Solutions aux exercices du chapitre 5 : Patrons de programmation en PRuby

Hiver 2017

Exercice 5.1 Extraction de parallélisme de `fact`?

Peut-on extraire du parallélisme de cet algorithme, tel que formulé?
(Indice : Arbre d'activation = ?)

Solution :

Non, puisqu'il n'y a qu'un seul appel récursif. On a besoin d'au moins deux (2) tâches pour faire de l'exécution parallèle!

Exercice 5.2 Arbre d'activation pour l'appel à `fact(20, 3)`.

Pour le Programme Ruby 5.4, dessinez l'arbre d'activation des appels de méthodes qui génèrent des *threads* pour l'appel `fact( 20, 3 )`.

Solution :

```
+ fact( 1, 10, 3 )
   + fact( 1, 5, 3 )
      + fact( 1, 3, 3 )
      + fact( 4, 5, 3 )
   + fact( 6, 10, 3 )
      + fact( 6, 8, 3 )
      + fact( 9, 10, 3 )
+ fact( 11, 20, 3 )
   + fact( 11, 15, 3 )
      + fact( 11, 13, 3 )
      + fact( 14, 15, 3 )
   + fact( 16, 20, 3 )
      + fact( 16, 18, 3 )
      + fact( 19, 20, 3 )
```

Exercice 5.3 Fonction récursive parallèle pour calculer `fact(n)`, mais avec le *thread* parent qui fait du travail utile.

Le Programme Ruby 5.5 crée deux (2) `future`s pour évaluer les deux appels récursifs en parallèle.

Peut-on améliorer ce programme pour créer des *threads* de granularité plus grossière — donc réduire le nombre de *threads* créés — tout en restant autant parallèle?

Indice : Que fait le *thread* parent pendant que ses enfants — les deux appels récursifs — s'exécutent?

Solution :

Le *thread* parent est inactif pendant que ses enfants s'exécutent. On a donc un *thread* qui en crée deux autres, alors qu'un seul suffirait, puisque le *thread* parent pourrait faire du travail utile en solutionnant l'un des sous-problème.

```
def fact( n, seuil )
  def fact_( i, j, seuil )
    # Probleme simple.
    return (i..j).reduce(&:*) if j - i <= seuil

    # Probleme complexe.
    mid = (i+j) / 2
    r1 = PRuby.future { fact_(i, mid, seuil)   }
    r2 = fact_(mid+1, j, seuil)

    r1.value * r2
  end

  fact_( 1, n, seuil )
end
```

Exercice 5.4 Arbre d'activation des *threads* pour l'appel à `fact(20, 3)` avec version améliorée de `fact`.

Dessinez l'arbre d'activation des *threads* pour l'appel `fact( 20, 3 )` pour la version améliorée de `fact` produite pour l'exercice précédent.

Solution :

```
+ fact( 1, 10, 3 )
  + fact( 1, 5, 3 )
    + fact( 1, 3, 3 )
  + fact( 6, 8, 3 )
+ fact( 11, 15, 3 )
  + fact( 11, 13, 3 )
+ fact( 16, 18, 3 )
```

Exercice 5.5 Performances si deux gros tableaux?

Est-ce que cette méthode sera performante si on traite deux gros tableaux?

Solution :

Non : trop de *threads* de granularité trop fine — avec très peu d'instructions à exécuter!

Exercice 5.6 Sommation des éléments d'un tableau avec parallélisme récursif.

Écrivez une méthode `sommation_tableau` qui reçoit en argument un tableau `a` (un `Array`) composé de nombres (`Numeric`) et qui retourne la somme de ces nombres, par exemple :

```
sommation_tableau( [] ).
  must_equal 0

sommation_tableau( [99] ).
  must_equal 99

sommation_tableau( [1, 20, 300, 4000] ).
  must_equal 4321
```

De plus, cette méthode doit utiliser du *parallélisme récursif* — approche diviser-pour-régner dichotomique — et doit utiliser la construction `PRuby.pcall` ou `PRuby.future`.

Notez qu'il n'est pas nécessaire d'introduire de troncation de la récursion (avec un seuil). Vous pouvez donc diviser jusqu'au cas de base *trivial*.

