INF5171 — Programmation concurrente et parallèle
Examen final (Automne 2016)

Durée: 13h30 – 16h30 **Documentation :** Documentation personnelle autorisée.

Nom: _____

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

1	2	3	4	5	6	7	Total
/5	/5	/5	/10	/5	/10	/10	/50

- Quelques éléments de Ruby :
  - Lorsqu'on crée un `Hash`, les deux principales façons de spécifier une valeur par défaut (la valeur retournée si la clé n'est pas présente) sont les suivantes :

```
hash = Hash.new( valeur_simple )          # Avec une valeur primitive
hash = Hash.new { |h, k| h[k] = Foo.new } # Avec un objet
```

  - Les deux instructions avec garde suivantes ont le même effet :

```
instruction until cond
instruction while !cond
```

  - La méthode `step` permet de générer les items d'un `Range` avec un incrément :

```
>> (0...50).step(20).each { |k| puts k }
0
20
40
=> 0...50

>> (0...50).step(23).map { |k| k }
=> [0, 23, 46]
```

  - La méthode `even?` indique si un nombre est pair ou non :

```
>> 2.even?                    >> (2+1).even?
=> true                       => false
```

- En Java, la classe `AtomicInteger` possède entre autres les méthodes suivantes :

```
int getAndIncrement()
  Atomically increments by one
       the current value.
  Returns: the previous value
```

```
int get()
  Gets the current value.
```

1. Exclusion mutuelle et synchronisation conditionnelle (Ruby) (5 pts)

Soit le programme Ruby présenté à la page suivante.

a. Indiquez quelles sorties peuvent être imprimées par ce programme? Indiquez aussi si un interblocage (*deadlock*) est possible ou non.

b. Même question, mais cette fois, dans la méthode `Valeur#inc`, on remplace «`@cond.signal`» par «`@cond.broadcast`».

```
class Valeur
  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
    @mutex.synchronize do
      @cond.wait( @mutex ) until yield @val
    end
  end
end
-------------------------------------------
val = Valeur.new(0)
x = 3
mutex = Mutex.new

[Thread.new do
   val.attendre { |val| val > 0 }
   mutex.synchronize { x += 2 }
 end,

 Thread.new do
   mutex.synchronize { x += 4 }
   val.inc
 end,

 Thread.new do
   val.attendre { |val| val > 0 }
   mutex.synchronize do
     x *= 3
     val.inc
   end
 end
].map(&:join)

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

2. Programme mystère avec espace de tuples (Ruby) (5 pts)

Soit le module `Foo` à la page suivante, qui utilise la classe `TupleSpace` du Devoir #2.

a. Quelle sera la valeur de `r` après l'appel suivant :

```
elems = [1, 2, 3, 4, 5, 6, 7]
r = Foo.run( elems, 4, 2 ) { |x| x.even? }
```

b. De façon générale, que fait la méthode `Foo.run`? Quel nom **plus significatif** peut-on lui donner?

c. Que se passe-t-il si, dans la méthode `run`, on supprime la ligne suivante :

```
nb_threads.times { ts.take [:termine] }
```

d. Que se passe-t-il si, toujours dans `run`, on supprime plutôt la ligne suivante — et on laisse celle avec «`ts.take [:termine]`» :

```
nb_threads.times { ts.put [:stop] }
```

```
module Foo
  def self.travailleur( ts, bloc )
    loop do
      tache = ts.take( /^(tache|stop)$/ )
      return [:termine] if tache[0] == :stop

      _, inf, vals = tache
      (0...vals.size).each do |k|
        ts.put [:resultat, inf + k, bloc.call(vals[k])]
      end
    end
  end

  def self.run( a, nb_threads, taille_tache, &bloc )
    ts = TupleSpace.create
    nb_threads.times do
      ts.eval { travailleur(ts, bloc) }
    end

    (0...a.size).step(taille_tache).each do |inf|
      sup = [inf + taille_tache - 1, a.size - 1].min
      ts.put [:tache, inf, a[inf..sup]]
    end

    res = Array.new(a.size)
    res.size.times do
      _, index, r = ts.take :resultat
      res[index] = r
    end

    nb_threads.times { ts.put [:stop] }
    nb_threads.times { ts.take [:termine] }

    res
  end
end
```

3. Programme mystère (Java) (5 pts)

```
public static int mystere( int[] elems,
                           Function<Integer, Boolean> f,
                           int nbThreads ) {
  ExecutorService pool = Executors.newFixedThreadPool(nbThreads);
  int[] res = new int[elems.length];
  AtomicInteger ai = new AtomicInteger(0);

  for ( int numThread = 0; numThread < nbThreads; numThread++ ) {
    final int numThreadf = numThread;
    pool.submit( () -> {
        for( int k = numThreadf; k < elems.length; k += nbThreads ) {
          if ( f.apply(elems[k]) ) {
            res[ai.getAndIncrement()] = elems[k];
          }
        }
      } );
  }

  pool.shutdown();
  try { pool.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS); }
  catch (InterruptedException e) {}

  for( int i = 0; i < ai.get(); i++ ) { elems[i] = res[i]; }

  return ai.get();
}
```

a. Qu'est-ce qui sera imprimé par le fragment de code qui suit — si un seul résultat est possible, indiquez-le explicitement, sinon indiquez au moins deux (2) résultats :

