predicat` du programme OpenMP/C de la page 4.
- Concernant le type `vector<int>` en C++ (p. 7) :
  - Un objet `v` de type `vector<int>` dénote un vecteur d'entiers de longueur variable — donc comme un `Array` en Ruby.
  - `v.size()` = Nombre d'éléments présents dans le vecteur.
  - `vector<int>()` = vecteur vide — donc `vector<int>().size() == 0`.
  - `v.push_back(x)` = ajoute l'élément `x` **à la fin** du vecteur `v`.

1. Exclusion mutuelle et synchronisation conditionnelle (Ruby) (10 pts)

Soit le programme Ruby présenté à la page suivante.

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

b. Même question, mais cette fois si on modifie le **troisième** *thread* comme suit :

Avant modification:

```
val.attendre { |v| v > 0 }
mutex.synchronize do
  ...
end
```

Après modification:

```
mutex.synchronize do
  val.attendre { |v| v > 0 }
  ...
end
```

```
class Entier
  def initialize( val_initiale )
    @val = val_initiale
    @mutex = Mutex.new
    @cond = ConditionVariable.new
  end

  def inc
    @mutex.synchronize do
      @val += 1
      @cond.signal
    end
  end

  def attendre( &predicat )
    @mutex.synchronize do   # Notez le '!'
      @cond.wait( @mutex ) while !predicat.call( @val )
    end
  end
end


mutex = Mutex.new
val = Entier.new( 0 )
x = 2

PRuby.pcall(
  lambda do
    mutex.synchronize { x = 3 * x }
    val.inc
  end,

  lambda do
    val.attendre { |v| v > 0 }
    mutex.synchronize { x = 2 * x; val.inc }
  end,

  lambda do
    val.attendre { |v| v > 0 }
    mutex.synchronize { x = x + 5; val.inc }
  end
)

puts "x = #{x}"
```

2. Fonctions `detect` et `count` (OpenMP/C) (10 pts)

a. On veut réaliser, en OpenMP/C, une fonction `detect`. Cette fonction reçoit en arguments un tableau, sa taille et un *prédicat*. La fonction retourne alors un index tel que l'élément associé satisfait le prédicat. Si aucun élément ne satisfait le prédicat, la fonction retourne `-1`. Si plusieurs éléments satisfont le prédicat, un des index est retourné, **au choix de l'implémenteur**.

En Ruby, Java, ou C++, le prédicat serait représenté par une lambda-expression. En C, le prédicat est plutôt représenté par un pointeur de fonction : voir plus bas.

Squelette de la fonction `detect`

La définition du type `Predicat` et le corps de la fonction `detect` sont comme suit :

```
typedef int (*Predicat)(int);

int detect( int elems[], int n, Predicat pred )
{
  int pos = -1;

  ... segment de code indiqué à la page suivante ...

  return pos;
}
```

Exemples d'utilisation

```
int estZero( int x ) { return x == 0; }

int elems1[7] = {1, 2, 0, 3, 4, 5, 6};
assert( detect(elems1, 7, estZero) == 2 );

int elems2[7] = {1, 2, 8, 3, 4, 5, 6};
assert( detect(elems2, 7, estZero) == -1 );
```

Ce qu'il faut faire

Pour chacun des segments de code de la page suivante, **indiquez si la fonction produit ou non le bon résultat**. Même si la réponse produite est bonne, **indiquez s'il s'agit ou non d'une bonne stratégie, en justifiant brièvement votre réponse**. Finalement, indiquez aussi **laquelle parmi ces solutions vous semble la meilleure**.

Note : Il n'est pas possible, à l'intérieur d'une boucle parallèle OpenMP, d'exécuter un `return` ou un `break`.

