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

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

Nom: _____

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

1	2	3	4	5	6	7	Total
/5	/15	/ 5	/10	/5	/5	/5	/50

- Répondez directement sur le questionnaire, en utilisant uniquement **le recto** des feuilles! (Vous pouvez utiliser le verso pour vos **brouillons**.)
- L'examen comporte sept (7) questions pour un total sur 50.

Quelques éléments 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 : La méthode `step` retourne un `Enumerator`, donc ni un `Range`, ni un `Array`. Les méthodes de la bibliothèque `PRuby` (`peach, peach_index, pmap,` etc.) **ne peuvent donc pas être utilisées sur l'objet retourné par** `step`!

- `Array#flatten` retourne un tableau simple («aplati») d'éléments :

```
>> [[10, 20], [10, 40], [], [12], []].flatten
=> [10, 20, 10, 40, 12]
```

- `Enumerable#find` retourne le premier élément qui satisfait un prédicat, `nil` si aucun :

```
>> [10, 20, 30].find { |x| x >= 20 }
=> 20
>> [10, 20, 30].find { |x| x == 0 }
=> nil
```

- *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
```

1. Répartition des éléments d'un tableau avec `peach` (5 pts)

Soit `a` un `Array` de 10 éléments, où la valeur indique le temps requis pour traiter l'élément.

12	3	5	9	7	10	4	6	2	8

Pour chaque appel ci-bas, indiquez *i*) **quelles tâches seront attribuées à chaque *thread*** et *ii*) **quel sera le temps total d'exécution** — en ne tenant compte que des valeurs indiquées plus haut, donc en ignorant les autres surcoûts 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( static: true, nb_threads: 2 ) { ... }`

t_0 :

t_1 :

temps total =

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

t_0 :

t_1 :

temps total =

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

t_0 :

t_1 :

temps total =

2. Méthode `mystere` de la classe `Array` (15 pts)

Soit la méthode `mystere` définie **dans la classe** `Array` (`bornes_de_tranche` : cf. p. 1) :

```
def mystere
  nb_threads = PRuby.nb_threads

  futures = (0...nb_threads).map do |num_thr|
    PRuby.future do
      nb = 0
      bornes_de_tranche(num_thr, size, nb_threads).each do |k|
        nb += 1 if yield( self[k] )
      end

      nb
    end
  end

  futures.map(&:value).reduce(0, :+)
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 { |x| (x / 10).even? }.must_equal


[*1..100].mystere { |x| x <= 20 }.must_equal
```

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

c. Est-ce qu'une méthode `mystere` utilisant du **parallélisme de boucles** (`peach` et/ou `peach_index`) serait une bonne façon de faire? Justifiez brièvement votre réponse.

d. Complétez la méthode `mystere_rec_ij` pour avoir du **parallélisme *fork-join* récursif dichotomique**. Cette méthode ne comporte pas de *seuil* : la **récursion parallèle** se poursuit donc jusqu'au cas de base trivial (le plus simple possible).

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

def mystere_rec_ij( i, j )




end
```

3. Mesures de performances (5 pts)

Soit une machine parallèle *à mémoire partagée* **comportant 16 coeurs/processeurs**. On a mesuré le temps requis par un **programme parallèle** pour traiter un groupe de données avec différents nombres de *threads* (1, 2, 4, 8) et on a obtenu les temps moyens suivants (en ms, calculés pour plusieurs répétitions) :

Nb. threads	**Temps**
1	150
2	60
4	40
8	50

Le temps requis par une **version séquentielle** du programme avec les mêmes données est de 100 ms.

a. Quelle est **l'accélération relative** lorsqu'on utilise 4 *threads*?

b. Quelle est **l'accélération absolue** lorsqu'on utilise 4 *threads*?

c. Quel serait le comportement du programme, quant à **son accélération**, si on l'exécutait sur les mêmes données **mais en utilisant 16 *threads***? Expliquez brièvement pourquoi le programme aurait ce comportement.

d. **(Bonus 2 pts)** En regard de l'accélération **absolue**, pour quel nombre de *threads* obtient-on la meilleure **efficacité** et quelle est cette efficacité (exprimée en **pourcentage**)?

4. Collisions entre planètes d'un système planétaire (10 pts)

L'extrait de Code Ruby 1 présente une extension de la classe `SystemePlanetaire` pour traiter les collisions entre planètes dans un système planétaire — tel qu'illustré en cours et expliqué dans un courriel.

