INF5171 — Programmation concurrente et parallèle
Examen intra (Automne 2017)

Durée: 13h30 – 16h30
Documentation : Documentation autorisée, dont tablette ou *laptop* — en «lecture»!

Nom: _____

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

1	2	3	4	5	6	7	Total
/6	/12	/ 6	/6	/6	/6	/8	/50

- Répondez directement sur le questionnaire, en utilisant uniquement **le recto** des feuilles! (Vous pouvez utiliser le verso pour vos **brouillons**.)

Quelques rappels Ruby :

- `Range#step(k)` permet d'obtenir les éléments en incrément de `k` :

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

Note : `step` retourne un `Enumerator`, pas un `Range` ou un `Array`. Les méthodes `peach`, `peach_index`, `pmap`, etc., de `PRuby` **ne peuvent donc pas être utilisées directement sur le résultat de** `step`!

- *Si nécessaire*, vous pouvez utiliser — sans la définir — la méthode suivante, qui retourne un `Range` qui représente les index de la tranche attribuée au *thread* `num_thread` :

```
#  num_thread: Numero du thread
#  n: Nombre d'elements a repartir entre les threads
#  nb_threads: Nombre de threads entre lesquels repartir les elements
#
def bornes_de_tranche( num_thread, n, nb_threads )
  ...
end
```

- Un appel à «<<» sur un tableau modifie le tableau **et le retourne** en résultat :

```
>> a = [10, 20]
=> [10, 20]
>> a << 30
=> [10, 20, 30]
>> a
=> [10, 20, 30]
```

1. Répartition des éléments d'un tableau avec `peach` (6 pts)

Soit `a` un `Array` de 12 éléments, où la valeur indique le temps requis pour traiter l'élément.

10	3	6	9	8	10	4	3	1	9	12	6

Pour chaque appel ci-bas, indiquez *i*) **quelles valeurs seront traitées par chaque *thread*** et *ii*) **quel sera le temps total d'exécution**.

Note : On suppose que les *threads* obtiennent les tâches selon l'ordre de priorité de leur numéro, i.e., si deux *threads* veulent une tâche «en même temps», c'est le *thread* avec le plus petit numéro qui obtient la tâche en priorité.

---

a. `a.peach( nb_threads: 2 ) { |x| ... }`

t_0 :

t_1 :

temps total =

b. `a.peach( static: 3, nb_threads: 2 ) { |x| ... }`

t_0 :

t_1 :

temps total =

c. `a.peach( dynamic: 2, nb_threads: 2 ) { |x| ... }`

t_0 :

t_1 :

temps total =

2. Méthode `mystere` de la classe `Array` (12 pts)

Soit la méthode `mystere` **de la classe** `Array` — pour `bornes_de_tranche`, cf. p. 1 :

```
class Array
  def mystere
    nb_threads = [PRuby.nb_threads, size].min

    futures = (0...nb_threads).map do |num_thr|
      PRuby.future do
        res = true
        bornes_de_tranche( num_thr, size, nb_threads ).each do |k|
          res = false if res && !yield(self[k])
        end

        res
      end
    end
    futures.reduce(true) { |r, x| r && x.value }
  end
  ...
```

---

a. Complétez les cas de test suivants pour qu'ils s'exécutent avec succès :

```
[10, 20, 30, 40].mystere { |x| x }.must_equal

[10, 20, 30, 40].mystere(&:even?).must_equal

[10, 20, 30, 40].mystere { |x| (x / 10).even? }.must_equal

[*0..100].mystere { |x| x < 100 }.must_equal
```

b. De façon plus **générale**, que fait cette méthode? (Donc, on veut savoir «QUOI?» et non pas «COMMENT?»!) Quel **nom plus significatif** peut-on lui donner?

c. Donnez une mise en oeuvre de la méthode `mystere` utilisant du **parallélisme de boucles** (`peach` ou `peach_index`). Est-ce que cette méthode produit **toujours** le bon résultat? Justifiez brièvement votre réponse (au verso si nécessaire).

```
def mystere




end
```

d. Complétez la méthode `mystere_rec_ij` pour utiliser du **parallélisme *fork-join* récursif dichotomique**, **sans seuil de récursion** — i.e., cas de base = `i == j`!

```
def mystere
  return true if empty?
  mystere_rec_ij( 0, size - 1 ) { |x| yield(x) }
end

def mystere_rec_ij( i, j )









end
```

3. Mesures de performances (6 pts)

Soit une machine parallèle *à mémoire partagée* **comportant 15 processeurs**. On a mesuré le temps requis par un **programme parallèle** pour traiter un groupe de données avec différents nombres de *threads* (1, 5, 10, 15) et on a obtenu les temps suivants (temps moyens, en *sec.*, calculés pour plusieurs répétitions **après** une période de «réchauffement») :

Nb. thr.	**Temps**
1	110
5	30
10	12
15	15

Le temps pour un **programme séquentiel** avec les mêmes données est de **100** *sec.*

a. Complétez le tableau suivant donnant les **accélérations absolues**.