(i)
```
#pragma omp parallel
{
  for( int i = 0; i < n; i++ ) {
    if ( (*pred)(elems[i]) ) {
      pos = i;
    }
  }
}
```

(ii)
```
#pragma omp parallel for
for( int i = 0; i < n; i++ ) {
  if ( (*pred)(elems[i]) ) {
    #pragma omp critical
    pos = i;
  }
}
```

(iii)
```
#pragma omp parallel for
for( int i = 0; i < n; i++ ) {
  if ( (*pred)(elems[i]) ) {
    pos = i;
  }
}
```

b. Soit le fragment de code suivant, qui définit une fonction qui compte le nombre d'eléments de `elems` qui satisfont le prédicat `pred`.

```
int count( int elems[], int n, Predicat pred )
{
  int nb = 0;


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

    if ( (*pred)(elems[i]) ) {

      nb += 1;

    }

  }

  return nb;
}
```

Supposons que le temps d'exécution d'un appel à `pred` soit très variable d'un élément à un autre — parfois court, parfois long.

Quelle(s) directive(s) OpenMP faudrait-il ajouter pour obtenir un temps d'exécution le meilleur possible pour ce programme? Vous devez aussi indiquer **explicitement** la stratégie de répartition des itérations à utiliser.

3. Programme mystère avec réduction parallèle (TBB/C++) (5 pts)

```
// Note: Voir page 1 pour le type vector<int>.
static vector<int> foo( vector<int> elems, function<bool (int)> pred ) {
  return parallel_reduce(
      blocked_range<size_t>(0, elems.size()),

      vector<int>(), // Vecteur vide.

      [=]( blocked_range<size_t> r, vector<int> res ) {
        for( size_t k = r.begin(); k < r.end(); k++ ) {
          if( pred(elems[k]) ) {
            res.push_back( elems[k] ); // On ajoute a la fin.
          }
        }
        return res;
      },

      [=]( vector<int> r1, vector<int> r2 ) {
        for( int i = 0; i < r2.size(); i++ ) {
          r1.push_back( r2[i] ); // On ajoute a la fin.
        }
        return r1;
      }
  );
}
```

a. Qu'est-ce qui sera retourné dans la variable `r` par l'appel suivant :

```
// On suppose elems = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 }
vector<int> r = foo( elems, []( int x ) { return x % 3 == 0; } );
```

b. De façon plus générale, que fait la fonction `foo`? Quel nom **plus significatif** pourrait-on lui donner?

4. `MStructure` (Java) (15 pts)

Une `MStructure` est une forme de tableau qui permet une synchronisation entre producteurs et consommateurs :

- Une cellule d'une `MStructure` ne peut être lue, avec `take`, que lorsqu'elle est **pleine**. La lecture de la cellule la **vide de son contenu**. Une lecture d'une cellule vide suspend le lecteur jusqu'à ce qu'une écriture soit effectuée — donc comme une `IStructure`.
- Une cellule d'une `MStructure` peut être écrite, avec `put`, plusieurs fois. Toutefois, pour qu'une écriture soit valide, la cellule **doit être vide**, sinon une **erreur** est signalée.
- Le nombre de cellules et la valeur initiale des cellules sont spécifiés au moment où la `MStructure` est créée, donc **dans le constructeur d'une classe qui met en oeuvre l'interface.**

L'état d'une cellule d'une `MStructure` alterne donc entre *plein* (valeur spécifiée à la création), *vide* (après un `take`), *plein* (après un `put`), *vide* (après un `take`), etc.

```
interface MStructure<T> {
  /**
   * Obtient le contenu de la ieme cellule, mais suspend si vide
   * jusqu'a ce que pleine
   *
   * @require 0 <= i && i < taille de la MStructure
   * @ensure La ieme cellule est vide
   *
   * @param i Index
   * @return La valeur qui etait dans la ieme cellule
   */
  T take( int i );

  /**
   * Ecrit dans la ieme cellule.
   *
   * @require 0 <= i && i < taille de la MStructure
   * @require La ieme cellule est vide
   * @ensure La ieme cellule contient v
   *
   * @param i Index
   * @param v Valeur a ecrire
   */
  void put( int i, T v ) throws AlreadyFullException;
}

class AlreadyFullException extends RuntimeException{
}
```