Solution :

```ruby
def sommation_tableau_rec( a )
  return 0 if a.size == 0

  sommation_tableau_rec_ij( a, 0, a.size-1 )
end

def sommation_tableau_rec_ij( a, i, j )
  return a[i] if i == j # Cas de base.

  # Cas recursif.
  mid = (i + j) / 2

  r1 = PRuby.future { sommation_tableau_rec_ij(a, i, mid) }
  r2 = sommation_tableau_rec_ij(a, mid+1, j)

  r1.value + r2
end
```

Exercice 5.7 Sommation des éléments d'un tableau avec parallélisme récursif et utilisation de *tranches* de tableaux.

Soit la méthode suivante qui se veut une solution à l'exercice précédent :

```
def sommation_tableau( a )
  return 0 if a.size == 0
  return a[0] if a.size == 1

  mid = a.size / 2
  r1 = PRuby.future { sommation_tableau(a[0..mid-1]) }
  r2 = sommation_tableau( a[mid...a.size] )
  r1.value + r2
end
```

Que peut-on dire de cette solution?

Solution :

Elle va produire le bon résultat, et de façon relativement efficace quant à la création des *threads*, puisqu'un seul `future` est créé, le *thread* parent s'occupant de l'une des tâches.

Toutefois, elle contient une source d'inefficacité : **chaque tranche de tableau utilisée comme argument à un appel récursif entraine la création d'une copie du sous-tableau** ☹

Ce comportement des tranches de tableau passées en argument est illustrée dans les exemples suivants :

```
def mettre_a_zero_indices( a, indices )
  indices.each { |i| a[i] = 0 }
end

def mettre_a_zero_tranche( a )
  (0...a.size).each { |i| a[i] = 0 }
end

a = [10, 20, 30, 40, 50]
mettre_a_zero_indices( a, 2..3 )      # a est passe par reference...
DBC.assert a == [10, 20, 0, 0, 50]    # donc est modifie

mettre_a_zero_tranche( a[2..3] )      # Cree une copie de la tranche...
DBC.assert a == [10, 20, 30, 40, 50]  # donc a n'est pas modifie.
```

Exercice 5.8 Sommation des éléments d'un tableau avec parallélisme itératif à granularité grossière.

Comme dans l'exercice précédent, écrivez une méthode `sommation_tableau` qui reçoit en argument un tableau `a` composé de nombres et qui retourne la somme de ces nombres.

Toutefois, cette méthode doit utiliser du *parallélisme itératif à granularité* **grossière** et doit utiliser la construction `PRuby.pcall`.

Pour simplifier, vous pouvez supposer que **le nombre d'eléments du tableau est divisible par le nombre de *threads*.** Vous pouvez donc utiliser la fonction suivante :

```
def bornes_tranche( k, n, nb_threads )
  b_inf = k * n / nb_threads
  b_sup = (k+1) * n / nb_threads - 1
  b_inf..b_sup
end
```

Solution :

```
def sommation_seq( a, bornes_tranche )
  bornes_tranche.reduce(0) { |somme, k| somme + a[k] }
end

def sommation_tableau( a )
  nb_threads = PRuby.nb_threads
  DBC.require a.size % nb_threads == 0

  r = Array.new(nb_threads)
  PRuby.pcall (0...nb_threads),
    lambda do |k|
      bornes = bornes_tranche(k, a.size, nb_threads)
      r[k] = sommation_seq( a, bornes )
    end

  r.reduce(:+)
end
```

Exercice 5.9 Nombre de *threads* pour deux matrices.

Supposons `a` de taille $n_1 \times n_2$ et `b` de taille $n_2 \times n_3$.
Combien de *threads* seront créés?
Est-ce une bonne idée?

Solution :

$n_1 \times n_3$ *threads* seront créés, ce qui n'est clairement pas une bonne idée si n_1 et n_3 sont grands.

Exercice 5.10 Méthode `pmax` pour la classe `Ensemble`.

Soit la classe `Ensemble` traitée dans le labo #1. dont voici une version possible :

```
class Ensemble
  def initialize( *elements )
    @elements = []
    elements.each do |x|
      @elements << x unless @elements.include? x
    end
  end

  def max
    fail "L'ensemble est vide" if cardinalite == 0

    m = @elements[0]
    @elements.each do |x|
      m = [m, x].max
    end
    m
  end
  ...
end
```

On veut une version parallèle de cette méthode — `pmax`.