Nb. thr.	**Acc. absolue**
5	
10	
15	

b. Complétez le tableau suivant donnant les **accélérations relatives**.

Nb. thr.	**Acc. relative**
5	
10	
15	

c. Un collègue suggère d'ajouter des processeurs à cette machine. Est-ce qu'un tel ajout **pour le traitement de ce problème avec ces mêmes données** en vaudrait la peine? Justifiez brièvement votre réponse.

d. En regard de l'accélération **absolue**, pour quel nombre de *threads* obtient-on la meilleure **efficacité** et quelle est cette efficacité (en **pourcentage**)?

4. Classification de données par la méthode des k-moyennes (6 pts)

L'extrait de Code Ruby 1 présente la **spécification** d'une méthode `representants_finaux`, pour la classification de données par la méthode des k-moyennes du devoir #1.

Alors que `representants` retourne les «vrais» barycentres (les vraies moyennes) de chacun des groupes, `representants_finaux` retourne, pour chaque groupe, **le point du groupe parmi les «vrais» points qui est le plus près du barycentre**.

La différence entre les deux méthodes est illustrée par le cas de test suivant, le point `p2` (2.0) étant celui qui est le plus près du barycentre (3.0) :

```
p1, p2, p3 = [1.0, 2.0, 6.0].map { |v| Point.new("_", Vector[v]) }
cl = Classificateur.new( p1, p2, p3 )
cl.run( 1 )

cl.representants.map(&:position).must_equal [Vector[3.0]]
cl.representants_finaux.must_equal [p2]
```

Donnez une mise en oeuvre parallèle (page 7) de cette méthode.

```
#
# Retourne la liste des représentants des groupes, mais choisis
# parmi les points existants. Plus spécifiquement, le représentant
# final d'un groupe est le point du groupe qui est le plus près du barycentre.
#
# @return [Array<Point>]
#
# @require run a été appelée au moins une fois
#
# @ensure result.size == nb_groupes spécifié lors du dernier appel à run
# @ensure result.all? { |r| points.include?(r) }
# @ensure result.all? { |r| r est  plus près du barycentre
#                             que tous les autres points du même groupe }
#
def representants_finaux
  ...
end
```

Extrait de code Ruby 1: Méthode `representants_finaux`.

```
# @return [Array<Point>]
#
# @ensure result.size == nb_groupes spécifié lors du dernier appel à run
# @ensure result.all? { |p| points.include?(p) }
#
# @note Le code fait référence aux variables d'instance @representants et
#       @points pour identifier les points et représentants appropriés.
#
def representants_finaux









end
```

5. Fonctions sur des vecteurs de nombres flottants (6 pts)

Des opérations souvent utilisées lorsqu'on manipule des vecteurs de nombres flottants sont le *SAXPY* (*Single-precision $A \cdot X$ Plus Y*) et le produit scalaire :

- `saxpy` reçoit un scalaire `a` et deux vecteurs `x` et `y` (de même taille), **multiplie** chaque élément `x[i]` par `a` puis **ajoute** le résultat à `y[i]` ;
- `produit_scalaire` reçoit deux vecteurs `x` et `y` (de même taille) et fait la somme du produit des `x[i]` et `y[i]`. (Cette méthode est utilisée, par ex., pour combiner une ligne (`x`) et une colonne (`y`) lors d'un produit matriciel.)

Le code ci-bas (et page suivante) présente deux fonctions en `C` pour ces opérations. Pour chacune de ces fonctions, écrivez une version parallèle en utilisant des constructions PRuby.

Votre solution doit être **aussi simple que possible**, tout en étant intéressante au niveau parallélisme **pour ce problème**. Le cas échéant, vous devez spécifier **de façon explicite** le mode de répartition des tâches.

Remarques : Dans les versions Ruby, la taille des tableaux n'a pas besoin d'être transmise en argument, puisqu'on peut l'obtenir avec `size` — donc pas de paramètre `n`. De plus, vous pouvez aussi supposer que les tableaux à traiter **ont plus d'éléments que le nombre de coeurs/processeurs** (i.e., `n > PRuby.nb_threads`).

```
void saxpy( float a, float x[], float y[], int n )
{
  for( int i = 0; i < n; i++ ) {
      y[i] += a * x[i];
  }
}
```

```
def saxpy( a, x, y )









end
```

```
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;
}
```

```
def produit_scalaire( x, y )









end
```

6. Production d'un index inversé (6 pts)

Selon Wikipedia «un **index inversé** est une correspondance entre du contenu, comme des mots ou des nombres, et sa position dans un ensemble de données comme un enregistrement en base de données, un document ou un ensemble de documents».