Un exemple d'utilisation est présenté dans le segment de code Java suivant.

```
MStructure<Integer> ms = new MStructureMonitor<Integer>( 1, 200 );

Thread t0 = new Thread( () ->  { ms.put( 0, ms.take(0) + 10 ); } );
Thread t1 = new Thread( () ->  { ms.put( 0, ms.take(0) + 20 ); } );
Thread t2 = new Thread( () ->  { ms.put( 0, ms.take(0) + 30 ); } );
t0.start(); t1.start(); t2.start();

try { t0.join(); t1.join(); t2.join(); } catch( Exception e ){}

assertEquals( (Integer) 260, ms.take(0) );
```

[5] a) Une classe `MStructureMonitor<T>` qui met en oeuvre l'interface `MStructure<T>` est présentée aux pages 10–11. Le constructeur reçoit la valeur initiale à utiliser pour les cellules, donc initialement elles sont toutes pleines. Dans cette mise en oeuvre, une cellule vide est représentée par `null`, donc si `elems[i] == null`, alors la $\texttt{i}^{ème}$ cellule est vide.

Complétez la méthode `take` (p. 11).

[10] b) Le programme Java de la page 12 présente une méthode pour calculer un `histogramme` sur les éléments d'un tableau, comme l'illustre l'exemple qui suit.

```
int elems[] = { 10, 1, 3, 3, 3, 2, 9, 1, 1, 1, 3, 10 };
int histoAttendu[] = { 0, 4, 1, 4, 0, 0, 0, 0, 0, 1, 2 };

MStructure<Integer> histo = histogramme( elems, 3 );

for( int k = 0; k < histoAttendu.length; k++ ) {
  assertEquals( (Integer) histoAttendu[k], histo.take(k) );
}
```

Complétez la méthode `histogramme` (p. 12) sous les conditions suivantes :

- Elle doit utiliser et retourner une `MStructure` pour représenter l'histogramme.
- Elle doit activer et utiliser exactement `nbThreads` *threads*, créés par l'intermédiaire d'un *pool de threads* et ce **en utilisant des lambda-expressions**.
- La répartition des éléments à traiter entre les *threads* doit se faire de façon **statique**, et ce en utilisant une approche de **parallélisme de données**, non de résultat. Donc, chaque élément du tableau analysé **ne doit être traité qu'une** (et une seule) **fois**.

```
public class MStructureMonitor<T> implements MStructure<T> {
  private T[] elems;
  private ReentrantLock[] verrous;
  private Condition[] pleins;

  public MStructureMonitor( int n, T valInit ) {
    elems = (T[]) new Object[n];
    verrous = (ReentrantLock[]) new ReentrantLock[n];
    pleins = (Condition[]) new Condition[n];

    for( int i = 0; i < n; i++ ) {
      elems[i] = valInit;  // Note: elems[i] == null => ieme cellule est vide!
      verrous[i] = new ReentrantLock();
      pleins[i] = verrous[i].newCondition();
    }
  }

  public void put( int i, T v ) {
    verrous[i].lock();
    if ( elems[i] == null ) {
      elems[i] = v;
      pleins[i].signal();
    } else {
      throw new AlreadyFullException();
    }
    verrous[i].unlock();
  }

  public T take( int i ) {
    // A COMPLETER SUR LA PAGE SUIVANTE
  }
}
```

```
public class MStructureMonitor<T> implements MStructure<T> {
  private T[] elems;
  private ReentrantLock[] verrous;
  private Condition[] pleins;

  public MStructureMonitor( int n, T valInit ) {
    // VOIR PAGE PRECEDENTE.
  }

  public void put( int i, T v ) {
    // VOIR PAGE PRECEDENTE.
  }

  public T take( int i ) {









   }
}
```

```
MStructure<Integer> histogramme( int[] elems, int nbThreads ) {
  assert elems.length % nbThreads == 0;
  int valMax = Arrays.stream(elems).max().getAsInt();
  for( int e: elems ) { assert 0 <= e && e <= valMax; }

  MStructure<Integer> histo = new MStructureMonitor<Integer>(valMax+1, 0);
  return histo;
}
```