La méthode `traiter_collisions_seq` définit une version **séquentielle** du traitement des collisions entre planètes. Donnez la mise en oeuvre (page 7) des versions parallèles suivantes :

a. Parallélisme *fork/join* à granularité fine — la plus fine possible (sans récursion!).

b. Parallélisme de données — en spécifiant **explicitement le mode de répartition**.

```
def simuler( nb_iterations, dt )
  nb_iterations.times do
    forces = send "calculer_forces_#{mode}"
    send "deplacer_#{mode}", forces, dt
    send "traiter_collisions_#{mode}"
  end
end


def traiter_collisions_seq
  planetes_apres = []
  @planetes.each do |p|
    planetes_apres << traiter_collision_avec(p)
  end

  @planetes = planetes_apres.flatten
  @nb_planetes = @planetes.size
end


def traiter_collision_avec( planete )
  autre_planete_tres_pres = @planetes.find { |p| planete.tres_pres?(p) }

  if autre_planete.nil?
    [planete]
  else
    planete.exploser(autre_planete)
  end
end
```

Extrait de code Ruby 1: Extension de la classe `SystemePlanetaire`.

```
# a.
def traiter_collisions_forkjoin_fin
```

```
  @planetes = planetes_apres.flatten
  @nb_planetes = @planetes.size
end
```

```
# b.
def traiter_collisions_par_donnees
```

```
    @planetes = planetes_apres.flatten
    @nb_planetes = @planetes.size
end
```

5. Calcul de la distance d'édition entre des chaines (5 pts)

Étant donné un tableau de chaines de caractères — où certaines chaines sont courtes, d'autres longues, d'autres très longues — on veut calculer la distance d'édition **entre les paires de chaines adjacentes**, tel que décrit dans la méthode `distances` ci-bas.

```
# Distance d'edition entre deux chaines (Cf. notes de cours: section 9.1).
# Temps d'execution ∈ O( ch1.size × ch2.size )
def distance( ch1, ch2 ); ...; end

def distances( *chaines )
  resultats = Array.new(chaines.size/2)

  (0...chaines.size).step(2).each do |k|
    resultats[k/2] = distance( chaines[k], chaines[k+1] )
  end

  resultats
end
```

Écrivez une version parallèle de la méthode `distances` en utilisant **la façon qui vous semble la plus appropriée étant donné les caractéristiques du problème**. Justifiez brièvement votre choix.

```
def distances( *chaines )









end
```

6. Fonction `mystere` utilisant un sac de tâches (5 pts)

a. Que fait la méthode `mystere` ci-bas, définie dans une extension de la classe `Array`?

b. Est-ce utile d'employer une telle répartition dynamique pour ce problème? Justifiez brièvement votre réponse.

```
class Array
  TAILLE_TACHE = ... # Un fixnum > 0

  def bool_to_int( b );  b ? 1 : 0; end

  def mystere_ij( i, j )
    (i..j-1).reduce(0) { |nb, k| nb + bool_to_int(self[k] == self[k+1]) }
  end

  def mystere
    return 0 if empty?

    nb_threads = PRuby.nb_threads = [PRuby.nb_threads, size].min
    taches = (0...size).step(TAILLE_TACHE).map do |k|
      [k, [size-1, k+TAILLE_TACHE-1].min]
    end

    res = PRuby::TaskBag.run( nb_threads, *taches ) do |sac|
      nb = 0
      sac.each do |i, j|
        nb += mystere_ij( i, j )
        nb += bool_to_int(self[j] == self[j+1]) if j < nb_threads - 1
      end
      nb
    end

    res.reduce(0, :+)
  end
end
```

7. Parallélisme de flux : Un pipeline `mystere` (5 pts)

Remarques préliminaires :

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

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

```
>> puts ["10", "20", "30"]
10
20
30
=> nil
```

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

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

---

a. Qu'est-ce qui sera imprimé par le segment de code présenté à la page suivante?

b. Supposons qu'on exécute ce pipeline sur une machine comportant un grand nombre de coeurs/processeurs et qu'on associe **un *thread* à chaque processus du pipeline**. *a*) Quels seront alors les *threads*/processus qui pourront s'exécuter **en parallèle** («en même temps»)? Justifiez brièvement votre réponse. *b*) Et donc quel est le **degré maximum de parallélisme** de ce programme?

```
BANG = '!!'

a = lambda do |cin, cout|
  cin.each do |x|
    "#{x}".each_char { |c| cout << c }
    cout << BANG; cout << x
  end
  cout.close
end

b = lambda do |cin, cout|
  r = 0
  while (x = cin.get) != PRuby::EOS
    if x == BANG
      cout << [cin.get, r]
      r = 0
    else
      r += x.to_i if x !=  '.'
    end
  end
  cout.close
end

c = lambda do |cin, cout|
  cin.sort.reverse.each { |x| cout << x }
  cout.close
end

d = lambda do |cin, cout|
  cin.each { |x| cout << "#{x[0]} => #{x[1]}" }
  cout.close
end


res = []
( PRuby.pipeline_source( [11.3, 2.83, 0.324, 41.12] ) |
  a | b | c | d |
  PRuby.pipeline_sink(res)
).run

p res
```