```
int[] a = new int[]{ 5, 4, 3, 2, 1, 3, 2, 3, 4, 5 };
int r = mystere( a,  (x) -> x <= 3, 2 );
for( int i = 0; i < r; i++ ) { System.out.print( a[i] + " " ); }
```

b. De façon générale, que fait `mystere`? Quel nom **plus significatif** peut-on lui donner?

4. Mise en oeuvre parallèle de la méthode `group_by` (Ruby) (10 pts)

La méthode `group_by`, appliquée sur un `Array`,[1] produit un `Hash` où chaque clé est une valeur retournée par l'application du bloc et où la définition associée est la liste des éléments qui génèrent cette clé (via l'évaluation du bloc). Des exemples sont présentés ci-bas.

```
>> a = [100, 200, 300, 400]
=> [100, 200, 300, 400]

>> a.group_by { |x| x }
=> {100=>[100], 200=>[200], 300=>[300], 400=>[400]}

>> a.group_by { |x| x >= 222 }
=> {false=>[100, 200], true=>[300, 400]}

>> a.group_by { |x| x / 100 }
=> {1=>[100], 2=>[200], 3=>[300], 4=>[400]}

>> a.group_by { |x| x % 2 }
=> {0=>[100, 200, 300, 400]}

>> a.group_by { |x| (x / 100) % 2 }
=> {0=>[200, 400], 1=>[100, 300]}
```

Le code Ruby suivant présente une mise en oeuvre séquentielle de cette méthode :

```
class Array
  def group_by
    hash = Hash.new { |h, k| h[k] = Array.new }

    (0...size).each do |k|
      key = yield self[k]
      hash[key] << self[k]
    end

    hash
  end
end
```

[1] En fait, la méthode s'applique à tout `Enumerable`.

Donnez une mise en oeuvre parallèle qui satisfait les conditions suivantes :

- Utilise uniquement les *threads* de base de Ruby — donc **n'utilise pas** `PRuby`!
- Crée exactement `nb_threads` *threads*, **sauf si le tableau est trop petit**, auquel cas crée un *thread* par élément.
- Distribue le travail par une **répartition cyclique** des éléments entre les *threads*.
- Assure l'exclusion mutuelle à un **niveau fin de granularité**, c'est-à-dire, **au niveau de chaque clé**. Donc, **vous ne devez pas utiliser un seul `Mutex` global!**

```
class Array
  def group_by_par( nb_threads = $NB_THREADS )
    assert nb_threads >= 1 && (size < nb_threads || size % nb_threads == 0)
```

```
  end
end
```

5. Fonctions sur des vecteurs (OpenMP/C) (5 pts)

Des opérations souvent utilisées lorsqu'on manipule des vecteurs et matrices de flottants sont le *SAXPY* (*Single-precision A · X Plus Y*) et le produit scalaire : `saxpy` reçoit un scalaire `a` et deux vecteurs `x` et `y` (de taille `n`), puis **multiplie** chaque élément `x[i]` par `a` et **ajoute** le résultat à `y[i]` ; `produit_scalaire` est utilisé, entre autres, pour combiner les éléments d'une ligne (`x`) et d'une colonne (`y`) lors d'un produit matriciel.

On veut paralléliser, **en OpenMP/C**, les fonctions ci-bas. Complétez les de façon **à les rendre les plus parallèles et efficaces possible.** Indiquez aussi quelle `schedule` il vous semble approprié d'utiliser

---

```
void saxpy( float a, float x[], float y[], int n )
{



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


      y[i] = a * x[i] + y[i];

  }

}



float produit_scalaire( float x[], float y[], int n )
{
  float ps = 0.0;


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


      ps += x[i] * y[i];

  }


  return ps;
}
```

6. Fonctions sur des polynomes (TBB/C++) (10 pts)

Soit $p(x)$ un polynôme de degré n :

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_{n-1} x^{n-1} + a_n x^n$$

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

  $$p'(x) = a_1 + 2a_2 x + 3a_3 x^2 + \ldots + (n-1)a_{n-1} x^{n-2} + n a_n x^{n-1}$$

  Dans le cas spécial où $p(x) = a_0$ (une constante), $p'(x) = 0$.

- Quant à la `valeur` du polynome p au point v, c'est simplement la valeur obtenue évaluant p en v :

  $$p(v) = a_0 + a_1 v + a_2 v^2 + a_3 v^3 + \ldots + a_{n-1} v^{n-1} + a_n v^n$$

Complétez (page suivante) les fonctions `dérivée` et `valeur` pour les rendre les plus parallèles et efficaces possible, et ce **en utilisant les constructions** `parallel_for` et/ou `parallel_reduce` **avec** `blocked_range` de TBB/C++.

Note :

- La valeur v^k peut être calculée en utilisant la fonction standard suivante, où `v` est un `double` et `k` un entier :

  `std::pow(v, k) // Calcule` v^k.