5. Mise en oeuvre d'un `fixedThreadPool` simple (Ruby) (10 pts)

Le programme Ruby à la page 14 présente une mise en oeuvre, partielle et simplifiée, d'un `fixedThreadPool` — semblable à celui disponible en Java — défini dans la classe `ExecutorService`. Version **simplifiée** notamment parce qu'on ne définit pas de méthode `shutdown` ou `shutdown_now`!

Quant au programme Ruby à la page 15, il présente une mise en oeuvre partielle d'une classe `FutureTask` utilisée par la classe `ExecutorService` — c'est la présence d'une méthode `value` **bloquante** qui fait qu'un `FutureTask` est un **future**!

Le segment de code qui suit présente un petit exemple simple.

```
es = ExecutorService.fixedThreadPool( 2 )

f1 = es.submit { 10 }
f2 = es.submit { sleep 0.1; 20 }
f3 = es.submit { 10 + 20 }
f4 = es.submit { 2 * 20 }

assert f1.value == 10 && f2.value == 20 && f3.value == 30 && f4.value == 40
```

a. **Complétez le code de la méthode `submit` de la classe `ExecutorService`** (p. 14).

b. **Complétez le code de la méthode `value` de la classe `FutureTask`** (p. 15).

```
class ExecutorService
  def ExecutorService.fixedThreadPool( nb_threads )
    new( nb_threads )
  end

  def initialize( nb_threads )
    @nb_threads = nb_threads
    @taches_a_traiter = Array.new
    @mutex = Mutex.new
    @tache_disponible = ConditionVariable.new

    # On active les threads.
    nb_threads.times do
      Thread.new do
        while true
          @mutex.lock
          @tache_disponible.wait( @mutex ) while @taches_a_traiter.empty?
          tache = @taches_a_traiter.pop
          @mutex.unlock
          tache.run
        end
      end
    end
  end

  def submit( &tache )
  # Cree un objet FutureTask et l'ajoute dans les taches a traiter.
  # Retourne le future ainsi cree.









  end
end
```

```
class FutureTask
  def initialize( tache )
    @tache = tache
    @resultat = nil    # Indique un resultat non disponible!
    @mutex = Mutex.new
    @prete = ConditionVariable.new
  end

  def value
  # Retourne le resultat de l'evaluation de la tache,
  # mais bloque si pas encore disponible.









  end

  def run
    # On execute la tache.
    r = @tache.call
    @mutex.synchronize do
      # On indique que le resultat est disponible.
      @resultat = r
      @prete.signal
    end
  end
end
```

6. Programme mystère SIMD (Ruby) (Bonus 5 pts)

```
class Array
  def foo
    return [] if size == 0
    DBC.require (lg = Math.log(size, 2)).to_i.to_f == lg

    res = Array.new( size )
    barriere = Barriere.new( size )
    PRuby.pcall( 0...size,
                 lambda do |k|
                   res[k] = self[k]
                   dist = 1
                   while dist < size
                     barriere.attendre
                     if (k+dist+1) % (2*dist) == 0
                       (k+1..k+dist).each do |j|
                         res[j] = yield( res[k], res[j] )
                       end
                     end
                     dist *= 2
                   end
                 end )
    res
  end
end
```

Indiquez la valeur appropriée pour `res1` et `res2` pour que les cas de tests ci-bas réussissent.

```
# Cas avec 4 elements.
  a1 = [5, 2, 8, 10]

  res1 =

  assert a1.foo(&:*) == res1

# Cas avec 8 elements.
  a2 = [10, 20, 30, 40, 100, 200, 300, 400]

  res2 =

  assert a2.foo(&:+) == res2
```