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

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

Nom: _____

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

1	2	3	4	5	Total
/20	/10	/ 5	/5	/10	/50

- L'examen comporte cinq (5) questions pour un total sur 50. L'examen comporte aussi une question pouvant donner lieu **à des points bonus**, question optionnelle, qui sera corrigée de façon plus stricte. (Il est donc possible d'obtenir une note > 50!)
- Répondez directement sur le questionnaire. (Notez que l'espace alloué sur la page n'est pas nécessairement une indication de l'espace requis pour la réponse!)
- À moins d'indication contraire, vous devez écrire du code **clair et simple**, sans vous soucier d'optimiser des petits détails de programmation — **KISS**!
- La méthode `Range#step(k)` permet d'obtenir les éléments en incrément de `k` :

```
(0..10).step(3).map { |x| x } == [0, 3, 6, 9]
(0..10).step(10).map { |x| x } == [0, 10]
(0...10).step(10).map { |x| x } == [0]
```

- La méthode `Array#reverse` produit un tableau avec les éléments en ordre inverse :

```
[1, 2, 3, 4].reverse == [4, 3, 2, 1]
[*0...5].reverse == [4, 3, 2, 1, 0]
[].reverse == []
```

- La méthode `Enumerable#any?` retourne `true` si la collection satisfait un prédicat, sinon `false` :

```
a = [10, 20, 30,40]
a.any? { |x| x == 30 } == true
a.any? { |x| x.even? } == true
a.any? { |x| x > 100 } == false
```

- *Si nécessaire*, vous pouvez utiliser, sans les définir, les fonctions suivantes :

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

1. Calcul de la dérivée d'un polynome (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 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 `derivee` :

```
#
# Calcul de la derivee d'un polynome
#
# @return [Polynome] Nouveau polynome qui represente la derivee.
#
def derivee
 ...
end
```

a. Écrivez une version parallèle de la fonction `derivee` utilisant du parallélisme **de granularité fine** — la plus fine et parallèle possible — et utilisant du parallélisme *fork/join* avec l'instruction `PRuby.pcall`.

b. Écrivez une version parallèle de la fonction `derivee` utilisant du parallélisme **à granularité grossière** et utilisant au plus `$nb_threads` *threads*.

   Plus précisément, pour cette deuxième version (granularité grossière) :

   - Vous **ne devez pas** utiliser du parallélisme de style *fork/join*.
   - Vous pouvez supposer que `$nb_threads` est une variable **globale** (préfixe $) déjà définie et à laquelle vous pouvez vous référer.

   En outre, vous devez aussi **indiquer quel patron de programmation et quelle stratégie de répartition** vous avez utilisés et **expliquer brièvement pourquoi** vous avez choisi cette approche.

2. Arbres `N`-aires — parallélisme récursif et de données (10 pts)

Soit les classes ci-bas qui permettent de définir des **arbres `N`-aires** — donc des arbres où les noeuds internes ont toujours **exactement** `N` enfants.

```
N = ...    # Nombre d'enfants pour les noeuds internes.

class Arbre
  attr_reader :valeur

  def initialize( valeur ); @valeur = valeur; end
end

class Feuille < Arbre
  def satisfait?; yield self; end
end

class Noeud < Arbre
  def initialize( valeur, *enfants )
    DBC.require enfants.size == N && enfants.all? { |enf| enf.kind_of? Arbre }

    @enfants = enfants
    super( valeur )    # Pour initialiser le champ @valeur
  end

  def satisfait?( &condition )
    return true if yield self

    # Parcours complet (i.e., non-court-circuite) des sous-arbres.
    @enfants.any? { |enf| enf.satisfait? &condition }
  end
end
```

C'est la préconditon de `Noeud#initialize` qui assure qu'on a toujours un **arbre `N`-aire** : soit on a une `Feuille` sans enfant, soit on a un `Noeud` avec exactement `N` enfants, eux-mêmes des `Arbre` `N`-aires. On note aussi que chaque noeud, `Feuille` ou `Noeud` interne, possède un attribut `@valeur`, défini dans la superclasse `Arbre`.