1. Peut-on simplement remplacer `each` par `peach` pour obtenir une version **avec parallélisme de boucles**?

2. Si on veut utiliser du **parallélisme de données**, à quoi ressemblerait le code de `pmax`?

Solution :

1. Non, car il y aurait une possibilité de situation de compétition entre les *threads* pour accéder à la variable `m`.
2. On utilise `preduce`.

```
def pmax
  fail "L'ensemble est vide" if cardinalite == 0

  @elements.preduce(@elements[0]) { |m, x| [m, x].max }
end
```

Exercice 5.11 Méthode `pselect` sur une collection de type `Array`.

Donnez une mise en oeuvre d'une méthode `pselect` qui est version **parallèle** de `select`.

```
>> a = [*1..10]
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> a.pselect { |x| x.even? }
=> [2, 4, 6, 8, 10]

>> a.pselect { |x| x > 9 }
=> [10]

>> a.pselect { |x| x < 0 }
=> []
```

Votre mise en oeuvre doit être aussi parallèle que possible... *bien qu'une partie puisse être faite de façon séquentielle* — difficile de faire autrement ☹

Hypothèse/indice :

- On suppose que c'est **l'évaluation du prédicat sur un élément** qui est coûteuse (longue à exécuter).
- Faites le travail en deux passes : une première parallèle, l'autre séquentielle.
- La méthode `compact` supprime les `nil` :

  `[10, 20, nil, 99, nil].compact == [10, 20, 99]`

Solution :
Solution en deux passes — `compact` supprime les éléments `nil`, i.e., ceux pour lesquels `yield x` était faux :

```
def pselect
  pmap { |x| x if yield x }
    .compact
end
```

Autre solution semblable :

```
def pselect
  res = pmap { |x| x if yield x }
  res.compact!

  res
end
```

Solution récursive :

```
def pselect_rec( i, j, &predicat )
  return predicat.call(self[i]) ? [self[i]] : [] if i == j

  m = (i + j) / 2
  res1 = PRuby.future { pselect_rec( i, m, &predicat ) }
  res2 = pselect_rec( m+1, j, &predicat )
  res1.value.concat res2
end

def pselect( &predicat )

  return [] if empty?

  pselect_rec( 0, size-1, &predicat )
end
```

Exercice 5.12 Temps d'exécution et accélération pour des tâches de temps variable.

Supposons une répartition statique des tâches entre quatre (4) *threads* où chaque *thread* est exécuté par un processeur indépendant et où le temps requis pour que chaque *thread* traite son groupe de tâches est comme suit — l'unité de mesures n'a pas d'importance :

- *Thread 0 :* 10
- *Thread 1 :* 40
- *Thread 2 :* 20
- *Thread 3 :* 20

1. Quel sera le temps total d'exécution?
2. Quelle sera l'accélération?
3. Quelle pourrait être la meilleure accélération **si on réussisait à bien répartir le travail**?

Solution :

1. Le temps total d'exécution sera alors... de 40 unités, et ce même si le premier *thread* a complété après 10 unités et les autres après 20.

   Donc, pendant une majeure partie du temps requis pour exécuter le programme, trois des processeurs ne feront rien ☹

2. L'accélération sera donc de 90 / 40 = 2.25.

3. La meilleure accélération pourrait être de 4.

Exercice 5.13 Le défaut de la solution statique pour `wc` ☹

Quel est le principal défaut du Programme Ruby 5.22?

Solution :

Dans la méthode `wc`, les différents fichiers (`fichs`) seront répartis uniformément entre les *threads*. C'est-à-dire que chaque *thread* aura **le même nombre de fichiers à traiter**. Or, ces fichiers pourraient être de tailles très variables. La quantité de travail fait par chaque *thread* pourrait donc varier grandement.

Exercice 5.14 Fonction `mystere`.

Que fait la fonction `mystere`? Quelle stratégie de programmation est utilisée?

Solution :

Calcul de `n!` avec une approche diviser-pour-régner dichotomique, mais **sans récursivité**.