Par exemple, soit une liste de `mots = ["mot_1", "mot_2", "mot_3"]` et une liste de `documents = [d1, d2, d3, d4, d5]`. Un appel à `index_inverse( mots, documents )` pourrait produire un résultat tel que le suivant :

```
[["mot_1", ["d1", "d4"]], ["mot_2", ["d2", "d4"]], ["mot_3", []]]
```

(Donc : `"mot_1"` apparaît dans les documents `d1` et `d4` — mais pas dans `d2`, `d3`, `d5` —, `"mot_2"` apparaît dans `d2` et `d4` — mais pas dans `d1`, `d3`, `d5` — et `"mot_3"` n'apparaît dans aucun document.)

Soit une classe `Document` possédant diverses méthodes, dont les deux suivantes :

```
class Document
  attr_reader :nom  # Chaine representant le nom du document

  # Determine si le document contient ou non le mot indique (=> Bool)
  def contient?( mot ); ...; end
end
```

Les méthodes ci-bas définissent différentes façons de produire un tel index inversé à partir d'une série de mots (`Array<String>`) et d'une série de `Documents` (`Array<Document>`).

Pour chaque version : *i*) indiquez si elle produit le bon résultat ; *ii*) si le résultat est bon, indiquez s'il s'agit d'une approche «intéressante» au niveau de l'expression et exploitation du parallélisme; par contre, si le résultat n'est pas correct, indiquez pourquoi.

---

a.
```
def index_inverse( mots, documents )
    res = (0...mots.size).map { |k| [mots[k], []] }

    mots.peach_index do |k|
      documents.each do |doc|
        res[k][1] << doc.nom if doc.contient?(mots[k])
      end
    end

    res
  end
```

b.
```
def index_inverse( mots, documents )
  res = (0...mots.size).map { |k| [mots[k], []] }

  mots.peach_index do |k|
    documents.peach do |doc|
      res[k][1] << doc.nom if doc.contient?(mots[k])
    end
  end

  res
end
```

c.
```
def index_inverse( mots, documents )
  mots.pmap( dynamic: true ) do |mot|
    res = []
    documents.each do |doc|
      res << doc.nom if doc.contient?(mot)
    end

    [mot, res]
  end
end
```

d.
```
def index_inverse( mots, documents )
  mots.pmap do |mot|
    [mot,
     documents.preduce([]) do |res, doc|
       doc.contient?(mot) ? res << doc.nom : res
     end]
  end
end
```

7. Parallélisme de flux : Un pipeline `mystere` (8 pts)

Remarques :

- La méthode `p` affiche sur `STDOUT` l'objet indiqué. Dans le cas d'un `Array`, les crochets et les «,» sont aussi indiqués :

```
>> p [[10], 20, 30]
[[10], 20, 30]
=> [[10], 20, 30]
```

```
>> puts [10, 20]
10
20
=> nil
```

- Si un bloc doit recevoir deux arguments et qu'un tableau de taille 2 est transmis en argument, alors le tableau est automatiquement décomposé en deux arguments :

```
>> [[1, 2], [30, 40]].map { |x| x[0] }
=> [1, 30]
>> [[1, 2], [30, 40]].map { |x, y| x + y }
=> [3, 70]
```

Soit le code présenté à la page suivante, qui crée et lance un pipeline.

a. À coté de chaque processus `p1`, `p2` et `p3` à la page suivante, indiquez **en mots** ce qui est produit par ce processus sur son canal de sortie — i.e., indiquez ce que fait le processus (QUOI?), et non pas les valeurs spécifiques émises sur le canal. Note : On suppose que la source du pipeline est toujours composée de nombre **non négatifs**.

b. Qu'est-ce qui sera imprimé par l'instruction `p res`?

c. Supposons qu'on exécute ce pipeline **avec une grande quantité de données** (i.e., pas juste les 6 valeurs de l'exemple), sur une machine comportant un grand nombre de coeurs et qu'on associe **un *thread* à chaque processus du pipeline**.

(i) Quels seront les processus — y compris parmi `source` et `sink` — qui pourront s'exécuter **en parallèle**? Justifiez brièvement votre réponse.

(ii) Quel est le **degré maximum de parallélisme** de ce programme?

(iii) Pour augmenter le parallélisme, pourrait-on créer plusieurs instances d'un ou plusieurs des processus `p1`, `p2` ou `p3` et obtenir le même résultat final?

```
# Definitions des processus.

p1 = lambda do |cin, cout|                # Ce processus ...
  r = 0
  cin.each do |x|
    r = x if x > r
    cout << [x, r]
  end
  cout.close
end

p2 = lambda do |cin, cout|                # Ce processus ...
  n = 0
  cin.each do |x, r|
    n += 1
    cout << [x, r, n]
  end
  cout.close
end

p3 = lambda do |cin, cout|                # Ce processus ...
  t = 0.0
  cin.each do |x, r, n|
    t += x
    cout << [r, t / n]
  end
  cout.close
end



# Construction et execution du pipeline.
res = []

( PRuby::Pipeline.source([10, 20, 15, 55, 50, 30]) |
  p1 | p2 | p3 |
  PRuby::Pipeline.sink(res)
  ).run

p res
```