La fonction `satisfait?` permet de déterminer **si un arbre ou un de ses sous-arbres** satisfait une certaine propriété. Elle utilise la méthode `any?` (module `Enumerable`) qui retourne `true` si **un des éléments** de la collection satisfait la condition indiquée (voir première page).

Soit alors l'arbre ternaire `a0` suivant — donc `N = 3` :

```
a0 = Noeud.new( 10, Feuille.new(20), Feuille.new(30), Feuille.new(40) )
```

Les conditions suivantes sont alors vraies : parce que `a0` possède une `valeur` de 10 dans le 1^{er} cas et parce que le 3^{e} sous-arbre de `a0` possède une `valeur` de 40 dans le 2^{e}:

```
a0.satisfait? { |n| n.valeur == 10 } == true
a0.satisfait? { |n| n.valeur == 40 } == true
```

Note : Dans ce qui suit, les versions parallèles de `satisfait?`, comme la version séquentielle, n'ont pas besoin d'effectuer une évaluation *court-circuitée* — l'arbre est donc **toujours parcouru au complet**.

[5] a) Écrivez une version *parallèle* de `Noeud#satisfait?` utilisant du **parallélisme récursif de style fork/join** avec des `future`s.

```
def satisfait?( &condition )
```

```
end
```

[5] b) Générer un grand nombre de (petits) *threads* de façon récursive peut être coûteux. Bien que simple, une solution comme la précédente n'est généralement pas efficace.

Modifions l'interface de `satisfait?` en ajoutant un argument `parallele`, *optionnel*, qui indique si l'exécution doit se faire de façon parallèle — `true`, cas par défaut — ou de façon séquentielle — `false`.

Écrivez une version *parallèle* de `Noeud#satisfait?` utilisant du **parallélisme de données** qui va générer des *threads* **uniquement pour les enfants immédiats de la racine**. Pour les autres sous-arbres, les appels récursifs se feront ensuite de façon séquentielle.

Vous devez supposer que `N` **est grand — nettement plus grand que le nombre de *threads*!** — et que les sous-arbres sont de tailles **très variables** — certains sont très petits, d'autres très grands.

```
def satisfait?( parallele = true, &condition )









end
```

3. Calcul du nombre d'inversions avec un sac de tâches (5 pts)

Comme dans les laboratoires 2 et 3, on veut calculer le nombre *d'inversions* dans un `Array`, i.e., le nombre d'éléments adjacents du tableau qui ne sont pas dans le bon ordre.

La méthode présentée à la page suivante, incomplète, vise à effectuer ce calcul en utilisant un *pool de threads* et un sac de tâches (classe `TaskBag`) — donc à l'aide d'une répartition dynamique des tâches entre un nombre fixe de *threads*.

Dans ce cas-ci, le nombre de *threads* utilisé est celui spécifié par `PRuby.nb_threads`. Quant aux diverses tâches réparties entre les *threads*, elles comptent toutes `TAILLE_TACHE` éléments, sauf peut-être pour la dernière tâche si la taille du `Array` n'est pas divisible par `TAILLE_TACHE`.

Complétez le code de cette méthode.

Remarque : Vous pouvez supposer que la méthode suivante est déjà définie (dans `Array`) :

```
class Array
  ...

  def nb_inversions_seq_ij( i, j )
    (i+1..j).reduce(0) { |nb, k| self[k-1] > self[k] ? nb+1 : nb }
  end

  ...
end
```

```
TAILLE_TACHE = ...   # Constante determinant la granularite des taches.

def nb_inversions
  return 0 if empty?

  nb_threads = [PRuby.nb_threads, size].min
  sac = TaskBag.new( nb_threads )
  (0...size).step(TAILLE_TACHE).each do |k|
    sac.put [k, [size-1, k+TAILLE_TACHE-1].min]
  end

  futures = (0...nb_threads).map do
    # Code pour les travailleurs.
    PRuby.future do









    end
  end









end
```

4. Un pipeline `mystere` (5 pts)

Pour cette question, **vous avez le «choix» entre deux sous-questions** :

- Si vous répondez correctement à la première question, vous obtiendrez un maximum de 5 points.
- Si vous répondez correctement à la deuxième question, vous obtiendrez un maximum de 10 points, donc vous obtiendrez 5 points bonus.
- Vous pouvez répondre aux deux sous-questions, auquel cas je choisirai la réponse qui vous donnera le maximum de points.