C'est le sac de tâches qui contient les informations identifiant les sous-problèmes à résoudre : un `Range` indique les bornes de l'intervalle de nombres (consécutifs) à multiplier entre eux.

Un *thread*, via le `each`, prend une tâche, la décompose en deux sous-tâches jusqu'à ce que la sous-tâche gauche soit suffisamment simple pour la résoudre directement, alors que la sous-tâche droite est ajoutée le sac de tâches.

Remarque : On verra bientôt qu'une approche semblable — approche diviser-pour-régner dichotomique mais sans récursion — est utilisée avec les *Threading Building Blocks* d'Intel.

Exercice 5.15 Version séquentielle de la transformation de Jackson.

(À faire si vous avez *beaucoup* (beaucoup !) de temps ☺)
Écrivez une version *séquentielle*, en Java, de la méthode `transformer_jackson`.

Solution :

Solution omise ☺

Très compliqué à mettre en oeuvre dans un langage purement séquentiel!

Exercice 5.16 Programme `mystere` avec pipeline.

Qu'est-ce qui sera imprimé par le programme suivant?

```
foo = lambda do |cin, cout|
  cin.each do |x|
    cout << x if x % 2 == 0
  end
  cout.close
end

bar = lambda do |cin, cout|
  while (x = cin.get) != PRuby::EOS
    y = cin.get
    cout << y if y != PRuby::EOS
    cout << x
  end
  cout << x
  cout.close
end

baz = lambda do |cin, cout|
  cin.each do |x|
    cout << x
    cout << x
  end
  cout.close
end

res = []

(PRuby.pipeline_source([*0..8]) |
 foo |
 bar |
 baz |
 PRuby.pipeline_sink(res))

puts res.inspect
```

Solution :

`[]` : le pipeline est créé, mais son exécution n'est pas lancée, donc le tableau `res` n'est pas modifié ☺

Par contre, si on ajoute «`.run`» après la création du pipeline, alors le résultat suivant sera produit :

```
[2, 2, 0, 0, 6, 6, 4, 4, 8, 8]
```

Exercice 5.17 Somme avec `peach`.

Qu'est-ce qui sera imprimé par le programme ci-bas.

```
$ cat peach.rb
require 'pruby'

N = 100

a = [*1..N]   # a = [1, 2, 3, 4, ..., N]

total = 0
a.peach do |x|
  total += x
end

puts "total = #{total}"
```

Solution :

```
$ ruby peach.rb
total = 5034

$ ruby peach.rb
total = 5050

$ ruby peach.rb
total = 5050

$ ruby peach.rb
total = 4799

$ ruby peach.rb
total = 4821
```

Pourquoi?

Les méthodes `peach` et `peach_index` ne doivent être utilisées que si les itérations sont indépendantes les unes des autres!

Exercice 5.17 Modes de répartition des tâches.

Soit un tableau `a` de 12 éléments, où la valeur en rouge indique le temps requis pour traiter cet élément.

10	20	30	40	50	100	200	50	40	30	20	10

Supposons qu'on ne considère que les temps indiqués, donc en ignorant les autres surcoûts d'exécution.

Pour chaque appel ci-bas, indiquez **quelles tâches seront attribuées à chaque *thread*** et quel sera le temps total d'exécution.

Note : On suppose que les *threads* obtiennent les tâches dans l'ordre de priorité de leur numéro — donc le premier *thread* obtient la première tâche, etc., puis par la suite si deux *threads* veulent une tâche «en même temps», alors c'est le *thread* avec le plus petit numéro qui obtient une tâche en priorité.

1. `a.peach( static: true, nb_threads: 3 ) { ... }`
2. `a.peach( static: 1, nb_threads: 3 ) { ... }`
3. `a.peach( dynamic: true, nb_threads: 3 ) { ... }`

   Note : «`dynamic: true`» = «`dynamic: 1`»

Solution :

Les couleurs différentes indiquent un traitement par des *threads* différents.

1. `(static: true)`

10	20	30	40	50	100	200	50	40	30	20	10

   Temps total = 400

   `[10+20+30+40, 50+100+200+50, 40+30+20+10].max`

   Donc, pendant 300 unités de temps, deux des *threads* ne feront qu'attendre que le troisième *thread* termine.