```
// a.
Polynome dérivée( Polynome p ) {
  if ( p.degre == 1 ) {
    Polynome pPrime = allouer( 1 );
    pPrime.coefficients[0] = 0;
    return pPrime;
  }









}
// b.
double valeur( Polynome p, double v ) {









}
```

7. Cellule mutable `MCell` (Java) (10 pts)

Une `MCell` (*Mutable Cell*) est une forme de variable qui permet une synchronisation entre producteurs et consommateurs :

- Une `MCell` 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 un comportement avec lecture bloquante qui ressemble à celui d'une cellule d'une `IStructure`.
- Une `MCell` peut être écrite, avec `put`, **plusieurs fois** (donc contrairement à une cellule d'une `IStructure`). Toutefois, pour qu'une écriture soit valide, **la cellule doit être vide**, sinon une exception est signalée.
- La valeur initiale est spécifiée au moment où la `MCell` est créée, via le constructeur.

L'état d'une `MCell` alterne donc entre *plein* (valeur spécifiée à la création), *vide* (après un `take`), *plein* (après un `put`), *vide* (après un `take`), etc. Un exemple d'utilisation est présenté dans le segment de code Java suivant.

```
MCell<Integer> mc = new MCell<>(0);

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

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

assertEquals( (Integer) 60, mc.take() );
```

[5] a) Complétez les méthodes `put` et `take` de la classe `MCell<T>` présentée à la page suivante, et ce en utilisant les attributs indiqués. Notez que dans cette mise en oeuvre, une **cellule vide** est représentée par un champ `val == null`.

```
public class MCell<T> {
  private T val;
  private Lock verrou;
  private Condition pleine;

  public MCell( T valInitiale ) {
    val = valInitiale;
    verrou = new ReentrantLock();
    pleine = verrou.newCondition();
  }

  public void put( T v ) {


    if ( val != null ) { throw new AlreadyFullException(); }

    val = v;



  }

  public T take() {









  }
}
```

[5] b) Le programme Java de la page suivante présente une méthode pour calculer un `histogrammeDeNotes`, comme l'illustre l'exemple ci-bas. **Complétez la méthode** `histogrammeDeNotes` **en respectant les conditions suivantes** :

- La méthode doit activer et utiliser exactement `nbThreads` *threads*, créés par l'intermédiaire d'un *pool de threads* et **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**, et non avec du parallélisme de résultat. Donc, chaque élément du tableau `notes` **ne doit être traité qu'une seule fois**.
- Vous devez utiliser la méthode suivante pour déterminer la classe d'une note, i.e., l'index de l'histogramme (le «*bucket*») dans lequel la valeur indiquée doit être mise :

```
/*
 * Determine la classe (l'index) de val.
 *
 * Precondition:  0 <= val <= 100
 * Postcondition: 0 <= result < nbClasses
 *
 * Exemples avec les 4 classes [0, 25], (25, 50], (50, 75], (75, 100]
 *    numClasse(  0, 4) == 0
 *    numClasse( 10, 4) == 0
 *    numClasse( 40, 4) == 1
 *    numClasse( 75, 4) == 2
 *    numClasse( 99, 4) == 3
 *    numClasse(100, 4) == 3
 */
private int numClasse( int val, int nbClasses ) { ... }
```

- La terminaison doit s'effectuer **correctement et proprement**.

```
// Exemple d'utilisation de la methode histogrammeDeNotes.

int notes[] = { 97, 79, 89, 90, 58, 74, 100, 40, 90, 99, 91, 70 };

// Histogramme avec 4 classes, généré avec 2 threads.
MCell<Integer>[] histo = histogrammeDeNotes( notes, 4, 2 );

int histoAttendu[] = { 0, 1, 3, 8 };
for( int k = 0; k < histoAttendu.length; k++ ) {
   assertEquals( (Integer) histoAttendu[k], histo[k].take() );
}
```

```
@SuppressWarnings("unchecked")
MCell<Integer>[] histogrammeDeNotes( int[] notes,
                                     int nbClasses,
                                     int nbThreads ) {
  // Preconditions.
  assert notes.length % nbThreads == 0;
  for( int e: notes ) { assert 0 <= e && e <= 100; }

  MCell<Integer>[] histo = new MCell[nbClasses];
  for( int i = 0; i < nbClasses; i++ ) {
    histo[i] =                          // Initialisation a completer!
  }

  return histo;
}
```