Dans les deux cas, la méthode `PRuby.pipeline` est utilisée. Cette méthode permet de créer explicitement un pipeline avec une série de `lambda`s plutôt qu'avec un seul `lambda`. Ainsi, les deux pipelines suivants sont équivalents :

```
l1 = lambda ...
l2 = lambda ...
l3 = lambda ...

p1 = PRuby.pipeline_source( [10, 20, 30] ) |
     l1 |  l2 |  l3 |
     PRuby.pipeline_sink( res )

p2 = PRuby.pipeline_source( [10, 20, 30] ) |
     PRuby.pipeline( l1, l2, l3 ) |
     PRuby.pipeline_sink( res )
```

[5] a) Qu'est-ce qui sera imprimé par le segment de code suivant :

```
lambdas = (0..3).map do |k|
  lambda do |chan|
    chan.until_eos do |x|
      if k.even?
        chan.put x + k
        chan.put x + k + 1
      else
        chan.put x if x.even?
      end
    end
  end
end


res = []
( PRuby.pipeline_source([1, 2, 3, 4, 5]) |
  PRuby.pipeline( *lambdas ) |
  PRuby.pipeline_sink(res)
  ).run

puts res.inspect
```

[10] b) Soit la méthode `mystere`, présentée à la page suivante, définie dans la classe `Polynome` du devoir #1.

a. Qu'est-ce qui sera imprimé par le segment de code suivant :

```
p = Polynome.new( 10, 20, 30 )
puts p.mystere( 1, 2, 3 )
```

b. De façon plus générale, que fait cette méthode?

```
class Polynome
  ...

  def mystere( *xs )
    debut = lambda do |chan|
      chan.until_eos { |v| chan.put [v, 0] }
    end

    lambdas = @coeffs.reverse.map do |coeff|
      lambda do |chan|
        chan.until_eos do |v|
          x, y = v
          chan.put [x, y * x + coeff]
        end
      end
    end

    fin = lambda do |chan|
      chan.until_eos { |v| chan.put v.last }
    end

    results = []
    ( PRuby.pipeline_source(xs) |
      debut |
      (PRuby.pipeline *lambdas) |
      fin |
      PRuby.pipeline_sink(results)
      ).
      run

    results
  end
  ...
end
```

5. Vrai ou faux (10 pts)

Pour chacune des affirmations suivantes, dites si elle est *vraie* ou *fausse* et justifiez **brièvement** votre réponse (verso si nécessaire... mais bref).

a. Lorsqu'on utilise l'approche diviser-pour-régner récursive, si on veut pouvoir exploiter le parallélisme, il faut nécessairement un langage qui permet de créer et activer un nombre arbitraire de *threads*, et ce de façon dynamique (i.e., durant l'exécution).

b. Si on augmente le nombre de processeurs d'une machine parallèle et qu'on ajuste le nombre de *threads* en conséquence — par exemple, avec deux fois plus de processeurs, on double les *threads* —, cela améliore toujours les performances du programme.

c. Sur une machine à p processeurs, la meilleure stratégie pour obtenir un programme avec la meilleure accélération absolue est de décomposer le problème en exactement p sous-problèmes indépendants.

d. L'algorithme suivant, exprimé en `PRuby`, permet de faire la somme de deux vecteurs `a` et `b` (de taille `n` > 2) et de mettre le résultat dans `c` (taille `n`) :

```
PRuby.pcall\
  (0...n).step(2), { |i| c[i] = a[i] + b[i] },
  (1...n).step(2), { |j| c[j] = a[j] + b[j] }
```

Le **coût** de cet algorithme est $O(n \times n) = O(n^2)$.

e. Lorsqu'on veut traiter une série d'éléments, par exemple, les diverses lignes d'un tableau, on utilise une distribution cyclique des lignes entre les *threads* dans le but d'éviter les interférences de lecture/écriture entre les *threads*, et donc accroitre ainsi le parallélisme.