2. `(static: 1)`

10	20	30	40	50	100	200	50	40	30	20	10

   Temps total = 280

   `[10+40+200+30, 20+50+50+20, 30+100+40+10].max` =

   `[280, 140, 180].max`

3. `(dynamic: true)`

10	20	30	40	50	100	200	50	40	30	20	10

   Temps total = 250

   `[10+40+200, 20+50+50+40+20, 30+100+30+10].max` =

   `[250, 180, 170].max`

Exercice 5.18 Méthode `mystere` de la classe `Array`

Soit la méthode suivante définie dans la classe `Array` :

```
class Array
  def mystere
    res = Array.new( size )
    PRuby.pcall 0...size,
      ->(k) { res[k] = self[size-k-1] }

    res
  end
end
```

1. Que fait cette méthode? Quel nom plus significatif peut-on lui donner?
2. Écrivez une version équivalente de cette méthode, mais qui utilise plutôt du **parallélisme de boucles** — donc avec `peach` ou `peach_index`.
3. Même question, mais avec du **parallélisme de données** — plus spécifiquement avec `pmap`.
4. Même question, toujours avec du **parallélisme de données**, mais cette fois avec `preduce`.

   Note : Cette dernière méthode n'est pas triviale... et il faut utiliser l'argument (optionnel) `final_reduce` de la méthode `preduce`.

Solution :

1. Elle produit un tableau avec les éléments dans l'ordre inverse = `preverse`.

2. ```
   def preverse
     res = Array.new( size )

     (0...size).peach do |k|
       res[k] = self[size-k-1]
     end

     res
   end
   ```

3. ```
   def preverse
     (0...size).pmap { |k| self[size-k-1] }
   end
   ```

4. ```
   def preverse
     preduce([],
              final_reduce: ->(x, y) { y + x }) do |a, x|
     [x] + a
     end
   end
   ```

   ```
     def preverse
       preduce([],
                final_reduce: ->(x, y) { y.concat x }) do |a, x|
         a.empty? ? [x] : a.unshift(x)
       end
     end
   ```
   ```

Exercice 5.20 Exécution parallèle, ou non, d'un pipeline.

1. Dans le pipeline de la Figure 5.17, est-ce que les processus `generer_mots` et `filtrer_mots_invalides` pourraient s'exécuter en parallèle?

2. Dans le pipeline de la Figure 5.17, est-ce que les processus `filtrer_mots_invalides` et `supprimer_doublons` pourraient s'exécuter en parallèle?

3. Quel est le degré maximum de parallélisme de ce pipeline?

Solution :

1. Oui, le premier processus pourrait être en train de traiter des lignes qui viennent après d'autres lignes qui ont déjà été décomposées en mots et pour lesquelles le filtrage des mots invalides est en train de s'effectuer.

2. Non. Le processus `trier` doit avoir reçu **tous les mots <u>avant</u>** de pouvoir commencer à émettre, sur son canal de sortie, la liste ordonnée des mots — le dernier mot du fichier d'entrée est peut-être (?) celui qui vient en premier dans l'ordre alphabétique. Lorsque `supprimer_doublons` débutera véritablement son travail, le processus `filtrer_mots_invalides` (et `generer_mots`) aura donc terminé son travail.

   Il y a donc une différence importante entre `trier` et les autres processus : alors que les autres processus peuvent commencer à émettre sur le canal de sortie des éléments avant même que le flux du canal d'entrée soit complet et terminé, le processus `trier` ne peut commencer à émettre sur son canal de sortie *que lorsque le flux du canal d'entrée est terminé (canal fermé!)*.

3. Les processus de chacun des groupes suivants peuvent s'exécuter en parallèle :

   ```
   pipeline_source(fich_entree) |
   generer_mots |
   filtrer_mots_invalides |
   trier

   trier |
   suppprimer_doublons |
   pipeline_sink(fich_sortie)
   ```

   Donc, degré maximum de parallélisme = 4.

   Notons que `trier` peut s'exécuter en parallèle avec les deux groupes de processus parce qu'il s'exécutera en deux phases :

   (a) Avec le premier groupe pour lire les valeurs à trier.
   (b) Avec le deuxième groupe pour émettre, de façon incrémentale, les valeurs triées une fois que le résultat de l'appel à `sort` est obtenu via le `each`.